Marek Sokołowski

Efficient Data Structures And Graph Width Parameters

Doctoral dissertation

University of Warsaw

Faculty
of Mathematics, Informatics
and Mechanics

Supervisor:
dr hab. Michał Pilipczuk

Institute of Informatics
University of Warsaw

May 2024

Supervisor's statement

Hereby I confirm that the presented thesis was prepared under my supervision and that it fulfils the requirements for the degree of Doctor in the field of Natural Sciences in the discipline of Computer and Information Sciences.

Date	Signature

Author's statement

Hereby I declare that the presented thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date	Signature

Oświadczenie kierującego pracą

Oświadczam, że niniejsza praca została przygotowana pod moim kierunkiem i stwierdzam, że spełnia ona warunki do przedstawienia jej w postępowaniu o nadanie stopnia doktora w dziedzinie nauk ścisłych i przyrodniczych w dyscyplinie informatyka.

Data	Podpis

Oświadczenie autora pracy

Oświadczam, że niniejsza rozprawa doktorska została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem stopnia doktora w innej jednostce.

Niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data	Podpis

Abstract

This dissertation presents several contributions in structural graph theory: the study of combinatorial properties of various classes of graphs and algorithmic applications of these properties. We focus on families of graphs characterized by *graph width parameters*: integer values measuring how efficiently a graph can be decomposed into simpler pieces following a set of prescribed rules. We present research related to three such width parameters – *treewidth*, *rankwidth*, and *twin-width*. We show evidence that it is possible to construct efficient data structures for classes of graphs with bounded values of these parameters, enabling us to answer a variety of queries about the input graphs, as well as apply certain types of updates to the graphs.

In Chapter 3 we show the results of the article "Dynamic treewidth" [KMN+23]. We prove that given a dynamic n-vertex graph of small treewidth k undergoing edge insertions and removals in the fully dynamic model, we can maintain a *tree decomposition* of the graph witnessing at each point of time that the graph has treewidth at most $6k+5$. A single update is processed by our data structure in subpolynomial amortized time $\mathcal{O}_k(2^{\sqrt{\log n}\log\log n})$; here, the $\mathcal{O}_k(\cdot)$ notation hides factors depending only on k. Next, we show that most dynamic programming schemes applicable to tree decompositions in the usual static setting can also be maintained dynamically within the same time bounds. As an application, we show a dynamic variant of the classic Courcelle's theorem [Cou90]: The satisfaction of a fixed property of graphs expressible in the monadic second-order logic with edge subset quantification (CMSO_2) can be tracked in dynamic graphs of treewidth at most k, also within the same time complexity bounds.

Then in Chapter 4, based on the work "Almost-linear time parameterized algorithm for rankwidth via dynamic rankwidth" [KS24], we generalize the dynamic treewidth data structure to a much more general graph parameter, rankwidth. For a dynamic n-vertex graph updated by edge deletions and additions whose rankwidth is at most k at all times, we maintain a *rank decomposition* of the graph of width at most $4k$, again in amortized time $\mathcal{O}_k(2^{\sqrt{\log n}\log\log n})$. Since rankwidth generalizes the notion of treewidth to the setting of dense graphs, we also introduce a framework of *dense updates* to the graph, allowing us to redefine the adjacencies using a formula of monadic second-order logic within any chosen t-vertex subgraph of the maintained graph in time almost linear in t rather than t^2.

As a consequence, we give an algorithm that for an n-vertex, m-edge graph and an integer k, in time $\mathcal{O}_k(n^{1+o(1)})+\mathcal{O}(m)$ either produces a rank decomposition of the graph of width at most k, or correctly determines that the rankwidth of the graph exceeds k. In the positive case, our algorithm can also return a graph expression witnessing that the cliquewidth of the graph is at most $2^{k+1}-1$. This improves upon a previous result of Fomin and Korhonen, who achieved $\mathcal{O}_k(n^2)$ time algorithm for the problem. Moreover, given a fixed property of graphs encoded in the monadic second-order logic *without* edge subset quantification (CMSO_1), we can verify whether the property holds in a graph of rankwidth at most k within the same time bounds.

Chapter 5 is based on the results from the work "Fully dynamic approximation schemes on planar and apex-minor-free graphs" [KNPS24] and presents dynamic counterparts of the classic approximation schemes for Maximum Weighted Independent Set and Minimum Weighted Dominating Set in planar graphs, described in the static setting by Baker [Bak94]. For a fully dynamic vertex-weighted planar graph G, updated by edge insertions and removals, as well as weight modifications, and a fixed real value $\varepsilon>0$, we maintain the following summaries of the graph:

- a $(1-\varepsilon)$-approximation of the maximum weight of an independent set in time $\mathcal{O}_\varepsilon(n^{o(1)})$, and
- a $(1+\varepsilon)$-approximation of the minimum weight of a dominating set in time $\mathcal{O}_{\varepsilon,\Delta}(n^{o(1)})$ under the additional assumption that at all times, all vertices of the graph have degrees bounded by Δ.

The techniques heavily rely on the structural properties of graphs of bounded treewidth. The same techniques are also applicable to arbitrary classes of graphs excluding a fixed apex graph as a minor – such as the class of toroidal graphs or any class of graphs embeddable in a fixed surface – so the algorithmic results above lift also to these classes.

Next, Chapter 6 displays the findings from our work "Compact representation for matrices of bounded twin-width" [PSZ22]. Namely, we propose a data structure that stores a static binary d-twin-ordered $n\times n$ matrix and can be queried for its entries. The data structure requires $\mathcal{O}_d(n)$ bits of space – which is asymptotically optimal due to a result of Bonnet et al. [BGdM+21] – and can answer every query in worst-case $\mathcal{O}(\log\log n)$ time.

Finally, in Chapter 7 we present our work "Graphs of bounded twin-width are quasi-polynomially χ-bounded", where we prove that, for every integer d, there exists a quasi-polynomial function $f_d(\omega) \in 2^{\mathcal{O}(\log^{4d+3}\omega)}$ such that every graph of twin-width at most d and maximum clique size ω can be properly colored using at most $f_d(\omega)$ colors. This comes close to a positive resolution of a conjecture of Bonnet et al. [BGK+21b], claiming that graphs of bounded twin-width are *polynomially* χ-bounded.

Streszczenie

W niniejszej rozprawie prezentujemy wyniki badań nad strukturalną teorią grafów: dziedziną nauki analizującą właściwości różnych klas grafów oraz algorytmiczne zastosowania tych własności. Skupiamy się na klasach grafów, które scharakteryzowane są *parametrami szerokości grafów*: całkowitoliczbowymi wartościami mierzącymi, jak efektywnie dany graf można rozłożyć na prostsze części zgodnie z zadanym zbiorem reguł. Prezentujemy wyniki badań dotyczących trzech parametrów tego typu: *szerokości drzewiastej* (treewidth), *szerokości rzędowej* (rankwidth) oraz *szerokości bliźniaczej* (twin-width). Pokazujemy, że jesteśmy w stanie tworzyć wydajne struktury danych dla klas grafów o ograniczonych wartościach tych parametrów. Takie struktury danych pozwalają na efektywne odpowiadanie na zapytania dotyczące zróżnicowanych własności grafów wejściowych, jak również na aplikowanie różnych typów przekształceń tych grafów.

W Rozdziale 3 prezentujemy wyniki pochodzące z artykułu „Dynamic treewidth" [KMN+23]. Wykazujemy, że dla dowolnego n-wierzchołkowego grafu o małej szerokości drzewiastej k, aktualizowanego w pełni dynamicznie poprzez dodawanie i usuwanie krawędzi, możemy utrzymywać *dekompozycję drzewiastą* grafu świadczącą o tym, że szerokość drzewiasta grafu nie przekracza $6k + 5$. Pojedyncza aktualizacja grafu jest przetwarzana przez naszą strukturę danych w podwielomianowym czasie zamortyzowanym $\mathcal{O}_k(2^{\sqrt{\log n}\log\log n})$; notacja $\mathcal{O}_k(\cdot)$ ukrywa czynniki zależne jedynie od k. Pokazujemy również, że większość schematów programowania dynamicznego na dekompozycjach drzewiastych, zaprojektowanych dla statycznych grafów, można utrzymywać również w dynamicznych grafach – w tej samej, podwielomianowej złożoności czasowej. Zastosowaniem tego wyniku jest dynamiczna wersja klasycznego twierdzenia Courcelle'a [Cou90]: w tej samej złożoności czasowej, nasza struktura danych może utrzymywać w dynamicznym grafie o szerokości drzewiastej nieprzekraczającej k informację o tym, czy graf spełnia dowolną ustaloną własność grafów wyrażoną w języku monadycznej logiki drugiego rzędu z kwantyfikacją po podzbiorach wierzchołków (CMSO_2).

Następnie, w Rozdziale 4, bazującym na pracy „Almost-linear time parameterized algorithm for rankwidth via dynamic rankwidth" [KS24], opisujemy podobną strukturę danych dla znacznie ogólniejszego parametru grafowego – szerokości rzędowej. Dla dynamicznego n-wierzchołkowego grafu o szerokości rzędowej co najwyżej k, aktualizowanego poprzez usuwanie i dodawanie krawędzi, utrzymujemy *dekompozycję rzędową* o szerokości co najwyżej $4k$, ponownie w zamortyzowanym czasie $\mathcal{O}_k(2^{\sqrt{\log n}\log\log n})$. Ponieważ szerokość rzędowa jest uogólnieniem szerokości drzewiastej do grafów gęstych, wprowadzamy model *gęstych aktualizacji* grafu, pozwalający na przedefiniowanie sąsiedztw w dowolnym wybranym t-wierzchołkowym podgrafie utrzymywanego grafu za pomocą formuł monadycznej logiki drugiego rzędu w czasie zależnym prawie liniowo od t zamiast t^2.

Wykorzystując powyższe wyniki, prezentujemy algorytm, który wczytuje graf mający n wierzchołków i m krawędzi oraz liczbę naturalną k i który w złożoności czasowej $\mathcal{O}_k(n^{1+o(1)}) + \mathcal{O}(m)$ konstruuje dekompozycję rzędową grafu o szerokości co najwyżej k lub poprawnie stwierdza, że szerokość rzędowa grafu jest ściśle większa niż k. W pozytywnym przypadku nasz algorytm zwraca również wyrażenie grafowe poświadczające, że *szerokość klikowa* grafu jest ograniczona z góry przez $2^{k+1} - 1$. Algorytm ten poprawia poprzedni wynik Fomina oraz Korhonena, którzy zaprojektowali dla tego samego problemu algorytm w złożoności $\mathcal{O}_k(n^2)$. Ponadto, w tej samej złożoności czasowej możemy sprawdzić, czy graf spełnia ustaloną własność grafów zapisaną w monadycznej logice drugiego rzędu *bez* kwantyfikacji po podzbiorach krawędzi (CMSO_1).

Rozdział 5 bazuje na pracy „Fully dynamic approximation schemes on planar and apex-minor-free graphs" [KNPS24]. W tym rozdziale pokazujemy dynamiczne warianty klasycznych schematów aproksymacyjnych dla problemów maksymalnego ważonego zbioru niezależnego oraz minimalnego ważonego zbioru dominującego w grafach planarnych, rozważanych w modelu statycznych grafów przez Baker [Bak94]. Dokładniej, dla dowolnej ustalonej rzeczywistej wartości $\varepsilon > 0$, tworzymy strukturę danych utrzymującą następujące zbiorcze informacje na temat dynamicznego grafu planarnego G z rzeczywistymi wagami wierzchołków:

- $(1-\varepsilon)$-aproksymację maksymalnej wagi zbioru niezależnego w G w złożoności czasowej $\mathcal{O}_\varepsilon(n^{o(1)})$; oraz
- $(1+\varepsilon)$-aproksymację minimalnej wagi zbioru dominującego w G w złożoności czasowej $\mathcal{O}_{\varepsilon,\Delta}(n^{o(1)})$ przy dodatkowym założeniu, że stopnie wszystkich wierzchołków grafu G są w każdym momencie ograniczone przez Δ.

Metody używane przez nas w projektowaniu tej struktury danych wykorzystują wielorakie własności grafów o ograniczonej szerokości drzewiastej. Te same metody działają nie tylko dla klasy grafów planarnych,

lecz również dla dowolnych klas grafów niezawierających ustalonego *grafu szczytowego* (*apex graph*) jako minora – na przykład dla klasy grafów toroidalnych czy dla dowolnej klasy grafów zanurzalnych w ustalonej powierzchni. Tak więc wymienione wyżej wyniki działają również dla tych klas grafów.

Dalej, w Rozdziale 6 pokazujemy wyniki pochodzące z pracy „Compact representation for matrices of bounded twin-width" [PSZ22]. Dokładniej, pokazujemy zwięzłą strukturę danych, która utrzymuje statyczną zero-jedynkową macierz $n \times n$, która jest *d-uporządkowana bliźniaczo* (*d-twin-ordered*) i którą można odpytywać o poszczególne elementy macierzy. Struktura danych zajmuje $\mathcal{O}_d(n)$ bitów pamięci – asymptotycznie optymalną liczbę bitów zgodnie z wynikiem Bonneta i in. [BGdM+21] – i zwraca odpowiedź na dowolne zapytanie w czasie $\mathcal{O}(\log \log n)$.

Wreszcie, w Rozdziale 7 prezentujemy pracę „Graphs of bounded twin-width are quasi-polynomially χ-bounded", gdzie pokazujemy, że dla każdej liczby naturalnej d istnieje quasi-wielomianowa funkcja $f_d(\omega) \in 2^{\mathcal{O}(\log^{4d+3} \omega)}$ o następującej własności: każdy graf o szerokości bliźniaczej nieprzekraczającej d i rozmiarze maksymalnej kliki równej ω ma liczbę chromatyczną ograniczoną z góry poprzez $f_d(\omega)$. Wynik ten jest bliski pozytywnemu rozstrzygnięciu hipotezy Bonneta i in. [BGK+21b] twierdzącej, że grafy o ograniczonej szerokości bliźniaczej są *wielomianowo* χ-ograniczone – to znaczy, funkcja $f_d(\omega)$ w opisie powyżej jest wielomianowa względem ω.

Acknowledgements

First and foremost, I would like to sincerely thank my supervisor Michał Pilipczuk for becoming my mentor for the last four years: showing me how to do my best in research; sharing with me his vast knowledge about graph theory, parameterized algorithms, and logic; infecting me and everyone around with enthusiasm about our research; and being an endless source of open problems to tackle. This dissertation would have never become a real thing without you. Too bad this thesis does not include any proofs involving your favorite combinatorial trick[1] – the factorization forest theorem of Simon... I hope you can live with this.

Next, I am grateful to my fellow current and past PhD students working with algorithms and graph theory in Warsaw – Łukasz Bożyk, Konrad Majewski, Jana Masaříková, Karolina Okrasa, Wojtek Nadara, Wojtek Przybyszewski and Marcin Smulewicz. You all made my doctoral studies here in Warsaw joyful and satisfying, and without you, my first steps in academia would be much more stressful and daunting.

I would then like to acknowledge all my collaborators in past and present projects. Thank you for showing me how diverse our area of research is, and thank you for demonstrating that inspiration for research can come from the most unexpected places.

I would have probably never become a researcher if not for sports programming – a type of mind sport where competitors attempt to solve algorithmic and coding puzzles as quickly as possible. Firstly, I would like to thank the high school activity club in Białystok for introducing me to this kind of contests. Then I extend my thanks to all my teammates in team programming competitions in Warsaw – in particular Wojtek Nadara and Marcin Smulewicz, with whom I developed my algorithmic skills the most. I believe that every single tough graph problem and every single difficult data structure problem we solved at a team contest brought me one step closer to completing this dissertation.

Last but not least, I would like to say a huge thank you to my family, who supported me emotionally throughout my entire doctoral studies. Even though you may still not fully grasp what "bounded treewidth" or "quasi-polynomially χ-bounded" means, you have somehow always managed to understand what I was struggling with and continued to give me moral support. Thank you again for this.

Attribution of support. The results presented in this dissertation were part of projects that have received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, grant agreement No. 948057 (BOBR).

Attribution of graphics. Most of the figures present in this work are sourced from the original publications [KMN+23, KS24, PSZ22, PS23]. Figures 3.1 and 3.2 have been created by Tuukka Korhonen. The remaining figures have been prepared by the author of the dissertation.

[1] Citation needed.

Contents

Chapter 1

Introduction

It has been long known that some computational problems are harder than others. Some, such as the *halting problem*, are simply undecidable – there is provably no foolproof method that can tell apart terminating and nonterminating computer programs. However, even when we restrict our attention to solvable problems, we find problems of wildly varying complexity. For some – such as Maximum Matching – we can design algorithms correctly resolving any possible instance *efficiently* – that is, in time polynomial in the size of the input [Edm65]. On the other hand, the time hierarchy theorem [HS65] asserts that some decidable problems do not admit such algorithms. In fact, we can explicitly name computational problems for which this is the case, such as the evaluation of positions in generalized variants of chess and Go [FL81, Rob83] or testing reachability in Vector Addition Systems [CO21].

For many decades, computer scientists have focused their attention on the complexity a particular kind of solvable problems: the yes/no *decision problems* whose solutions are *efficiently verifiable*. So for instance, in the Satisfiability problem, we are given a Boolean formula φ with variables $x_1, \ldots, x_n$, and we are to decide whether some assignment of true/false values to the variables makes φ true; or in Vertex Cover, we are given a graph and an integer $k \in \mathbb{N}$, and we have to decide whether the graph contains a *vertex cover* of size at most k: the set of at most k vertices incident to every edge of the graph. In both cases, the solution (the assignment of Boolean values to the variables or the set of at most k vertices of the graph) has size polynomial in the size of the input and its correctness can be readily verified. In a breakthrough series of works by Cook, Levin, and Karp [Coo71, Lev73, Kar72], some problems – including Satisfiability and Vertex Cover – have been found to be the *most difficult* efficiently verifiable decision problems in the following sense: A correct efficient algorithm for just one of these difficult problems can be adapted to efficiently solve *all* efficiently verifiable problems. Such problems have since been named NP-*complete*. After half of a century of sustained efforts, it is still not clear whether NP-complete problems can be solved efficiently; however, the majority of the community believes that these problems do not admit efficient algorithms [Gas19].

When faced with a task of designing an algorithm – not necessarily foolproof or running in polynomial time – for an NP-complete problem, there are several approaches an algorithm designer can take. Apart from a plethora of heuristic techniques, a popular route is to design *approximation algorithms* that do not solve the task at hand exactly, but rather efficiently compute a solution that is provably close to the optimum [Joh74, WS11]. Another avenue is to *restrict* the space of possible inputs to the problem. For instance, in directed graphs, the problem of existence of an even-length simple path – without repeating vertices or arcs – connecting a pair of prescribed vertices is known to be NP-complete [LP84]. On the other hand, the very same problem restricted to the class of *planar* directed graphs is polynomial-time solvable [Ned99].

Yet another direction is to introduce a "difficulty parameter" to the problem at hand, hoping to find an algorithm that works efficiently for instances with the small values of the parameter. This is precisely the case for Vertex Cover: Given an arbitrary n-vertex graph G and a parameter $k \in \mathbb{N}$, we can determine if G contains a vertex cover of size k in time $2^{\mathcal{O}(k)} \cdot \mathcal{O}(n)$ [BG91, CKX10]. The idea of *two-dimensionality* of time complexity of algorithms – with polynomial dependency on the size of the instance at the cost of possibly superpolynomial dependency on the parameter – is foundational for the field of *parameterized complexity*, first formalized by Downey and Fellows [DF99]. Following the modern treatment of parameterized complexity [FG06, DF13, CFK+15], we say that a problem is *fixed-parameter tractable* (fpt) if we can solve every instance of size n with parameter k in time $f(k) \cdot n^c$ for some constant

$c > 0$ and some computable function f. We also conveniently use the shorthand notation $\mathcal{O}_k(n^c)$ for this time complexity.

Over the decades, it has been understood that various algorithm design techniques listed above can be combined so as to produce efficient algorithms for some difficult computational problems in varied settings. A particularly fitting example of this phenomenon is the technique of shifting introduced by Baker [Bak94], who showed that, given a vertex-weighted n-vertex planar graph G and an accuracy parameter $\varepsilon > 0$, we can produce a $(1-\varepsilon)$-approximation of the maximum weight of an independent set in G in time $2^{\mathcal{O}(1/\varepsilon)} \cdot n$. To this end, we efficiently cover G with $k := \mathcal{O}(1/\varepsilon)$ subgraphs $G_1, \ldots, G_k$, so that some $(1-\varepsilon)$-approximate maximum weighted independent set in G is precisely the maximum weighted independent set in some G_i. Then we argue that each subgraph G_i possesses a strong structural property of *k-outerplanarity*, allowing us to find the sought independent set in each G_i in time $2^{\mathcal{O}(k)} \cdot n$. In the end, the $(1-\varepsilon)$-approximate independent set is output in time $\mathcal{O}_\varepsilon(n)$.

The technique of Baker shows the viability of the following metaapproach: Many difficult graph computational problems can be solved efficiently – that is, in time $\mathcal{O}_k(n^c)$ for some $c > 0$ – when given an n-vertex graph with the bounded value k of some graph parameter. This approach has been applied to numerous important graph parameters: treewidth [Arn85, AP89, ALS91], rankwidth [CMR00, BTV10, GH10, Lam20], treedepth [FY17, PW18, NPSW23], Hadwiger number [Gro03, LPPS22], vertex cover number [FLM+08, KT16, FLMT18] and twin-width [BKTW20, BGK+21b], to name a few. In this dissertation, we will consider various parameterized problems related to three of these parameters: treewidth, rankwidth, and twin-width; we will overview them in a moment, in Section 1.1.

Recently, a new trend appeared in the field of parameterized complexity: the design of *parameterized data structures* [AMV20]. Our goal is to construct a data structure that maintains a dynamically changing graph and can be queried for the satisfaction of some property of a dynamic graph. The objective of such a data structure is to answer such queries more efficiently than by simply restarting the computation from scratch each time. So for instance, the existence of a path of length k in a static n-vertex, m-edge graph can be tested in time $\mathcal{O}_k(n+m)$ [AYZ95]; but then, if an n-vertex graph is updated dynamically by edge insertions and removals, the existence of such a path can be maintained by a suitable data structure in time $\mathcal{O}_k(1)$ per update [DKT14, CCD+21]. The main focus of this thesis is to combine this trend with the notion of graph parameters above: We present efficient data structures that can be queried for various properties of graphs of bounded treewidth, rankwidth, and twin-width; and that can solve various computational problems for fully dynamic graphs of this kind. We also show some new combinatorial results related to these classes that either have already been employed in the design of our efficient data structures, or that we hope to apply in the setting of such data structures in the future.

1.1 An introduction to graph width parameters

We will now take some time to introduce the graph width parameters examined in this thesis: *treewidth*, *rankwidth*, and *twin-width*. This section should be taken as a high-level overview of these parameters. The formal treatment of these parameters, as well as the preliminary observations and proofs regarding graphs of bounded tree-, rank-, and twin-width, can be found in Sections 2.3 to 2.5.

1.1.1 Treewidth

If any structural graph parameter deserves recognition for its importance in the research on structural graph theory, it is definitely treewidth: a measure that assigns to each graph an integer representing, very informally, how similar the graph is to a tree. More formally, given a nonempty graph G, we define the treewidth of G as the minimum integer $k \geqslant 0$ for which there exists a tree T and a function $\mathsf{bag} : V(T) \to 2^{V(G)}$ such that:

- each vertex of G belongs to some bag: for all $v \in V(G)$, there is $x \in V(T)$ with $v \in \mathsf{bag}(x)$;
- each edge of G belongs to some bag: for all $uv \in E(G)$, there is $x \in V(T)$ with $\{u, v\} \subseteq \mathsf{bag}(x)$;
- each vertex of G belongs to the bags forming a connected subtree of T: for every $v \in V(G)$, the set of nodes $\mathsf{bag}^{-1}(v)$ of T is connected;
- all bags are small, i.e., $|\mathsf{bag}(x)| \leqslant k+1$ for each $x \in V(T)$.

A pair (T, bag) with the properties as above is called a *tree decomposition* of G (of width at most k).

The notion of treewidth and the notions equivalent to treewidth were discovered independently under various names, for example by Bertelè and Brioschi [BB73], Halin [Hal76], and Parsons [Par78]. However, the modern treatment of treewidth should arguably be attributed to Robertson and Seymour [RS84], who rediscovered treewidth again and recognized it as a cornerstone of their Graph Minors project.

Over the years, treewidth has found a multitude of algorithmic applications:

- Various graph problems that are NP-complete in the general setting (e.g., Hamiltonian Cycle, Independent Set, Feedback Vertex Set etc.) become tractable when restricted to graphs of bounded treewidth [Arn85, AP89, ALS91, MT92, Bod93b, TP93]. Usually, these problems are solved by means of bottom-up dynamic programming on a tree decomposition of the graph: Root the decomposition T of G at an arbitrary vertex and devise a dynamic programming scheme, where the state of the dynamic programming at a node x corresponds to partial solutions for the subgraph of G whose tree decomposition is the subtree of T rooted at x. Moreover, the dynamic programming state at x can be efficiently determined given the contents of $\mathsf{bag}(x)$ and the states computed for the children of x.

  This framework yields algorithms that process n-vertex graphs of treewidth k in time $f(k) \cdot \mathcal{O}(n)$. In many cases, the function f is singly exponential in k – usually of the form $2^{\mathcal{O}(k)}$ or $k^{\mathcal{O}(k)}$ – and considerable effort has been made to optimize the dependency of k in the time complexity of these algorithms [ABF+02, vRBR09, BBL13, FLPS16, FLPS17, FLP+18, CNP+22].

- More generally, Courcelle [Cou90] showed that as long as a graph property can be encoded by a sentence φ expressed in the *monadic second-order logic with quantification over edge subsets* (CMSO_2)[2], the property can be tested in graphs of treewidth at most k in time $f(k, \varphi) \cdot \mathcal{O}(n)$ for some (nonelementary) computable function f: This follows from an observation that the validation of the property encoded by φ can be efficiently performed using the dynamic programming framework on tree decompositions described above. This result was later generalized to optimization and counting problems [ALS91]. Under reasonable assumptions, such algorithms cannot exist for general graphs of large treewidth [Kre12]; so in a sense, treewidth is the complexity-theoretic boundary of effective testing of graph properties expressible in CMSO_2.

- *Shifting* is an algorithm design technique that is a blueprint of various fixed-parameter tractable algorithms and efficient approximation schemes in planar graphs (and more generally, classes of graphs excluding a fixed graph as a *minor*). The method essentially provides an efficient reduction in which a computational problem can be solved for a planar graph G by selecting a number of subgraphs of G of small treewidth, solving the problem for these subgraphs using standard dynamic programming techniques, and combining the results to produce an optimum or almost-optimum solution to the problem on the original graph G. A precursor of this technique is the aforementioned work of Baker [Bak94]. The approach was later generalized to arbitrary minor-free classes of graphs and beyond [Gro03, Dvo18].

- Another approach to the design of efficient fixed-parameter tractable algorithms in structurally-restricted classes of graphs, such as planar graphs, is *bidimensionality*. Here, the algorithm implements the following win/win approach: The input graph either has small treewidth – in which case the problem at hand can be solved efficiently, once again using dynamic programming – or it exhibits a complicated structure (usually of the form of a *grid minor*) – in which case the answer to the problem is trivally positive or trivially negative. This method allows us to construct fixed-parameter tractable algorithms for problems like determining the existence of vertex cover of size at most k in a planar graph of size n in $2^{\mathcal{O}(\sqrt{k})} \cdot n$ time [ABF+02, DFHT05], even though for general graphs, the solution to this problem in time $2^{o(k)} \cdot n^{\mathcal{O}(1)}$ is unlikely [CJ03]. More remarkably, a dominating set of size at most k in a planar graph can be found in time $2^{\mathcal{O}(\sqrt{k})} \cdot n$ [ABF+02], even though the under reasonable complexity assumptions, there is no such algorithm for general graphs with running time $\mathcal{O}_k(n^{\mathcal{O}(1)})$ [DF95].

- The method above can be refined, producing a framework called the *irrelevant vertex technique*. In this case, in graphs of large treewidth, one can find an obstruction for treewidth (usually of the form of *grid minors*, *clique minors*, or *flat walls*), which is subsequently used to choose a vertex that can be safely removed from the graph without changing the answer to the problem at hand. These vertex removals are performed until treewidth drops below some prescribed threshold dependent on the parameters of

[2]The CMSO_2 logic for graphs permits logic formulas involving variables representing vertices, edges, sets of vertices, and sets of edges, together with symbols $\neg$, $\vee$, $\wedge$, $\in$, $\forall$, $\exists$, true, false with their usual semantics. Formulas can also use the predicate $E(x, y)$ testing whether vertices x and y are connected by an edge, the predicate $\mathsf{inc}(x, z)$ testing whether vertex x is incident to an edge z, and predicates of the form $\mathsf{mod}_{a,m}(X)$ checking whether the set X of vertices or edges has size k (mod m). Quantification can be performed over the vertices, edges, sets of vertices, and sets of edges of the graph. A formal definition of the logic formalisms appearing in this thesis is given in Section 2.6.

the instance, at which point the remaining part of the instance can be resolved by the usual dynamic programming scheme. This technique has been successfully applied in the past to show efficient parameterized algorithms for problems such as Vertex Disjoint Paths [RS95, KKR12, KPS24], Planar Vertex Deletion [MS12] and Minor Testing [RS95, KKR12, KPS24].

Computing tree decompositions. A vast majority of the results above crucially relies on our ability to efficiently compute tree decompositions of graphs of optimum or near-optimum width. Computing treewidth exactly is known to be NP-hard [ACP87], and the Small Set Expansion Conjecture precludes the existence of constant-factor approximation algorithms for treewidth in polynomial time [WAPL14]. However, there exists a host of efficient parameterized algorithms that, given a graph G and integer k, in time $\mathcal{O}_k(n^{\mathcal{O}(1)})$ either output a tree decomposition of G of width k or close to k, or correctly determine that the treewidth of G exceeds k. The following list presents a subjective selection of the most noteworthy algorithms of this kind. Henceforth, we let n denote the number of vertices of the input graph G.

- As part of their Graph Minors project, Robertson and Seymour [RS86a] devised a $2^{\mathcal{O}(k)} \cdot n^2$ time method of computing a 4-approximation of treewidth, which given a graph of treewidth at most w, outputs a tree decomposition of width at most $4w+3$. Due to its simplicity, the algorithm is featured in books on parameterized algorithms [FG06, CFK+15] and taught in courses on parameterized algorithms.

- The first linear-time algorithm for treewidth was given by Bodlaender [Bod96]: In time $2^{\mathcal{O}(k^3)} \cdot n$, it outputs a tree decomposition of optimum width of a graph of treewidth k. In this work, an earlier result of Bodlaender and Kloks [BK96] is invoked: Given a tree decomposition (T, bag) of width ℓ of a graph G and an integer $k \leqslant \ell$, we can determine if treewidth of G is at most k in time $2^{\mathcal{O}(k\ell^2)} \cdot n$ using dynamic programming on (T, bag); in the positive case, the decomposition of G of width k can be recovered within the same time bounds.

- The work of Bodlaender, Drange, Dregi, Fomin, Lokshtanov, and Pilipczuk [BDD+16] is the first algorithm to produce a 5-approximation of treewidth in time $2^{\mathcal{O}(k)} \cdot n$. In order to appreciate the strength of the result, consider *any* linear-time algorithm parameterized by treewidth that accepts a graph together with its width-k tree decomposition and solves some parameterized problem in time $2^{\mathcal{O}(k)} \cdot n$; this includes dynamic programming algorithms for problems such as Maximum Independent Set, Minimum Dominating Set, or Hamiltonian Cycle. Chaining such an algorithm with the result of Bodlaender et al., we find that the algorithm can run in time $2^{\mathcal{O}(k)} \cdot n$ on any n-vertex graph of treewidth k even if its tree decomposition is *not* present on input.

- Recently, Korhonen [Kor21] introduced a technique that enabled him to devise a 2-approximation algorithm of treewidth, also in time $2^{\mathcal{O}(k)} \cdot n$. This framework has since been enhanced by Korhonen and Lokshtanov [KL23] to determine tree decompositions of optimum width in time $2^{\mathcal{O}(k^2)} \cdot n^4$, improving upon the dependency on k from the work of Bodlaender.

Note that it remains an open question whether there exist algorithms producing optimum-width tree decompositions in $2^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$ time [DF99]. The author of the thesis is not aware of any lower bounds against parameterized exact algorithms or parameterized constant-factor approximations for treewidth in time $2^{o(k)} \cdot n^{\mathcal{O}(1)}$, however the existence of such an approximation algorithm should be considered massively surprising. More information on algorithms computing tree decompositions can be found in the work of Korhonen and Lokshtanov [KL23] and in a survey article by Pilipczuk [Pil20].

1.1.2 Rankwidth

The concept of treewidth is inherently unable to analyze *dense* graphs. For instance, an n-vertex graph of treewidth k has at most $\mathcal{O}(kn)$ edges, and whenever a graph contains a clique K_t as a minor, its treewidth is necessarily at least $t-1$. Of course, as mentioned before, the toolchain of treewidth is useful also in the analysis of some classes of graphs of unbounded treewidth – such as the class of planar graphs or various classes of graphs excluding a fixed graph as a minor – but these graphs are still intrinsically sparse. However, there are many well-structured classes of dense graphs. Take for example *cographs* ([Sei74, Jun78]) or *distance-hereditary graphs* ([How77]) for which various algorithmic problems can be solved more efficiently than in the general setting [CLB81, DM88, MN93, HHHK02].

For these reasons, the definition of treewidth has been adapted to the dense setting, producing graph parameters such as *cliquewidth* (defined by Courcelle, Engelfriet, and Rozenberg [CER93]) and *rankwidth* (defined by Oum and Seymour [OS06]), the latter of which we formalize below.

Let G be an undirected graph. A rank decomposition of G is a pair $\mathcal{T} = (T, \lambda)$, where T is a cubic tree (where all nonleaf vertices have degree exactly 3) with at least two leaves, and $\lambda : V(G) \to L(T)$ is a bijection between the set of vertices of G and the set of leaves of T. For every edge $xy \in E(T)$, let M_{xy} be the binary matrix formed by recording the adjacencies between the vertices of G mapped to leaves of T closer to x than y and the vertices mapped to the leaves of T closer to y than x. The width of a rank decomposition of G is the maximum rank of any matrix M_{xy}, where the ranks are determined in the binary field GF(2). Finally, the rankwidth of G is the minimum possible width of a rank decomposition of G.

We delay the formal definition of cliquewidth to Section 2.4. For the purposes of this introduction, it is enough to know that the cliquewidth of a graph is the minimum integer k for which the graph can be represented as a *k-expression*, and that rankwidth and cliquewidth are functionally related – if a graph has cliquewidth at most k, then its rankwidth is also at most k; and conversely, if a graph has rankwidth at most k, then its cliquewidth is at most $2^{k+1} - 1$.

Rankwidth is a stronger graph parameter than treewidth: Whenever a graph has treewidth at most k, it also has rankwidth at most $k + 1$ [Oum08b]. On the other hand, there exist classes of graphs of bounded rankwidth and unbounded treewidth, such as cliques and cographs (rankwidth at most 1) and distance-hereditary graphs (which are exactly the graphs of rankwidth at most 1 [Oum05]). Crucially, a case could be made that rankwidth is *precisely* a "dense" variant of treewidth: The Courcelle's theorem for treewidth naturally lifts to the setting of graphs of bounded cliquewidth [CMR00]. More precisely, whenever a graph property can be encoded by a formula φ expressible in the *monadic second-order logic* (CMSO_1) *without* quantification over edge subsets, then the property can be tested in graphs of cliquewidth at most k in time $\mathcal{O}_{k,\varphi}(n)$. As in the case of treewidth, this result has also been extended to optimization [CMR00] and counting and enumeration problems [CMR01]. So problems like Maximum Independent Set, Minimum Dominating Set, Feedback Vertex Number, c-Coloring for fixed $c \in \mathbb{N}$ can be solved in linear time for graphs of bounded rankwidth or cliquewidth, assuming access to a suitable decomposition of the graph.

Computing rankwidth. The first algorithm to approximate rankwidth was given by Oum and Seymour [OS06], who provided a 3-approximation algorithm for rankwidth in time $2^{\mathcal{O}(k)}n^9 \log n$. Notably, their work actually introduces rankwidth in order to approximate cliquewidth – so, in fact, the main result of their work is an fpt factor-$2^{\mathcal{O}(k)}$ approximation of cliquewidth. However, their result generalizes to the computation of *branchwidth* of submodular symmetric functions, so it can be used to approximate, for example, the branchwidth of graphs or matroids.

After the introduction of rankwidth by Oum and Seymour, a program to optimize the approximation factor and the running time of their algorithm has commenced. We refer to Table 1.1 for the overview of the algorithms for computing rankwidth; here, we only mention the most important results from the point of view of the thesis. The first cubic-time 3-approximation algorithm for rankwidth was devised by Oum [Oum08a]. This was generalized by Jeong, Kim, and Oum [JKO21], who exactly computed the branchwidth of the so-called *subspace arrangements* over finite fields – the notion subsuming rankwidth of graphs. However, the cubic barrier has not been overcome for a long time, even though this goal was explicitly posed by Oum [Oum17]. A breakthrough result of Fomin and Korhonen resulted in an $\mathcal{O}_k(n^2)$ time algorithm computing rankwidth of graphs exactly [FK22], which is asymptotically optimal for dense graphs. The current state-of-the-art result determines rank decompositions of optimum width in time $\mathcal{O}_k(n^{1+o(1)}) + \mathcal{O}(m)$ ([KS24], also Chapter 4 of this thesis).

On the negative side, the NP-completeness of cliquewidth was shown by Fellows, Rosamond, Rotics, and Szeider [FRRS09], and the NP-completeness of rankwidth was proved by Oum [Oum08a].

1.1.3 Twin-width

The notion of twin-width has a fairly curious origin, stemming from the Permutation Pattern problem: Given a permutation σ (a *pattern*) of $\{1, \ldots, k\}$ and a permutation π of $\{1, \ldots, n\}$, determine whether σ is a subpattern of π.[3] The existence of a tractable parameterized algorithm for this problem was confirmed by Guillemot and Marx [GM14], who presented an $2^{\mathcal{O}(k^2 \log k)} \cdot n$-time algorithm implementing

[3] We say that σ is a subpattern of π if there exists a strictly increasing function $f : \{1, \ldots, k\} \to \{1, \ldots, n\}$ such that, for every $i, j \in \{1, \ldots, k\}$, $\sigma(i) < \sigma(j)$ implies $\pi(f(i)) < \pi(f(j))$.

Reference	APX	TIME	Remarks
[OS06]	$3k+1$	$\mathcal{O}(8^k n^9 \log n)$	Works for connectivity functions
[OS07]	exact	$\mathcal{O}(n^{8k+12} \log n)$	Works for connectivity functions
[Oum08a]	$3k+1$	$\mathcal{O}(8^k n^4)$	
[Oum08a]	$3k-1$	$\mathcal{O}_k(n^3)$	
[CO07]	exact	$\mathcal{O}_k(n^3)$	Does not provide a decomposition
[HO08]	exact	$\mathcal{O}_k(n^3)$	
[JKO21]	exact	$\mathcal{O}_k(n^3)$	Works for spaces over finite fields
[FK22]	exact	$\mathcal{O}_k(n^2)$	
[KS24]	exact	$\mathcal{O}_k(n \cdot 2^{\sqrt{\log n} \log\log n}) + \mathcal{O}(m)$	Exposition in Chapter 4

Table 1.1: Overview of algorithms for computing rankwidth. Here n is the number of vertices, m is the number of edges, and k is the rankwidth of the input graph. Unless otherwise specified, each of the algorithms outputs in $\mathcal{O}(\text{TIME})$ a decomposition of width given in the APX column. Wherever the dependency on k in the time complexity is not stated explicitly, it is at least double-exponential in k. The table is taken from similar tables in [FK22] and [KS24].

the following win/win strategy: Either π is "very complicated" and contains all permutations of size k as subpatterns, or π is "simple" and admits a specific decomposition of width bounded by a function of k. Such a decomposition can be computed efficiently given π, and then Permutation Pattern can be solved in linear time given the decomposition of π. The authors asked whether such a decomposition can be adapted to the graph-theoretic setting. Bonnet, Kim, Thomassé and Watrigant answered this question positively [BKTW20], providing the following definition of *twin-width* of graphs.

Let G be a graph with $n \geqslant 1$ vertices. For two disjoint subsets $P, Q \subseteq V(G)$ we say that the pair P, Q is *pure* if either there are no edges between P and Q, or every vertex of P is connected with an edge with every vertex of Q. Otherwise, the pair P, Q is impure. We then define a *contraction sequence* of G as a sequence of partitions $\mathcal{P}_n, \mathcal{P}_{n-1}, \ldots, \mathcal{P}_1$ of $V(G)$, where $\mathcal{P}_n$ contains all vertices of the graph as singletons, $\mathcal{P}_1$ contains a single set $V(G)$, and $\mathcal{P}_i$ is constructed from $\mathcal{P}_{i+1}$ by merging two sets of $\mathcal{P}_i$. The *width* of a contraction sequence is the minimum integer d such that, for every partition $\mathcal{P}_i$ ($i \in \{1, \ldots, n\}$) and every part $P \in \mathcal{P}_i$, there are at most d other parts $Q \in \mathcal{P}_i$ such that the pair P, Q is impure. Then the twin-width of G is the minimum width of any contraction sequence of G.

Twin-width vastly generalizes the notions of treewidth and rankwidth: Whenever a class of graphs has bounded treewidth or bounded rankwidth, it also has bounded twin-width [BKTW20]. However, various other classes of well-structured graphs have bounded twin-width, including planar graphs [JP22, HJ23] and, more generally, classes of graphs excluding a fixed graph as a minor [BKTW20, BKW22], partially ordered sets (posets) of bounded width [BKTW20, BH21] and k-dimensional grids for fixed $k \in \mathbb{N}$ [BKTW20].

On the other hand, twin-width is computationally robust: Assuming access to the contraction sequence of a graph of twin-width d, we can, for instance, verify the satisfaction of any graph property expressible as a sentence φ of *first-order logic* (FO)[4] in time $\mathcal{O}_{d,\varphi}(n)$ [BKTW20]; this unifies analogous results for the special cases of minor-free classes of graphs [FG01], posets of bounded width [GHL+15] and the aforementioned graphs of bounded rankwidth. Notably, a decomposition of a graph of twin-width d allows us to test many such graph properties – for example, k-Independent Set, k-Clique or k-Dominating Set – more efficiently, in time $2^{\mathcal{O}_d(k)} \cdot n$ [BGK+21b]. Twin-width has been recently used as a crucial ingredient of fixed-parameter tractable algorithms for several seemingly unrelated problems, such as k-Independent Set for visibility graphs of simple polygons [BCK+22] and Directed Multicut with three terminal pairs parameterized by the size of the cutset [HJL+23].

Computing twin-width. Unfortunately, when it comes to determining or approximating twin-width of graphs, it seems that significant progress is yet to be made. On the positive side:

- For many specific classes of well-structured graphs that have bounded twin-width (e.g., classes of graphs of bounded treewidth or rankwidth, minor-free classes of graphs, posets of bounded width, etc.), contraction sequences of small width can be constructed efficiently "by hand", by exploiting the structural properties of these classes [BKTW20].

[4] In the first-order logic, quantification is only allowed over the single vertices of the graph.

- Graphs of twin-width 0 are precisely cographs [BKTW20], and graphs of twin-width 1 have a very simple structure [BKR+21], so determining whether a graph has twin-width 0, 1, or more than 1 can be done in polynomial time.

- For graphs with *feedback edge number* (the minimum size of a subset of edges in a graph that hits all the cycles in the graph) bounded by k, there exists an algorithm running in time $\mathcal{O}_k(n^{\mathcal{O}(1)})$ that outputs a contraction sequence of the graph of width exceeding the twin-width of the graph by at most 1 [BGR24].

- Another positive result concerns *ordered graphs* [BGdM+21]. In an ordered graph, the vertices of G are linearly ordered with a linear order $<$. Let us define that for two disjoint subsets $P, Q \subseteq V(G)$ of vertices of an ordered graph, the pair P, Q is *order-pure* if P, Q is pure and, additionally, all vertices of P are before Q with respect to $<$ or vice versa. Then the remaining part of the definition of twin-width of ordered graphs is analogous to the unordered case; in particular, the width of a contraction sequence of G is the minimum integer d such that, for every partition $\mathcal{P}$ of the contraction sequence and every part $P \in \mathcal{P}$, there are at most d other parts $Q \in \mathcal{P}$ such that the pair P, Q is not *order-pure*.

  Then for any ordered graph G and integer $d \in \mathbb{N}$, in time $\mathcal{O}_d(n^{\mathcal{O}(1)})$ we can either compute a contraction sequence of G of width at most $2^{\mathcal{O}(d^4)}$, or correctly determine that G has twin-width more than d.

However, Berge, Bonnet, and Déprés showed a negative result: It is NP-complete to decide whether twin-width of a graph is at most 4 [BBD22]. So we cannot hope for an efficient algorithm computing twin-width of a graph precisely, even in the parameterized setting. Still, this result does not preclude the existence of twin-width approximation algorithms akin to [BGR24] or [BGdM+21]. It could also be possible that some graph parameter functionally related to twin-width *can* be computed exactly by an efficient parameterized algorithm. This would be somewhat similar to the case of cliquewidth and rankwidth: The existence of an algorithm deciding in time $\mathcal{O}_k(n^{\mathcal{O}(1)})$ whether a given n-vertex graph has cliquewidth at most k is still an open problem [FRRS09], and in fact it is not even clear whether graphs of cliquewidth at most 4 can be recognized in polynomial time. However, such algorithms already exist for rankwidth.

1.2 Research objectives

A vast majority of the algorithms presented above exhibit a major disadvantage – they only work efficiently in the *static setting*, where the algorithm is only required to process one fixed input instance within its runtime. So a treewidth computation or approximation algorithm will input a single fixed graph and return an (almost-)optimum width tree decomposition of the graph; and the algorithm of Baker [Bak94] will process a single fixed vertex-weighted planar graph and approximate the maximum weight of an independent set in the graph. However, in many real-life scenarios, we must be prepared to handle *dynamically* changing inputs efficiently: An edge may be introduced to the graph or removed from the graph at any point of time, or a weight of a vertex might change. After such an update we would wish to update the answer to the computational problem at hand more efficiently than by recomputing the answer from scratch.

Historically, the design of efficient graph algorithms in the dynamic setting has long been the main focus of multiple bodies of works. Arguably, the most notorious dynamic graph problem is that of *dynamic connectivity*, where one is given an undirected, initially edgeless, n-vertex graph G that is updated by edge insertions and removals. The goal is to design a data structure that maintains G dynamically and answers queries of the form *"are two given vertices u, v in the same connected component of G?"*. The problem, together with its variants, has attracted considerable attention throughout the history [Tar75, SE81, Fre85, EIT+92, HK99, HdLT01, KKM13, HHK+23]. Currently the most efficient solution to the problem [HHK+23] presents a randomized data structure that handles a single edge update in *amortized* time $\mathcal{O}(\log n(\log\log n)^2)$ and resolves any query in *worst-case* time $\mathcal{O}(\log n/\log\log\log n)$. Here, *amortized* means that a data structure guarantees that q first updates to the graph are processed in total time $\mathcal{O}(q \log n(\log\log n)^2)$ – but some updates might take significantly more time than others. However, there also exist data structures with polylogarithmic worst-case update and query times [KKM13, GKKT15]. Also, efficient data structures have been designed for the problems such as *dynamic planarity* [EGIS96, HR20b, HR20a], *dynamic biconnectivity* [HdLT01, PSS19, HRT18, HvR23], *dynamic minimum spanning*

trees [Fre85, HdLT01, HRW15, Wul17, NS17, NSW17] or *dynamic all-pairs shortest paths* [Kin99, Tho04, ACK17, GW20, Mao23].

A very natural research question is then whether dynamic data structures can be used for the computation of various graph parameters:

Objective 1. *Design efficient data structures that compute or approximate graph parameters in the dynamic graph model.*

Alman, Mnich, and Vassilevska Williams [AMV20] proposed a positive result for the *feedback vertex number*: the minimum size of a set of vertices that hits every cycle in the graph. They showed a data structure that is initialized with an initially edgeless n-vertex graph G and a parameter k, updates it under edge insertions and removals in amortized time $\mathcal{O}_k(\log n)$, and after each query it returns whether the feedback vertex number of G is at most k. (In fact, their algorithm in the positive case also returns the *feedback vertex set* – the sought set of at most k vertices of G.) This result was later accompanied by a work of Majewski, Pilipczuk, and the author of this thesis [MPS23]: The data structure of Alman et al. can be additionally initialized with a property of graphs encoded as a sentence φ in the CMSO_2 logic, and the satisfaction of φ in G can be tracked by the data structure in amortized time $\mathcal{O}_{k,\varphi}(\log n)$ per edge update. This data structure is then used to maintain the solution to the dynamic Cycle Packing problem (*does G contain k vertex-disjoint cycles?*) in amortized $\mathcal{O}_k(\log n)$ time per graph update.

A similar result of this kind is presented by Dvořák, Kupec, and Tůma [DKT14] and subsequently improved by Chen et al. [CCD+21], who maintain in a dynamic graph G whether the *treedepth* of G is at most k in worst-case update time $2^{\mathcal{O}(k^2)}$; and in a positive case, a suitable decomposition of G of width k (an *elimination forest*) is maintained. This result is then used to track the existence of a k-vertex path in a dynamic graph G in worst-case update time $2^{\mathcal{O}(k^2)}$, and the existence of a cycle of length at least k in worst-case update time $\mathcal{O}_k(\log n)$.

The results of [AMV20, CCD+21] above highlight an interesting phenomenon: Efficient algorithms and data structures for graph parameters sometimes incidentally yield efficient dynamic data structures for graph problems that appear unrelated to the basic task of maintaining graph decompositions dynamically. So a reasonable avenue of research is to examine what other kinds of dynamic graph problems are amenable to this strategy:

Objective 2. *Determine how efficient graph decomposition algorithms may help us to design efficient data structures for dynamic graph problems.*

The reader is encouraged to check other works [CSTV93, DT13, DKT14, IO14, GMP+22, OPR+23, MPZ24] that design parameterized data structures for various dynamic problems, not necessarily restricted to graph inputs.

However, interesting data structure design problems do not necessitate that the considered graph should be dynamic. In fact, a perfectly viable theme of such a design problem could be to preprocess a *static* graph (or more generally, a static input instance) so that various properties of the instance could be tested efficiently at a later point of time. In this thesis, we will specifically look into the case of *compact* data structures – roughly speaking, data structures that store an input instance using an amount of memory that is close to the information-theoretical optimum. More formally, let $S(n)$ be the information-theoretical minimum number of bits required to store a valid input instance (n-vertex graph selected from some well-structured class of graphs, $n \times n$ matrix chosen from a class of matrices, etc.). We then say that a data structure storing the instance is compact if it has asymptotically optimal bitsize, i.e., it takes $\mathcal{O}(S(n))$ bits. The challenge here is to design compact data structures that can be constructed efficiently and that still allow querying for various properties of the instance (such as whether two vertices of the graph are connected by an edge).

An example result of this kind is by Kamali [Kam18], who presented a compact data structure for graphs of bounded cliquewidth. Namely, he proved that an n-vertex of cliquewidth k can be implicitly stored in a data structure of bitsize $\mathcal{O}_k(n)$, implementing vertex adjacency queries in constant time and neighbor enumeration queries in constant time per neighbor. He also showed that $\Omega(kn)$ bits are required to store an n-vertex of cliquewidth k, thus demonstrating that his data structure is compact.

This obviously raises the question whether compact data structures could be designed for even more general classes of graphs, such as those of bounded twin-width:

Objective 3. *Design data structures for well-structured graphs (or, more generally, well-structured input instances) with low memory requirements, so that the properties of these instances can be queried efficiently.*

Finally, for some very expressive graph parameters such as twin-width, the combinatorial foundations underlying these parameters might not be strong enough yet, preventing us from designing effective algorithms and data structures parameterized by these parameters. Therefore it makes perfect sense to analyze and prove various strong combinatorial properties of these parameters, hoping that they will allow us to actually design such algorithms and data structures:

Objective 4. *Find and prove strong combinatorial properties of twin-width that may enable us to design efficient parameterized data structures.*

In the following section we will present results related to the research objectives listed above.

1.3 Our results

In this dissertation we give an exposition of our research on the objectives stated in Section 1.2. This section is dedicated to an overview of the results encompassed in this thesis.

1.3.1 Dynamic treewidth and rankwidth

We present dynamic data structures that maintain tree and rank decompositions of graphs of approximately optimum width in amortized subpolynomial time per update.

Dynamic treewidth. For treewidth, our result reads as follows:

Theorem 1.3.1 ([KMN+23]). *There is a data structure that for an integer $k \in \mathbb{N}$, fixed upon initialization, and a dynamic graph G, updated by edge insertions and deletions, maintains a tree decomposition of G of width at most $6k+5$ whenever G has treewidth at most k. More precisely, at every point in time the data structure either contains a tree decomposition of G of width at most $6k+5$, or a marker* "Treewidth too large"*, in which case it is guaranteed that the treewidth of G is larger than k. The data structure can be initialized on k and an edgeless n-vertex graph G in time $g(k) \cdot n$, and then every update takes amortized time $2^{f(k)\cdot\sqrt{\log n \log\log n}}$, where $g(k) \in 2^{k^{\mathcal{O}(1)}}$ and $f(k) \in k^{\mathcal{O}(1)}$ are computable functions.*

Moreover, upon initialization the data structure can be also provided a CMSO_2 sentence φ, and it can maintain the information whether φ is satisfied in G whenever the marker "Treewidth too large" *is not present. In this case, the initialization time is $g(k,\varphi) \cdot n$ and the amortized update time is $h(k,\varphi) \cdot 2^{f(k)\cdot\sqrt{\log n \log\log n}}$, where g, h, and f are computable functions.*

We can apply a fairly standard trick to rewrite the time complexity bound $2^{f(k)\cdot\sqrt{\log n \log\log n}}$ to the form $\widehat{f}(k) \cdot 2^{\sqrt{\log n}\log\log n}$ for a computable function $\widehat{f}$: If the value of the parameter k is sufficiently small compared to the size of the graph n (say, when $f(k) < \sqrt{\log\log n}$), we simply have $2^{f(k)\cdot\sqrt{\log n \log\log n}} \leqslant 2^{\sqrt{\log n}\log\log n}$. Otherwise, we infer that $\sqrt{\log n \log\log n} \leqslant 2^{f(k)^2} \cdot f(k)$ and so $2^{f(k)\cdot\sqrt{\log n \log\log n}}$ can be bounded by some function $\widehat{f}(k)$.

We remark that the update time of the form $2^{\mathcal{O}(\sqrt{\log n \log\log n})}$ is sandwiched between the polylogarithmic time (i.e., any time complexity of the form $\mathcal{O}(\log^C n)$ for constant $C > 0$) and polynomial time (i.e., any time complexity of the form $\mathcal{O}(n^\delta)$ for constant $\delta > 0$). To the best of our knowledge, no nontrivial – with update time sublinear in n – data structure for treewidth was known prior to this result. On the other hand, there exists an $\Omega(\log n)$ lower bound on the amortized time complexity of a single update: The data structure of Theorem 1.3.1 can model the problem of dynamic connectivity in forests, which admits an $\Omega(\log n)$ amortized lower bound as proved by Pătraşcu and Demaine [PD06].

Let us review the existing results related to the dynamic maintenance of tree decompositions of graphs.[5]

- None of the aforementioned static algorithms for computing tree decompositions is known to be liftable to the dynamic setting.
- There is a wide range of data structures for dynamic maintenance of forests and of various dynamic programming procedures working on them, see e.g. [BF99, ST83, AHdLT05, Nie18]. Unfortunately, the simple setting of dynamic forests omits the main difficulty of the dynamic treewidth problem: the need of reconstructing the tree decomposition itself upon updates. Consequently, we do not see how any of these approaches could be lifted to graphs of treewidth higher than 1.

[5] The following description is taken from [KMN+23].

- Bodlaender [Bod93a] showed that on graphs of treewidth at most 2, tree decompositions of width at most 11 can be maintained with worst-case update time $\mathcal{O}(\log n)$. This result also comes with a dynamic variant of Courcelle's Theorem: The satisfaction of any CMSO_2-expressible property φ on graphs of treewidth at most 2 can be maintained with worst-case update time $\mathcal{O}_\varphi(\log n)$. The approach of Bodlaender relies on a specific structure theorem for graphs of treewidth at most 2, which unfortunately does not carry over to larger values of the treewidth. In [Bod93a], Bodlaender also observed that for $k > 2$, update time $\mathcal{O}_k(\log n)$ can be achieved in the decremental setting, when only edge deletions are allowed. But this again avoids the main difficulty of the problem, as in this setting no rebuilding of the tree decomposition is necessary.

- Independently of Bodlaender, Cohen et al. [CSTV93] tackled the case $k = 2$ with worst-case update time $\mathcal{O}(\log^2 n)$, and the case $k = 3$ in the incremental setting with worst-case update time $\mathcal{O}(\log n)$. Frederickson [Fre98] studied dynamic maintenance of properties of graphs of treewidth at most k, but the updates considered by him consist of direct manipulations of tree decompositions; this again avoids the main difficulty.

- As discussed before, analogs of Theorem 1.3.1 have already been studied for two weaker graph parameters: treedepth and feedback vertex number. Dvořák et al. [DKT14] and Chen et al. [CCD+21] gave a dynamic variant of Courcelle's Theorem for treedepth: On a dynamic graph of treedepth at most d, the satisfaction of any fixed MSO_2-expressible[6] property φ can be maintained with worst-case update time $\mathcal{O}_{\varphi,d}(1)$. In turn, a dynamic variant of Courcelle's Theorem for feedback vertex number was given by Majewski et al. [MPS23], who builds upon the data structure of Alman et al. [AMV20]: They show that in a dynamic graph of feedback vertex number at most ℓ, the satisfaction of any fixed CMSO_2-expressible property φ can be maintained with amortized update time $\mathcal{O}_{\varphi,\ell}(\log n)$.

- Recently, Goranci et al. [GRST21] gave the first nontrivial result on the general dynamic treewidth problem: Using a dynamic algorithm for *expander hierarchy*, they can maintain a tree decomposition with $n^{o(1)}$-approximate width with amortized update time $n^{o(1)}$, under the assumption that the graph has bounded maximum degree. Note, however, that this result is rather unusable in the context of (dynamic) parameterized algorithms, because dynamic programming procedures on tree decompositions work typically in time exponential in the decomposition's width. Consequently, the result does not imply any dynamic variant of Courcelle's Theorem.

Observe that the data structure of Theorem 1.3.1 is resilient to updates that increase the treewidth of the dynamic graph G above k. In this case, the data structure exposes the marker "Treewidth too large", which is removed by the implementation of the data structure as soon as the treewidth of G drops below k again. Yet the amortized update time of the data structure is still of the form $2^{f(k)\cdot\sqrt{\log n \log\log n}}$, regardless of the treewidth of G at any point of time.

Theorem 1.3.1 can be immediately used to provide efficient data structures for the dynamic versions of several parameterized problems solved by the classical bidimensionality arguments. Consider for instance the following Planar Minor Containment problem:

Corollary 1.3.2 (Statement and proof from [KMN+23])**.** *Let H be a fixed planar graph. There exists a data structure that for an n-vertex graph G, updated by edge insertions and deletions, maintains whether H is a minor of G. The initialization time on an edgeless graph is $\mathcal{O}(n)$, and the amortized update time is $2^{\mathcal{O}(\sqrt{\log n \log\log n})}$.*

Proof. Since H is planar, there exists $\ell \in \mathbb{N}$ such that the $\ell \times \ell$ grid contains H as a minor. Consequently, by the Grid Minor Theorem [RS86b], there exists $k \in \mathbb{N}$ such that every graph of treewidth larger than k contains H as a minor. Further, it is easy to write a CMSO_2 sentence φ_H that holds in a graph G if and only if G contains H as a minor. It now suffices to set up the data structure of Theorem 1.3.1 for the treewidth bound k and sentence φ_H. Note there that if this data structure contains the marker "Treewidth too large", it is necessary the case that G contains H as a minor. □

The same argument can be used to show the existence of a dynamic data structure deciding Planar Dominating Set (given a dynamic planar graph, decide whether it contains a dominating set of cardinality at most k) with amortized update time $2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$.

[6] MSO_2 is a fragment of CMSO_2 where modular counting predicates are not allowed.

We finally remark that the data structure for dynamic rankwidth can be used to solve a certain class of optimization problems expressible in the variant of monadic second-order logic called $\mathsf{LinCMSO}_2$. This variant includes all optimization problems of the form "*given a* CMSO_2 *formula* $\varphi(X)$ *with one vertex set or edge set variable* X*, find the minimum/maximum-size vertex set* $A \subseteq V(G)$ *or edge set* $A \subseteq E(G)$ *for which* $\varphi(A)$ *is satisfied in* G"; however, we refer to Section 2.6 for a formal introduction of $\mathsf{LinCMSO}_2$.

Theorem 1.3.3. *The data structure of Theorem 1.3.1 can be also provided a* $\mathsf{LinCMSO}_2$ *sentence* φ *upon initialization and can maintain the value of* φ *in* G *whenever the marker* "Treewidth too large" *is not present. The initialization time is* $g(k,\varphi)\cdot n$ *and the amortized update time is* $h(k,\varphi)\cdot 2^{f(k)\cdot\sqrt{\log n \log\log n}}$ *for computable functions* g, h, f*.*

Hence by Theorem 1.3.3, optimization problems expressible in $\mathsf{LinCMSO}_2$ – such as Longest Path, Longest Cycle, Maximum Independent Set, Minimum Feedback Vertex Set – can be solved in dynamic graphs of bounded treewidth efficiently: The value of the optimum solution to each of these problems can be maintained by our data structure with subpolynomial amortized update time.

The full exposition of the proof of Theorems 1.3.1 and 1.3.3 is given in Chapter 3.

Dynamic rankwidth. We present also an analogous data structure for rank decompositions. In the following statement, $\mathsf{LinCMSO}_1$ denotes an optimization variant of the CMSO_1 logic analogous to $\mathsf{LinCMSO}_2$; we again delay the formal introduction of this variant to Section 2.6, but for the purposes of this introduction it is enough to know that $\mathsf{LinCMSO}_1$ can encode problems of the form "*find the minimum/maximum-size set* $A \subseteq V(G)$ *for which a given* CMSO_1 *formula is satisfied*".

Theorem 1.3.4 ([KS24])**.** *There is a data structure that is initialized with an integer* k *and an empty* n*-vertex dynamic graph* G*, and maintains a rank decomposition of* G *of width at most* $4k$ *under edge insertions and deletions, under the promise that the rankwidth of* G *never exceeds* k*. The initialization time is* $\mathcal{O}_k(n\log^2 n)$ *and the amortized update time is* $2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$*.*

Further, the data structure can be initialized with a $\mathsf{LinCMSO}_1$ *sentence* φ *and it can maintain the value of* φ *on* G*. In such a case, the initialization time of the data structure is* $\mathcal{O}_{k,\varphi}(n\log^2 n)$ *and the amortized update time is* $2^{\mathcal{O}_{k,\varphi}(\sqrt{\log n \log\log n})}$*.*

We stress that the data structure of Theorem 1.3.4 does *not* inherit the resiliency of its treewidth counterpart, so we have assume that the rankwidth of the dynamic graph is bounded from above by k at all times. This dissonance will be further explained in the overview of the dynamic rankwidth data structure in Section 4.1.1.

Theorem 1.3.4 immediately implies that given a graph G of rankwidth at most k with n vertices and m edges, we can construct a rank decomposition of G of width at most $4k+4$ in time $(n+m)\cdot 2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$: We can prove that the edges of G can be inserted to the data structure in an order such that the rankwidth of the dynamic graph never grows above $k+1$.[7] So after m edge insertions, the data structure will hold a rank decomposition of G of width at most $4(k+1)$. However, both this time complexity and the width of the resulting decomposition can be improved further, as discussed below.

In the description above, a fairly natural issue is that in order to build a decomposition of an m-edge graph, we need to issue m edge updates to the data structure of Theorem 1.3.4. For instance, if the input graph is a clique, we necessarily have to apply as many as $\Omega(n^2)$ edge insertions. However, in this case it seems rather wasteful to insert edges one by one; somehow, we should be able to issue updates of the form "*given a set* $X \subseteq V(G)$*, insert all the edges between any pair of vertices in* X" and process them in time proportional to $|X|$ rather than $|X|^2$. We thus introduce a framework of performing dense updates of the graph that can modify many edges of the graph at once. We use the following definition from [KS24] verbatim: An *edge update sentence* is a tuple $\overline{e} = (\varphi, X, X_1, \ldots, X_p)$, where φ is a CMSO_1 formula with $p+1$ free set variables, and $X_i \subseteq X \subseteq V(G)$. Such edge update sentence re-defines all adjacencies inside the induced subgraph $G[X]$ by setting an edge between $u, v \in X$ if and only if G, together with the interpretations of the $p+1$ free variables as $(\{u,v\}, X_1, \ldots, X_p)$, satisfies φ. Then we define that $|\overline{e}| = |X|$ and that the length of $\overline{e}$ is the length of φ. Given this definition, we generalize Theorem 1.3.4 as follows:

Theorem 1.3.5 ([KS24])**.** *The data structure of Theorem 1.3.4, when furthermore initialized with a given integer* d*, can also support the following operations:*

[7] An example of such an ordering is as follows: An edge uv is added to the data structure before the edge $u'v'$ if and only if the pair $(\max(u,v), \min(u,v))$ is lexicographically smaller than $(\max(u',v'), \min(u',v'))$ with respect to some fixed total ordering of $V(G)$.

- $\mathsf{Update}(\overline{e})$: *Given an edge update sentence $\overline{e}$ of length at most d, either returns that the graph resulting from applying $\overline{e}$ to G would have rankwidth more than k, or applies $\overline{e}$ to update G. Runs in $|\overline{e}| \cdot 2^{\mathcal{O}_{k,d}(\sqrt{\log n \log\log n})}$ amortized time.*
- $\mathsf{LinCMSO}_1(\varphi, X_1, \ldots, X_p)$: *Given a $\mathsf{LinCMSO}_1$ sentence φ of length at most d with p free set variables and p vertex subsets $X_1, \ldots, X_p \subseteq V(G)$, returns the value of φ on $(G, X_1, \ldots, X_p)$. Runs in time $\mathcal{O}_d(1)$ if $X_1, \ldots, X_p = \emptyset$, and in time $\sum_{i=1}^{p} |X_i| \cdot 2^{\mathcal{O}_{k,d}(\sqrt{\log n \log\log n})}$ otherwise.*

The operation $\mathsf{LinCMSO}_1$ here is quite useful for us for a very prosaic reason: In the setting of dynamic graphs with dense edge updates, it is nontrivial to determine whether two vertices of a graph are connected by an edge. As one of the consequences, Theorem 1.3.5 asserts that the existence of edges between specific pairs of vertices in the graph can be queried efficiently even if dense updates are applied to the graph.

The generalization in Theorem 1.3.5 allows us to compute rank decompositions of graphs even more efficiently than sketched above:

Theorem 1.3.6 ([KS24]). *There is an algorithm that, given an n-vertex m-edge graph G and an integer k, in time $\mathcal{O}_k(n \cdot 2^{\sqrt{\log n}\log\log n}) + \mathcal{O}(m)$, either outputs a rank decomposition of G of width at most k or determines that the rankwidth of G is larger than k. The algorithm also outputs a $(2^{k+1}-1)$-expression for cliquewidth of G within the same running time.*

Moreover, every fixed graph problem that can be expressed in $\mathsf{LinCMSO}_1$ can be solved in time $\mathcal{O}_k(n \cdot 2^{\sqrt{\log n}\log\log n}) + \mathcal{O}(m)$ on graphs of rankwidth k.

This result is an improvement upon the previous quadratic-time algorithm of Fomin and Korhonen [FK22]; and it is only a factor of $n^{o(1)}$ away from the parameterized linear time complexity $\mathcal{O}_k(n+m)$. Interestingly, using Theorem 1.3.6, we can determine if the graph has rankwidth at most k in *true* linear time, independent on the parameter k, as long as the input graph is sufficiently dense (more precisely: as long as the average degree of the graph exceeds $f(k) \cdot 2^{\sqrt{\log n}\log\log n}$ for a fixed function f).

The results related to the dynamic rankwidth data structure and its applications are proved in Chapter 4.

1.3.2 Dynamic shifting

We then show that in some settings, the technique of Baker [Bak94] also applies in dynamic graphs. The following results apply to the class of planar graphs, and, more generally, to any *apex-minor-free* class of graphs. A class of graphs is apex-minor-free if all graphs in the class exclude as a minor a fixed *apex graph*: a graph that either is planar, or that becomes planar after the removal of one vertex. So the class of planar graphs is apex-minor-free since they exclude the clique K_5 as a minor; and so is the class of toroidal graphs, since all toroidal graphs exclude the biclique $K_{3,7}$ as a minor and $K_{2,7}$ is planar. (In general, every class of graphs of genus at most g is apex-minor-free: Graphs of genus at most g are $K_{3,\,4g+3}$-minor-free, which follows from [Rin65].)

We present dynamic efficient approximation schemes for arguably the most natural optimization problems amenable to the shifting technique: Maximum-Weight Independent Set and Minimum-Weight Dominating Set. In these problems, we associate a real weight with each vertex of the graph. In the dynamic variant, one can add or remove edges from the graph and update the weights assigned to the vertices of the graph. Moreover, for a fixed apex-minor-free class of graphs $\mathcal{C}$, we say that a data structure is *$\mathcal{C}$-restricted* if it assumes that at each point of time, the maintained graph is a member of $\mathcal{C}$. Then our result reads as follows:

Theorem 1.3.7 ([KNPS24]). *Let $\mathcal{C}$ be a fixed apex-minor-free class of graphs and let $\varepsilon > 0$. Then there exists a $\mathcal{C}$-restricted fully dynamic graph data structure that in addition to maintaining a graph $G \in \mathcal{C}$, supports the following queries:*

- $\mathsf{QueryMWIS}()$: *Outputs a nonnegative real p satisfying $(1-\varepsilon)\mathsf{OPT}_{\mathsf{IS}} \leqslant p \leqslant \mathsf{OPT}_{\mathsf{IS}}$, where $\mathsf{OPT}_{\mathsf{IS}}$ is the maximum weight of an independent set in G; and*
- $\mathsf{QueryMWDS}()$: *Outputs a nonnegative real p satisfying $\mathsf{OPT}_{\mathsf{DS}} \leqslant p \leqslant (1+\varepsilon)\mathsf{OPT}_{\mathsf{DS}}$, where $\mathsf{OPT}_{\mathsf{DS}}$ is the minimum weight of a dominating set in G. This query is supported only under the additional assumption that at all times, the maximum degree of G is bounded by a constant Δ.*

The initialization time on a given n-vertex graph $G \in \mathcal{C}$ is $f(\varepsilon) \cdot n^{1+o(1)}$, and each update takes amortized time $f(\varepsilon) \cdot n^{o(1)}$, where $f(\varepsilon)$ is doubly-exponential in $\mathcal{O}(1/\varepsilon^2)$. Each query takes $\mathcal{O}(1)$ time.

Unfortunately, the data structure for Minimum-Weight Dominating Set requires the additional assumption that the maximum degree of the graph is bounded by a constant; it is not clear whether this restriction can be somehow bypassed.

The (amortized) update time of the data structure is also superlogarithmic and subpolynomial, similarly to the case of dynamic tree- and rankwidth data structures. However, in this case, the update time is *barely* subpolynomial:

$$f(\varepsilon) \cdot n^{\mathcal{O}\left(\frac{\log\log\log n}{\sqrt{\log\log n}}\right)}.$$

Curiously, the data structure of Theorem 1.3.7 is unrelated to the dynamic treewidth data structure of Theorem 1.3.1. It would be interesting to understand whether the dynamic treewidth data structure could be used to improve the time dependency on the size of the graph in Theorem 1.3.7.

The full exposition of this result can be found in Chapter 5.

1.3.3 Compact representation for bounded twin-width

Next, we prove that one can design compact data structures for objects of bounded twin-width. Our particular result considers *binary matrices*: matrices comprised of entries 0 and 1. One of the ways the notion of twin-width can be generalized to such matrices is through the definition of *d-twin-ordered* matrices. Intuitively, for binary matrices, one can consider contraction sequences comprising successively coarser partitions of the set of all rows and all columns of the matrix, where at each step of the contraction process, each set in the partition is either a set of consecutive rows (a *row block*) or a set of consecutive columns (a *column block*). Then at each step of the process, we can merge two consecutive row blocks or two consecutive column blocks. Then the matrix is d-twin-ordered if it admits a contraction sequence where at each step, each row (respectively, column) block "interacts nontrivially" with at most d column (resp., row) blocks. The formal definition of d-twin-ordered matrices is delayed to Section 2.5. For now, we encourage the reader to assume that a d-twin-ordered matrix is roughly equivalent to the adjacency matrix of a graph of twin-width d.

Let us now consider the problem of storing a d-twin-ordered square matrix compactly so that the entries of the matrix can be queried in an efficient manner. By the work of Bonnet et al. [BGdM+21], the number of distinct d-twin-ordered $n \times n$ binary matrices is $2^{\Theta_d(n)}$. Hence any faithful representation of such a matrix must necessarily have bitsize at least $\Omega_d(n)$. The challenge is to construct a data structure that stores such a matrix within *precisely* $\mathcal{O}_d(n)$ bits, allowing at the same time for an efficient access to the entries of the matrix.

Before we state our result, let us quickly present the previous knowledge on the problem. In all the following results, we assume the Word RAM model of computation, which allows us to perform constant-time computations (arithmetic and bitwise operations and comparisons) on integers with bitsize $\mathcal{O}(\log n)$. The following review of literature is borrowed verbatim from our work [PSZ22].

- Storing the matrix explicitly is a representation with bitsize $\mathcal{O}(n^2)$ and query time $\mathcal{O}(1)$.
- In [BGK+21a], Bonnet et al. presented an *adjacency labeling scheme* for graphs of bounded twin-width, which can be readily translated to the matrix setting. This scheme assigns to each row and each column of the matrix a *label* – a bitstring of length $\mathcal{O}_d(\log n)$ – so that the entry in the intersection of a row and a column can be uniquely decoded from the pair of their labels. In [BGK+21a] the time complexity of this decoding is not analyzed, but a straightforward implementation runs in time linear in the length of labels. This gives a representation with bitsize $\mathcal{O}_d(n \log n)$ and query time $\mathcal{O}_d(\log n)$.
- It follows from the results of [BGK+21b] that if matrix M is d-twin-ordered, then the entries 1 in M can be partitioned into $\ell = \mathcal{O}_d(n)$ rectangles, say $R_1, \ldots, R_\ell$ (see Lemma 2.5.2 for a proof). This reduces our question to 2*D orthogonal point location*: designing a data structure that for a given point in $(i, j) \in \{1, \ldots, n\}^2$, may answer whether (i, j) belongs to any of the rectangles $R_1, \ldots, R_\ell$. For this problem, Chan [Cha13] designed a data structure with bitsize $\mathcal{O}(n \log n)$ and query time $\mathcal{O}(\log\log n)$ assuming $\ell = \mathcal{O}(n)$. So we get a representation of M with bitsize $\mathcal{O}_d(n \log n)$ and query time $\mathcal{O}_d(\log\log n)$.
- For 2D orthogonal point location one can also design a simple data structure by persistently recording a sweep of the square $\{1, \ldots, n\}^2$ using a B-ary tree for $B = n^\varepsilon$, for any fixed $\varepsilon > 0$. This gives a representation with bitsize $\mathcal{O}_d(n^{1+\varepsilon})$ and query time $\mathcal{O}(1/\varepsilon)$. See Section 6.4 for details.

We present a data structure with an asymptotically optimum $\mathcal{O}_d(n)$ bitsize and doubly-logarithmic query time:

Theorem 1.3.8 ([PSZ22]). *Let $d \in \mathbb{N}$ be a fixed constant. Then for a given binary $n \times n$ matrix M that is d-twin-ordered one can construct a data structure that occupies $\mathcal{O}_d(n)$ bits and can be queried for entries of M in worst-case time $\mathcal{O}(\log \log n)$ per query. The construction time is $\mathcal{O}_d(n \log n \log \log n)$ in the Word RAM model, assuming M is given by specifying $\ell = \mathcal{O}_d(n)$ rectangles $R_1, \ldots, R_\ell$ that form a partition of symbols 1 in M.*

It remains a nagging open problem whether the query time in Theorem 1.3.8 could be brought down to a constant, perhaps depending on the parameter d, while maintaining the linear bitsize at the same time.

The full exposition of the data structure of Theorem 1.3.8 is given in Chapter 6.

1.3.4 Quasi-polynomial χ-boundedness of classes of bounded twin-width

Last but not least, we present combinatorial evidence that graphs of bounded twin-width have strong structural properties. In this thesis, we focus on the property of χ-*boundedness*. We say that a class of graphs $\mathcal{C}$ is χ-bounded if:

- it is hereditary (i.e., every induced subgraph of a graph in $\mathcal{C}$ is also a member of $\mathcal{C}$), and
- there exists a function $f : \mathbb{N} \to \mathbb{N}$ such that the vertices of every graph $G \in \mathcal{C}$ can be properly colored using at most $f(\omega(G))$ colors, where $\omega(G)$ is the size of the maximum clique in G. In other words, $\chi(G) \leqslant f(\omega(G))$, where $\chi(G)$ is the chromatic number of G.

The notion of χ-boundedness is a relative of *perfect graphs*: A graph G is perfect if for every induced subgraph H of G (including G itself), the vertices of H can be properly colored using at most $\omega(H)$ colors. Perfect graphs enjoy strong structural properties, allowing us to design polynomial-time algorithms for finding the maximum clique or maximum independent set [GLS84]. Example classes of perfect graphs are chordal graphs [Dir61] and distance-hereditary graphs [How77].

χ-boundedness is a slight weakening of the notion of perfect graphs. Many well-structured classes of graphs are χ-bounded, for instance circle graphs with the χ-bounding function $f(\omega) = \mathcal{O}(\omega \log \omega)$ [Dav22]; intersection graphs of axis-parallel rectangles in the Euclidean plane, also with the χ-bounding function $f(\omega) = \mathcal{O}(\omega \log \omega)$ [CW21]; the class of graphs excluding a t-vertex path P_t as an induced subgraph for every fixed $t \in \mathbb{N}$, with the χ-bounding function $f(\omega) = \mathcal{O}(t^{\omega-1})$ [Gyá87]; and the class of graphs of cliquewidth at most k for every fixed $k \in \mathbb{N}$, with the χ-bounding function $f(\omega) = \omega^{\mathcal{O}_k(1)}$ [BP20]. A result that a given class of graphs is χ-bounded does not usually have any algorithmic implications *per se*; however, a proof of χ-boundedness tends to involve a novel structural decomposition for a class of graphs, which may be exploited algorithmically in subsequent works. We refer to the survey by Scott and Seymour [SS20] for more information on the research and open problems in this area of combinatorial graph theory.

In one of the foundational works on twin-width [BGK+21b], Bonnet, Geniet, Kim, Thomassé and Watrigant proved that, for every fixed $d \in \mathbb{N}$, the class of graphs of twin-width at most d is χ-bounded, with the χ-bounding function *exponential* in the clique number ω:

Theorem 1.3.9 ([BGK+21b]). *For every $d \in \mathbb{N}$, every graph G of twin-width at most d and clique number ω satisfies*

$$\chi(G) \leqslant (d+2)^{\omega-1}.$$

The techniques used in the proof of Theorem 1.3.9 were used in the same work to develop several efficient algorithms for graphs of bounded twin-width: For instance, given a graph G with n vertices together with a contraction sequence of width d, and a source s, the single-source shortest paths tree rooted at s can be determined in time $\mathcal{O}_d(n \log n)$, even though G may have as many as $\Omega(n^2)$ edges.

In [BGK+21b], the authors asked a natural question whether the dependency on ω in the χ-bounding function in Theorem 1.3.9 could be improved to polynomial. In this thesis, we present a result that has made huge progress towards the goal of Bonnet et al. by showing that the χ-bounding function can be made *quasi-polynomial* in ω:

Theorem 1.3.10 ([PS23]). *For every $d \in \mathbb{N}$ there exists a constant $\gamma_d \in \mathbb{N}$ such that for every graph G of twin-width at most d and clique number ω, we have*

$$\chi(G) \leqslant 2^{\gamma_d \cdot \log^{4d+3} \omega}.$$

On a very high level, the idea behind the proof is as follows. By [BKTW20], matrices of small twin-width (such as d-twin-ordered matrices introduced in the previous section) exclude a pattern called a *k-mixed minor* for $k = \mathcal{O}(d)$: a partition of the set of rows into k row blocks and the set of columns into k column blocks such that every submatrix formed by the intersection of any row block with any column block is *mixed*. Here, a submatrix is mixed if it contains at least two distinct rows and at least two distinct columns. In our work, we introduce and work with a slightly more relaxed variant of the pattern that we call a *k-almost mixed minor*. Such obstructions are defined analogously to k-mixed minors, only that we do not require that the submatrix at the intersection of the ith row block with the ith column block be mixed. We then prove that the adjacency matrices of graphs of twin-width at most d also exclude k-almost mixed minors for $k = \mathcal{O}(d)$.

Now an important observation is that we can measure the complexity of a graph in terms of the size of the largest almost mixed minor in the adjacency matrix of the graph. Our contribution is the following graph decomposition scheme: Given a graph G whose adjacency matrix excludes a k-almost mixed minors, we can – roughly speaking and sweeping a lot of technical details under the rug – vertex-partition it into several subgraphs $G_1, G_2, \ldots, G_p$ such that:

- each subgraph G_i is *slightly* simpler than the original graph G: Its adjacency matrix still excludes a k-almost mixed minor, but the clique number of G_i is multiplicatively smaller than that of G; and
- the p-vertex graph H representing how the subgraphs $G_1, G_2, \ldots, G_p$ interact with each other is *significantly* simpler than the original graph G: Its adjacency matrix now excludes a $(k-1)$-almost mixed minor.

This enables us to perform a structural induction on the size of the maximum order of an almost mixed minor in the adjacency matrix of the graph, which eventually allows us to conclude that graphs with clique number ω and without k-almost mixed minors in their adjacency matrix have their chromatic number bounded by $2^{\mathcal{O}_k(\log^{k-1} n)}$.

The proof of Theorem 1.3.10 is presented in Chapter 7.

Follow-up work. After the publication of [PS23], Bourneuf and Thomassé refined our decomposition scheme, thus creating an object called the *delayed decomposition* of a graph [BT23]. Briefly, having obtained a vertex-partition of G into subgraphs $G_1, G_2, \ldots, G_p$ similar to the description above, they further decompose each subgraph G_i recursively until one-vertex subgraphs of G are produced. This way, they create a rooted tree T with the leaves of T identified with the vertices of G. Define for every node $x \in V(T)$ the subgraph G_x of G induced by the set of vertices of G identified with the leaves of T that are descendants of x. Then, for every nonleaf node $x \in V(T)$ with p children $x_1, \ldots, x_p$, (the adjacency matrix of) the p-vertex graph representing the interactions between the subgraphs $G_{x_1}, \ldots, G_{x_p}$ has a strictly smaller almost mixed minor than the maximum almost mixed minor of (the adjacency matrix of) G. This strategy eventually allows them to derive a *polynomial* bound on the χ-bounding function for bounded twin-width:

Theorem 1.3.11 ([BT23]). *For every $d \in \mathbb{N}$ there exists a constant $\alpha_d \in \mathbb{N}$ such that for every graph G of twin-width at most d and clique number ω, we have*

$$\chi(G) \leqslant \omega^{\alpha_d}.$$

Note that the degree of the polynomial must necessarily depend on the twin-width of the graph. This phenomenon appears already in the more restricted case of bounded cliquewidth: Bonamy and Pilipczuk [BP20] observed that the construction of Chudnovsky, Penev, Scott, and Trotignon [CPST13] produces for every $k \in \mathbb{N}$ a class of graphs of cliquewidth at most k for which any χ-bounding polynomial has degree at least $\Omega(\log k)$. A very similar construction has been independently proposed by Nešetřil, Ossona de Mendez, Rabinovich, and Siebertz [NdMRS21]. Since every class of graphs of bounded cliquewidth has bounded twin-width, this lower bound transfers to the setting of graphs of bounded twin-width.

As an ending remark, we mention that the delayed decomposition of Bourneuf and Thomassé inspired by our work has already found an algorithmic application: Pattern-free permutations can be effectively represented as a composition of *separable permutations* – permutations excluding two permutations $(2,4,1,3)$ and $(3,1,4,2)$ as subpatterns (or equivalently, permutations that can be created from the one-element permutation by performing *direct sums* and *skew sums*).

Theorem 1.3.12 ([BBGT24]). *Let σ be a fixed pattern that is a permutation of $\{1, \ldots, k\}$. Then there exists an integer $c_k \in \mathbb{N}$ such that every permutation π of $\{1, \ldots, n\}$ that excludes σ as a subpattern is a product of at most c_k separable permutations. Moreover, this product can be found in time $\mathcal{O}_k(n)$.*

We believe that the result of Theorem 1.3.12 could enable us to improve the query time of the compact data structure of Theorem 1.3.8 from $\mathcal{O}(\log \log n)$ to $\mathcal{O}_d(1)$, while maintaining the asymptotically optimal bitsize $\mathcal{O}_d(n)$. This is subject of a work in progress.

1.4 Organization of the thesis

The content of this thesis is composed of the following publications:

- Tuukka Korhonen, Konrad Majewski, Wojciech Nadara, Michał Pilipczuk, Marek Sokołowski, *Dynamic Treewidth* (Chapter 3, [KMN+23]). Paper presented at the 64th IEEE Symposium on Foundations of Computer Science (FOCS 2023).
- Tuukka Korhonen, Marek Sokołowski, *Almost-Linear Time Parameterized Algorithm For Rankwidth Via Dynamic Rankwidth* (Chapter 4, [KS24]). Paper accepted for presentation at the 56th Annual ACM Symposium on Theory of Computing (STOC 2024).
- Tuukka Korhonen, Wojciech Nadara, Michał Pilipczuk, Marek Sokołowski, *Fully Dynamic Approximation Schemes on Planar and Apex-Minor-Free Graphs* (Chapter 5, [KNPS24]). Paper presented at the 35th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2024).
- Michał Pilipczuk, Marek Sokołowski, Anna Zych-Pawlewicz, *Compact Representation For Matrices of Bounded Twin-Width* (Chapter 6, [PSZ22]). Paper presented at the 39th International Symposium on Theoretical Aspects of Comuter Science (STACS 2022).
- Michał Pilipczuk, Marek Sokołowski, *Graphs of Bounded Twin-Width Are Quasi-Polynomially χ-Bounded* (Chapter 7, [PS23]). Paper published in the Journal of Combinatorial Theory, Series B.

The publications above are imported mostly verbatim into this thesis, with minimal changes so as to preserve the consistency of the notation. Moreover, both this Introduction and Preliminaries (Chapter 2) include some passages taken from these works.

The presentation of the results is split into two parts. In Part I, we give an exposition of the results of the three first papers, related to the parameters treewidth and rankwidth. The part is opened by a full presentation of the data structure for treewidth (Chapter 3), followed by its rankwidth analog (Chapter 4). Afterwards, we present the dynamic shifting data structure for dynamic variants of Maximum Weighted Independent Set and Minimum Weighted Dominating Set (Chapter 5).

Then in Part II, we show the remaining two results on graphs and matrices of bounded twin-width. We begin with the compact data structure for d-twin-ordered matrices (Chapter 6) and conclude with the proof of quasi-polynomial χ-boundedness of graphs of bounded twin-width (Chapter 7).

Chapter 2

Preliminaries

2.1 Notation

2.1.1 Integers, sets and functions

The set $\mathbb{N}$ of natural numbers is comprised of all nonnegative integers, including 0. Whenever we write $[n]$ for $n \in \mathbb{Z}$, we mean the set $\{1, \ldots, n\}$ (which is empty for $n \leqslant 0$); and whenever we write $[\ell, r]$ for $\ell, r \in \mathbb{Z}$, we mean the set $\{\ell, \ell+1, \ldots, r\}$ (which is empty for $\ell > r$). All logarithms in this thesis are base-2.

For a set X and a natural number $k \in \mathbb{N}$, we define $\binom{X}{k}$ as the family of all subsets of X of size k. Also, for a set X and a family of sets $\{A_x \mid x \in X\}$, we define $\prod_{x \in X} A_x$ as the Cartesian product of all the sets A_x. Formally, $f \in \prod_{x \in X}$ is a function mapping each $x \in X$ to an element $f(x) \in A_x$. A partition of a set X is a collection of nonempty pairwise disjoint subsets of X whose union is X. For a partition $\mathcal{C}$ of X, we use notation $\bigcup \mathcal{C} = X$. If X, Y are two sets, then $X \triangle Y = (X \cup Y) \setminus (X \cap Y)$ is the symmetric difference of X and Y.

Suppose $f : X \to Y$ is a function and $X' \subseteq X$. The *restriction* of f to X' is the function $f|_{X'} : X' \to Y$ such that $f|_{X'}(x) = f(x)$ for each $x \in X'$.

2.1.2 Graphs

Basic notation. We use standard graph terminology. A graph G contains a set of *vertices* $V(G)$ and a set of *edges* $E(G)$. In this thesis, unless specified otherwise, all graphs are finite, undirected and simple. In particular, all graphs exclude self-loops and multiple edges between the same pair of vertices. Hence $E(G)$ can be considered to be a subset of $\binom{V(G)}{2}$. However, sometimes it will be more convenient to treat each edge uv of G as two ordered edges $\vec{uv}$ and $\vec{vu}$. For this reason we define the set of oriented edges of G as $\vec{E}(G) \coloneqq \bigcup_{uv \in E(G)} \{\vec{uv}, \vec{vu}\}$. In many algorithmic applications we shall assume the set $V(G)$ to be totally ordered; one can essentially assume that $V(G)$ is a finite subset of $\mathbb{N}$. Two vertices $u, v \in V(G)$ are *adjacent* if $uv \in E(G)$. A vertex $u \in V(G)$ is *incident* to an edge $e \in E(G)$ if u is one of the endpoints of e. For $X \subseteq V(G)$, we write $\overline{X}$ to mean $V(G) \setminus X$.

A *path* of length $\ell \geqslant 1$ is an ordered sequence $v_1 v_2 \ldots v_{\ell-1} v_\ell$ of ℓ distinct vertices so that any two consecutive vertices are adjacent. We denote by $\mathcal{P}_\ell(G)$ the set of paths of length ℓ in G.

Neighborhoods. When u is a vertex of G, we denote the open neighborhood of u by $N_G(u) \coloneqq \{v \in V(G) \mid uv \in E(G)\}$ and its closed neighborhood by $N_G[u] \coloneqq N_G(u) \cup \{u\}$. If $X \subseteq V(G)$, we define the closed and open neighborhoods of X as, respectively, $N_G[X] \coloneqq \bigcup_{u \in X} N_G[u]$ and $N_G(X) \coloneqq N_G[X] \setminus X$. In all the notation, we may drop G from the subscript if the graph is known from the context. Two vertices u, v are *twins with respect to* a set $X \subseteq V(G)$ if $N(u) \cap X = N(v) \cap X$. We will simply say that u, v are *twins* if u, v are twins with respect to $V(G) \setminus \{u, v\}$.

For two disjoint nonempty sets of vertices $X, Y \subseteq V(G)$, we say that the pair X, Y is *complete* if every vertex of X is adjacent to every vertex of Y, and *anti-complete* if no edge connects a vertex of X with a vertex of Y. (Note that these notions ignore the existence of edges between the pairs of vertices in X and between the pairs of vertices in Y.) Then a pair X, Y is *pure* if it is either complete or anti-complete. Otherwise, a pair X, Y is *impure*. We also say that X is *semi-pure* towards Y if every vertex of X is either complete or anti-complete towards Y. (If $X \cup Y = V(G)$, this is equivalent to saying that Y is a *module* in G.)

Subgraphs and minors. We use the following notation for denoting *induced subgraphs* of G: When $X \subseteq V(G)$, we write $G[X]$ to mean the subgraph of G with $V(G[X]) = X$ and $E(G[X]) = E(G) \cap \binom{X}{2}$. Sometimes it will be more convenient to write $G - X$ to mean $G[\overline{X}]$. We generalize the notation to bi- and multipartite induced subgraphs: If $X, Y \subseteq V(G)$ are disjoint, we define $G[X, Y]$ to be the subgraph of G with $V(G[X, Y]) = X \cup Y$, where $E(G[X, Y])$ contains exactly the edges of G with one endpoint in X and the other in Y. Similarly, if $\mathcal{C}$ is a collection of nonempty pairwise disjoint subsets of $V(G)$, then $G[\mathcal{C}]$ is a subgraph of G with $V(G[\mathcal{C}]) = \bigcup \mathcal{C}$ and with $uv \in E(G[\mathcal{C}])$ if and only if $uv \in E(G)$ and u, v belong to different parts of $\mathcal{C}$.

For a graph G, to *contract* an edge $uv \in E(G)$ is to replace the vertices u, v in G with a new vertex w such that $N(w) = N(u) \cup N(v) \setminus \{u, v\}$. Then a graph H is a *minor* of G if H can be constructed from G through a sequence of vertex or edge removals and edge contractions.

Connectivity, distances, and separations. Next, $\mathrm{cc}(G)$ denotes the partition of $V(G)$ into the set of connected components of G. Also, for $u, v \in V(G)$, we denote by $\mathsf{dist}_G(u, v)$ the distance between u and v in G, i.e., the minimum number of edges on a simple path between u and v. If u and v are in separate connected components of G, we put $\mathsf{dist}_G(u, v) = +\infty$. The maximum distance between any pair of vertices of G is called the *diameter* of G, and the value $\min_{u \in V(G)} \max_{v \in V(G)} \mathsf{dist}_G(u, v)$ is called the *radius* of G.

For a set $A \subseteq V(G)$, we define the *torso* of A in G, denoted $\mathsf{torso}_G(A)$, as the graph H on the vertex set A in which $uv \in E(H)$ if and only if $u, v \in A$ and there exists a path connecting u and v that is internally disjoint with A. Equivalently, $u, v \in A$ and we have that $uv \in E(G)$ or there exists a connected component $C \in \mathrm{cc}(G - A)$ for which $\{u, v\} \subseteq N_G(C)$.

Given two sets $A, B \subseteq V(G)$, we say that a set $S \subseteq V(G)$ is an (A, B)-*separator* if every path connecting a vertex of A and a vertex of B intersects S. If sets A, S, B form a partition of $V(G)$, then (A, S, B) is a *separation* of G. The *order* of this separation is $|S|$. Next, if every (A, B)-separator S has size at least $|A|$, then we say that A is *linked into* B. Equivalently (by Menger's theorem), there exist $|A|$ vertex-disjoint paths connecting A and B. Note that if $B \subseteq B'$ and A is linked into B, then A is also linked into B'.

Graph parameters. We say that a graph parameter is a function mapping a graph to a natural number with the property that any pair of isomorphic graphs are mapped to the same number. Some examples of graph parameters are treewidth, rankwidth and twin-width, already mentioned in the Introduction and formally defined in Sections 2.3 to 2.5. We additionally define the following graph parameters: $\omega(G)$ is the *clique number* of G (i.e., the size of the maximum clique in G) and $\chi(G)$ is the *chromatic number* of G (i.e., the minimum number of colors required in a proper coloring of G, that is, in a coloring of vertices of G in which no two vertices connected by an edge receive the same color).

Classes of graphs. A *class of graphs* is any family of graphs. We say that a class $\mathcal{C}$ of graphs is *monotone* if for every graph $G \in \mathcal{C}$, all subgraphs of G also belong to $\mathcal{C}$. Similarly, we say that $\mathcal{C}$ is *hereditary* if for every $G \in \mathcal{C}$, all induced subgraphs of G also are elements of $\mathcal{C}$; and *minor-closed* if for every $G \in \mathcal{C}$, all minors of G are in $\mathcal{C}$. Next, for a graph H, the class $\mathcal{C}$ is *H-minor-free* if no graphs in $\mathcal{C}$ contain H as a minor. More generally, let $\mathcal{P}$ be a property of graphs; we then say that the class $\mathcal{C}$ *has property $\mathcal{P}$* or *is $\mathcal{P}$* if all graphs in $\mathcal{C}$ satisfy $\mathcal{P}$.

Let p be a graph parameter. Then a class $\mathcal{C}$ of graphs *has bounded p* if there exists an integer $B \in \mathbb{N}$ such that $p(G) \leqslant B$ for all $G \in \mathcal{C}$. (So we will sometimes say that a class of graphs has bounded treewidth, rankwidth, cliquewidth, twin-width etc.) However, we will say that $\mathcal{C}$ is *χ-bounded* if there exists a *function* $f : \mathbb{N} \to \mathbb{N}$ such that $\chi(G) \leqslant f(\omega(G))$ for each $G \in \mathcal{C}$. The function f in this definition is called the *χ-bounding function.*[8]

2.1.3 Trees and forests

Basic definitions. A *forest* is an acyclic undirected graph. If a forest is additionally connected, then the graph is actually a *tree*. We reserve the name *nodes* for the vertices of the tree.

In many cases trees and forests will be *rooted*. In case of trees, this means that exactly one node is designated as the root of the tree, and all the remaining vertices have exactly one parent – the unique neighbor that is closer to the root of the tree. Then the rooted forests are simply collections of rooted

[8] It appears that many researchers prefer to use the name *χ-binding function* for the function f. The author of this thesis is prepared to fight fiercely over this terminology.

trees. Whenever a node p is a parent of a node x (which we denote $p = \mathsf{parent}(x)$), we say that x is a child of p. Two distinct nodes x, y are *siblings* if they have the same parent or they are both roots of trees in a rooted forest. A rooted tree (forest) is *binary* if every node has at most two children. We do not order the children of a node, so in particular we do not distinguish the "left" or the "right" child of a node in a binary tree. An unrooted tree is *subcubic* if every node is of degree at most 3, and *cubic* if all nodes are of degree 1 or 3. To *contract* a degree-2 node in a tree is to contract any edge incident to it. Unrooted cubic trees and rooted binary trees correspond to each other: A cubic tree can be created from a binary tree by contracting its root, and a binary tree is constructed from a cubic tree by subdividing an edge of the cubic tree and designating the newly created vertex as the root of the tree.

Leaves and prefixes. A node is a *leaf* of a rooted tree (forest) if it has no children; and in case of unrooted trees (forests), a leaf is a vertex of degree 1. For a tree (forest) T, define $L(T)$ as the set of leaves of T; and define $\vec{L}(T)$ as the set of *leaf edges* of T, i.e., the oriented edges pointing away from the leaves of T. There exists a natural bijection between $L(T)$ and $\vec{L}(T)$.

For an oriented edge $\vec{xy} \in \vec{E}(T)$, we denote by $L(T)[\vec{xy}] \subseteq L(T)$ the subset of the leaves of T that are closer to x than y. The set $\vec{L}(T)[\vec{xy}] \subseteq \vec{L}(T)$ is defined analogously, i.e., $\vec{L}(T)[\vec{xy}] = \{\vec{lp} \in \vec{E}(T) \mid l \in L(T)[\vec{xy}]\}$. When T is rooted and $t \in V(T)$, we use $L(T)[t]$ to denote the set of leaves that are descendants of t. The set $\vec{L}(T)[t]$ is defined analogously.

For a rooted tree T, we declare that a nonempty set $A \subseteq V(T)$ is a *prefix* of T if A is empty or $T[A]$ is a connected subgraph of T containing the root of T. Then in a rooted forest F, we say that $A \subseteq V(F)$ is a prefix of F if $A \cap T$ is a prefix of T for every connected component T of F. If A is disjoint from the set of leaves of T, we additionally say that A is *leafless*. If a node x does not belong to A, but its parent does, we say that x is an *appendix* of A. Equivalently, $\mathsf{App}_T(A) \coloneqq N_T(A)$ is the set of appendices of T_{conn}. For convenience, define also $\vec{\mathsf{App}}_T(A) \coloneqq \{\vec{xy} \in \vec{E}(T) \mid x \notin A,\, y \in A\} \subseteq \vec{E}(T)$ as the set of *appendix edges* of A. Observe that if a tree T is binary, then $|\mathsf{App}_T(A)| \leqslant |A| + 1$ holds for every prefix A of T. All these definitions lift naturally to the setting where T is an unrooted tree and A induces a connected subtree of T. Also, these definitions can be generalized to handle appendices of prefixes of rooted forests.

Ancestors and descendants. In a rooted forest F, we say that a node x is an *ancestor* of y if x lies on the unique simple path between y and the root of the tree containing y. Equivalently we will say that y is a *descendant* of x or write $x \preccurlyeq_F y$. Note that every node is an ancestor and a descendant of itself. Let $\mathsf{parent}_T(u)$ denote the parent of x in T (or $\perp$ if x is a root in T), and we denote the ancestors and the descendants of x as $\mathsf{anc}_F[x] \coloneqq \{y \mid y \preccurlyeq_F x\}$ and $\mathsf{desc}_F[x] \coloneqq \{y \mid x \preccurlyeq_F y\}$, respectively. We can generalize $\mathsf{anc}_F[x]$ to $\mathsf{anc}_F[A]$ for $A \subseteq V(F)$ by defining $\mathsf{anc}_F[A] = \bigcup_{x \in A} \mathsf{anc}_F[x]$; this set will be called the *ancestor closure* of A in F. It follows that a set $A \subseteq V(F)$ is a prefix of F if and only if $\mathsf{anc}_F[A] = A$.

For $x, y \in V(T)$ in a rooted tree T we define the *lowest common ancestor* of x and y in T, denoted $\mathrm{lca}(x, y)$, as the deepest node z of T that is an ancestor of both x and y in T. A set $S \subseteq V(T)$ is *lca-closed* if $\mathrm{lca}(x, y) \in S$ whenever $x, y \in S$. For a set $S \subseteq V(T)$ we define the *lca-closure* of S as the (unique) inclusion-wise minimal lca-closed set containing S in its entirety. It is a well-known fact that if X is the lca-closure of a nonempty set S, then $|X| \leqslant 2|S| - 1$.

We define the *depth* $\mathsf{depth}_F(x)$ of a node x in a rooted forest F is the distance between x and the root of the tree containing x, and its *height* $\mathsf{height}_F(x)$ as the maximum number of nodes on a path from x to a descendant of x. This implies that the depth of a root of a tree is 0, while the height of a leaf is 1. The height of a tree is the height of its root, and the height of a forest is the maximum height of a tree in the forest.

We can also define an ancestor-descendant relationship for the oriented edges of a rooted tree (forest) as follows. An oriented edge $\vec{xy}$ of a rooted tree T is directed towards the root if y is the parent of x, and away from the root otherwise. We say that an oriented edge $\vec{xy}$ of a (rooted or unrooted) tree T is a *predecessor* of an oriented edge $\vec{zw}$ if either $\vec{xy} = \vec{zw}$ or there is a path in T between y and z that avoids x and w. The set of predecessors of $\vec{zw}$ is denoted by $\mathsf{pred}_T(\vec{zw})$. If $\vec{xy}$ is a predecessor of $\vec{zw}$ then we say $\vec{zw}$ is a *successor* of $\vec{xy}$. If $\vec{xy}, \vec{yz} \in \vec{E}(T)$ with $x \neq z$, then $\vec{xy}$ is called a *child* of $\vec{yz}$.

2.1.4 Matrices

Unless explicitly stated otherwise, we will work with matrices with entries 0 and 1; we shall call such matrices *binary*. The *rank* of a binary matrix is defined in a standard way over the binary field GF(2). A *submatrix* of a matrix is created by restricting the set of rows and columns of a matrix arbitrarily.

In particular, if R is a subset of the set of rows of a matrix M and C is the subset of the set of columns of M, we say that $M[R, C]$ is the submatrix of M at the intersection of R and C. A subset of rows or columns of a matrix is *convex* if it is contiguous in the order of rows (columns) in the matrix. We also say that a *row block* is a nonempty convex set of rows and a *column block* is an analogous set of columns. A submatrix of a matrix created by restricting it to a row block and a column block is called a *zone* of a matrix. Then a *division* of matrix M is a pair $\mathcal{D} = (\mathcal{R}, \mathcal{C})$, where $\mathcal{R}$ is a partition of rows into row blocks and $\mathcal{C}$ is a partition of columns into column blocks. Note that such a division partitions M into $|\mathcal{R}| \cdot |\mathcal{C}|$ zones, each induced by a pair of blocks $(R, C) \in \mathcal{R} \times \mathcal{C}$. A *$t$-division* is a division where $|\mathcal{R}| = |\mathcal{C}| = t$. By $\mathcal{D}[i, j]$ we mean the zone of M at the intersection of the ith row block of $\mathcal{R}$ and the jth column block of $\mathcal{C}$. Similarly, by $\mathcal{D}[[i_1, i_2], [j_1, j_2]]$ we mean the zone of M comprised of the union of zones $\mathcal{D}[i, j]$ for $i \in [i_1, i_2]$, $j \in [j_1, j_2]$. If M is a symmetric matrix, we say that $\mathcal{D}$ is *symmetric* if $\mathcal{R} = \mathcal{C}$.

If G is a graph with vertices ordered according to a total order $<$, then the adjacency matrix of G is a binary matrix M with rows and columns both labeled $V(G)$ and both ordered according to $<$, where $M[u, v] = 1$ if and only if $uv \in E(G)$. For $X \subseteq V(G)$, we define the adjacency matrix M of the bipartite induced subgraph $G[X, \overline{X}]$ by indexing the rows of M by X and the columns of M by $\overline{X}$.

2.2 Parameterized and dynamic problems

In this section we give formal definitions of parameterized problems for dynamically changing inputs. Then we introduce a framework for describing the robustness and efficiency of the solutions to these problems. The formal treatment of the notion of parameterized complexity is standard (see e.g. [DF13, CFK+15]), while the description of dynamic parameterized problems and algorithms is inspired by the work of Alman, Mnich, and Vassilevska Williams [AMV20].

Model of computation. Across the entire thesis, unless specified otherwise, we will assume the *Word RAM* model of computation [FW90]. In the Word RAM model, we consider a machine whose working memory is an array A of w-bit machine words. The index at which any given machine word is stored is called an *address*. Basic manipulation of machine words can be performed in constant time; this includes arithmetic operations (such as addition, subtraction, multiplication and division), bitwise operations (such as bitwise *or*, *and*, *xor*, *not*, and bit shifts) and comparisons. Moreover, the model permits *random accesses* – instructions of the form $A[A[i]] \leftarrow A[j]$ or $A[i] \leftarrow A[A[j]]$ that perform loads or stores at the memory addresses prescribed by another machine word. Assuming that an algorithm in the Word RAM model can access the machine words at indexes from 0 to $s - 1$, inclusive, we define that the *space* of this algorithm is s machine words or sw bits. Note that the existence of random accesses requires that $s \leqslant 2^w$ so as to ensure that all machine words can be accessed.

Choosing n to be the size of input data to an algorithm in the Word RAM model, we require that $w = \Omega(\log n)$ so that the entire input can be read using random accesses. Throughout this thesis, we will actually make a traditional assumption that $w = \Theta(\log n)$.

In practice, we will not be particularly concerned about the precise definition of the model, and instead we will assume that we can perform arbitrary arithmetic and bitwise operations, comparisons and indirect accesses in constant time. It can be verified that all described algorithms indeed adhere to the formal definition of the model described above, however it is customary to omit this verification for the sake of brevity and clarity.

Classical and parameterized complexity. Fix a finite alphabet Σ; by convention, we usually assume $\Sigma = \{0, 1\}$. A word $x \in \Sigma^\star$ is a finite sequence of characters of Σ. A language over Σ is an arbitrary set $L \subseteq \Sigma^\star$ of words over Σ. A problem Π is called a decision problem if it is described by a language L_Π over Σ. Then a solution to the problem Π is any algorithm that, given an input word $x \in \Sigma^\star$, decides whether $x \in L_\Pi$. Next, a problem Π is a function problem when it is defined by a function $f_\Pi : \Sigma^\star \to \Sigma^\star$. In this setting, a solution to Π is any algorithm that, given an input word $x \in \Sigma^\star$, outputs $f_\Pi(x)$. An algorithm solving a problem (be it a decision or a function problem) is polynomial-time if it terminates in time $\mathcal{O}(|x|^c)$ for all possible inputs x for some $c > 0$.

In turn, the framework of *parameterized complexity* crucially relies on, in a sense, "two-dimensionality" of inputs. Formally, a problem Π shall be called a parameterized decision problem if it is described by a set $L_\Pi \subseteq \Sigma^\star \times \Sigma^\star$. Then a parameterized algorithm solves Π when, given as input a pair $(x, k) \in \Sigma^\star \times \Sigma^\star$, correctly decides whether $(x, k) \in L_\Pi$. Here, the value k is called the *parameter* of the input. Usually, the parameter will be a nonnegative integer encoded in unary, an encoding of a logic sentence or formula,

an encoding of a graph, or any tuple of these objects. In many cases, we will assume that k is a nonnegative integer encoded in unary, so we will conveniently assume L_Π to be a subset of $\Sigma^\star \times \mathbb{N}$, and then the size of the input (x,k) will be $|x|+k$. Similarly, a parameterized function problem is prescribed by a function $f_\Pi : \Sigma^\star \times \Sigma^\star \to \Sigma^\star$, and a parameterized algorithm solves Π when, given as input a pair $(x,k) \in \Sigma^\star \times \Sigma^\star$, correctly computes $f_\Pi(x,k)$.

An algorithm is *fixed-parameter tractable* (fpt) if it terminates in time $f(k)\cdot|x|^c$ for some constant $c>0$ and some computable function f. Usually we will write $\mathcal{O}_k(|x|^c)$ as a shorthand for $f(k)\cdot|x|^c$ for some computable f. Whenever k represents a tuple $(k_1,\ldots,k_t)$ of parameters, we will also write $\mathcal{O}_{k_1,\ldots,k_t}(|x|^c)$.

Dynamic parameterized problems. We now introduce a formalism for problems where we have to maintain the satisfaction of properties of dynamically changing inputs and respond to various queries about the current input. We define this formalism in a larger generality than [AMV20] in order to allow nonlocal updates to the input instance that potentially modify a significant part of the instance.

Fix a set $L_\Pi \subseteq \Sigma^\star \times \Sigma^\star$ of *admissible* inputs $(x,k) \in L_\Pi$. (For example, L_Π may encode a set of pairs (G,k), where G is a graph of treewidth k; or a set of pairs (G,H), where G is an H-minor-free graph.) A dynamic parameterized problem Π contains a set L_Π, zero or more *update schemes* $(\mathsf{update}_i, L_i^{\mathsf{update}})$, where $\mathsf{update}_i : \Sigma^\star \times \Sigma^\star \to \Sigma^\star$ and $L_i^{\mathsf{update}} \subseteq \Sigma^\star \times \Sigma^\star$, and one or more *query schemes* $(\mathsf{query}_i, L_i^{\mathsf{query}})$, where $\mathsf{query}_i : \Sigma^\star \times \Sigma^\star \to \Sigma^\star$ and $L_i^{\mathsf{query}} \subseteq \Sigma^\star \times \Sigma^\star$. A dynamic parameterized data structure maintains a pair $(x,k) \in \Sigma^\star \times \Sigma^\star$ under the following types of operations:

- $\mathsf{Initialize}(x,k)$: Initializes the data structure with an admissible input $(x,k) \in L_\Pi$;
- $\mathsf{Update}_i(y)$: Given an update description $y \in \Sigma^\star$ such that $(y,k) \in L_i^{\mathsf{update}}$, replaces x with $x' := \mathsf{update}_i(x,y)$;
- $\mathsf{Query}_i(y)$: Given a query description $y \in \Sigma^\star$ such that $(y,k) \in L_i^{\mathsf{query}}$, returns $\mathsf{query}_i(x,y)$.

One should assume that updates are usually of the form "add or remove a given edge uv from the graph" or "replace the weight of a vertex v with c"; however, we may permit more complex types updates, such as "given a set X of vertices of a dynamic graph G, replace the set of edges of G with $E(G)\triangle\binom{X}{2}$" or "given $X \subseteq V(G)$ and a binary logic formula $\varphi(x,y)$, add edge uv to G if and only if $u,v \in X$, $u \neq v$ and $\varphi(u,v)$". Note that updates cannot alter the value of the parameter k. However, updates and queries may be constrained by the parameter k through the sets L_i^{update}, L_i^{query}, so for example the length of the formula φ above could be upper-bounded by k.

Whenever a dynamic parameterized problem defines some updates, we should define whether updates may cause the input to become inadmissible (so for instance, if L_Π denotes graphs of treewidth at most k, we should define if updates can cause the treewidth of the dynamic graph to increase above k). We introduce three models of resiliency against such updates:

- In case a data structure is L_Π*-restricted*, we place an additional requirement that after each update, the current input is admissible.
- This requirement is not imposed if the data structure *weakly supports* L_Π *membership*. However, in this case, given an operation $\mathsf{Update}_i(y)$, the data structure must recognize that $(\mathsf{update}_i(x,y),k) \notin L_\Pi$ and *reject* this update. Rejected updates are not applied to the input.
- On the other hand, if the data structure *strongly supports* L_Π *membership*, all updates are applied to the input, regardless of whether the resulting input is admissible or not. However, whenever a query is made about an inadmissible input, the data structure is allowed to return $\mathsf{Input\ inadmissible}$ in lieu of the actual answer to the query.

We then define *amortized* and *worst-case* update and query time bounds. Let $T^{\mathsf{init}} : \mathbb{N} \times \Sigma^\star \to \mathbb{N}$ and $T_i^{\mathsf{update}}, T_i^{\mathsf{query}} : \mathbb{N} \times \mathbb{N} \times \Sigma^\star \to \mathbb{N}$ be some functions. Suppose the data structure is initialized with input (x,k) and let n be the size of the initial input. We will say that the initialization of the data structure takes time $T^{\mathsf{init}}(n,k)$, the ith update $\mathsf{Update}_i(y)$ takes amortized time $T_i^{\mathsf{update}}(n,|y|,k)$, and the ith query $\mathsf{Query}_i(y)$ takes amortized time $T_i^{\mathsf{query}}(n,|y|,k)$ whenever the *total* time spent by the data structure to initialize itself with an input (x,k) and process any sequence of operations is bounded from above by the sum of:

- $T^{\mathsf{init}}(n,k)$,

- $T_i^{\mathsf{update}}(n, |y|, k)$ for every processed update of the form $\mathsf{Update}_i(y)$, and
- $T_i^{\mathsf{query}}(n, |y|, k)$ for every processed query of the form $\mathsf{Query}_i(y)$.

We may additionally designate some of the operation time bounds as *worst-case*. If we say that $\mathsf{Update}_i(y)$ takes worst-case time $T_i^{\mathsf{update}}(n, |y|, k)$, we further require that the data structure processes each update $\mathsf{Update}_i(y)$ in time at most $T_i^{\mathsf{update}}(n, |y|, k)$. We analogously define worst-case query time bounds. Whenever we do not specify whether a given operation time bound is amortized or worst-case, we implicitly assume the time bound is worst-case.

We will then say that the data structure is *almost linear fixed-parameter tractable* if the initialization time bound is almost linear in the input size, i.e., $T^{\mathsf{init}}(n, k) \leqslant f(k) \cdot n^{1+o(1)}$ for some computable function f, and all operation time bounds are subpolynomial in n and almost linear in $|y|$, i.e., $T_i^{\mathsf{update}}(n, |y|, k), T_i^{\mathsf{query}}(n, |y|, k) \leqslant f(k) \cdot n^{o(1)} \cdot |y|^{1+o(1)}$.

2.3 Treewidth

Let G be an undirected graph. A *tree decomposition* of G is a pair $\mathcal{T} = (T, \mathsf{bag})$, where T is an arbitrary tree and bag is a function $\mathsf{bag} : V(T) \to 2^{V(G)}$ with the following properties:

- $\bigcup_{x \in V(T)} \mathsf{bag}(x) = V(G)$, i.e., every vertex of G belongs to some $\mathsf{bag}(x)$;
- *edge condition:* for every $uv \in E(G)$, there exists $x \in V(T)$ such that $\{u, v\} \subseteq \mathsf{bag}(x)$;
- *vertex condition:* for every $v \in E(G)$, the set $\{x \in V(T) \mid v \in \mathsf{bag}(x)\}$ is a connected subtree of T.

We will usually picture tree decompositions as trees, where within each node x of the tree we inscribe a "bag", that is, the set $\mathsf{bag}(x)$.

The *width* of a tree decomposition (T, bag) is the maximum size of any bag in the decomposition, minus 1. Then the *treewidth* of a graph is the minimum possible width of any tree decomposition of the graph.

For an edge $xy \in E(T)$, we define the *adhesion* of xy as $\mathsf{adh}(xy) := \mathsf{bag}(x) \cap \mathsf{bag}(y)$. If the decomposition is rooted (i.e., the tree T is rooted), then the adhesion of x, denoted $\mathsf{adh}(x)$, is the adhesion of the edge connecting x with the parent of x in T. The adhesion of the root of T is empty. We then also define the *component* of x as follows:

$$\mathsf{cmp}(x) := \left(\bigcup \{\mathsf{bag}(y) \mid y \text{ is a descendant of } x \text{ in } T\}\right) \setminus \mathsf{adh}(x).$$

We note that $(\mathsf{cmp}(t), \mathsf{adh}(t), V(G) \setminus (\mathsf{cmp}(t) \cup \mathsf{adh}(t)))$ is a separation of G.

We now list standard observations regarding tree decompositions and treewidth. The proofs of these observations are standard.

Observation 2.3.1 ([Die12])**.** *If $C \subseteq V(G)$ is a clique in G and (T, bag) is a tree decomposition of G, then there exists a node $x \in V(T)$ such that $C \subseteq \mathsf{bag}(x)$.*

Observation 2.3.2 ([CFK+15])**.** *If G is a graph of treewidth at most k, then $|E(G)| \leqslant k \cdot |V(G)|$.*

A classic result of Bodlaender and Hagerup implies that any tree decomposition of width k can be turned into a tree decomposition of the same graph of width at most $3k + 2$ and height logarithmic in the number of vertices of the graph.

Theorem 2.3.3 ([BH98, Lemma 2.2])**.** *Given a graph G and its tree decomposition (T, bag) of width k, one can compute in time $\mathcal{O}(k \cdot |V(T)|)$ a binary tree decomposition (T', bag') of G of height $\mathcal{O}(\log |V(T)|)$, width at most $3k + 2$, and with $|V(T')| = \mathcal{O}(|V(T)|)$.*

Alternative definition through elimination forests. We say that an *elimination forest* of a graph G is a rooted forest F with $V(F) = V(G)$ such that for every edge uv of G, we have $u \preccurlyeq_F v$ or $v \preccurlyeq_F u$. Note that elimination forests underlie the notion of *treedepth* of a graph G, defined as the minimum height of an elimination forest of G; however, since the properties of treedepth are not examined in this thesis, we choose not to explore any further introductory results on treedepth here.

If F is an elimination forest of G, then with every vertex u we can associate its *reachability set* defined as $\mathsf{Reach}_F(u) := N(\mathsf{desc}_F[u])$. Note that as F is an elimination forest, $\mathsf{Reach}_F(u)$ consists of strict ancestors

of u, that is, $\mathsf{Reach}_F(u) \subseteq \mathsf{anc}_F[u] \setminus \{u\}$. So in other words, $\mathsf{Reach}_F(u)$ comprises all strict ancestors of u that have a neighbor among the descendants of u. It can be easily seen that if F is an elimination forest of G, then endowing F with a bag function $u \mapsto \{u\} \cup \mathsf{Reach}_F(u)$ yields a tree decomposition of G. As proved in [BP22], for every graph G there is an elimination forest F of G such that the tree decomposition obtained from F in this way has optimum width, that is, width equal to the treewidth of G.

We then state and prove a variant of the Bodlaender–Hagerup lemma for elimination forests. The definition and the proof is sourced from [KNPS24], though the techniques used in the proof should be considered standard.

Lemma 2.3.4 ([KNPS24])**.** *Let G be a graph on n vertices given together with a tree decomposition of width w with $\mathcal{O}(n)$ nodes. Then, in time $\mathcal{O}(wn \log n)$, one can compute an elimination forest F of G with the following properties:*

1. *F has height $\mathcal{O}(w \cdot \log n)$;*
2. *for each $u \in V(F)$, we have $|\mathsf{Reach}_F(u)| = \mathcal{O}(w)$;*
3. *for each $u \in V(F)$, the graph $G[\mathsf{desc}_F[u]]$ is connected.*

Proof. Recall we are given a tree decomposition, say (T_0, bag_0), of G, of width w and with $\mathcal{O}(n)$ nodes. Using Theorem 2.3.3, we turn it into a tree decomposition (T, bag) of G of width at most $3w + 2$ and height $\mathcal{O}(\log n)$, and root it at an arbitrary node r. Now, let us turn this tree decomposition into an elimination forest by "straightening" each bag. Let $\mu : V(T) \to 2^{V(G)}$ be the function that given a node t of T, returns the *margin* of t: the subset of vertices $v \in \mathsf{bag}(t)$ such that $v \notin \mathsf{bag}(t')$ for any strict ancestor t' of t. In other words, $v \in \mu(t)$ if and only if t is the shallowest node whose bag contains v.

Now, let us define an elimination forest F of G by defining the parent relation. Let us order the vertices of each $\mu(t)$ arbitrarily; say that $\mu(t)$ is ordered $v_1^t, v_2^t, \ldots, v_{|\mu(t)|}^t$. For $2 \leqslant i \leqslant |\mu(t)|$ we let v_{i-1}^t be the parent of v_i^t in F. If t is the root of T, then we let v_1^t to be the root of F. Otherwise, we set the parent of v_1^t to $v_{|\mu(t')|}^{t'}$, where t' is the closest ancestor of t such that $\mu(t')$ is nonempty. One can readily check that this in fact yields a valid elimination forest of height no larger than $\mathcal{O}(w \cdot \log n)$.

As for the property that $|\mathsf{Reach}_F(u)| = \mathcal{O}(w)$, observe that if $u \in \mu(t)$ for some $t \in V(T)$ (as noted before, t is determined uniquely), then $\mathsf{desc}_F[u] \subseteq \mu(\mathsf{desc}_T[t])$, i.e., for every $v \in \mathsf{desc}_F[u]$, t is an ancestor of all nodes whose bags contain v. Therefore, $N_G[\mathsf{desc}_F[u]] \subseteq \mathsf{bag}(\mathsf{desc}_T[t])$. By the construction of F we have that $\mathsf{bag}(\mathsf{desc}_T[t]) - \mathsf{bag}(t) \subseteq \mathsf{desc}_F[u]$, which implies $\mathsf{Reach}_F(u) = N_G(\mathsf{desc}_F[u]) \subseteq \mathsf{bag}(t)$. We conclude that $|\mathsf{Reach}_F(u)| \leqslant 3w + 3$, as required.

However, such F does not have to necessarily fulfill the third condition. Fortunately, the problem is easy to fix in a standard way while maintaining the satisfaction of properties 1 and 2. We create a forest F' on vertex set $V(G)$, where $\mathsf{parent}_{F'}(u)$ is defined as the deepest vertex of $\mathsf{Reach}_F(u)$ in F. Note that $\mathsf{Reach}_F(u) \subseteq \mathsf{anc}_F[u]$ and $\mathsf{anc}_F(u)$ forms a path in F, so such vertex is well defined; as a special case, if $\mathsf{Reach}_F(u)$ is empty, then we let u be a root of F. One can readily check that F' defined in this way is a valid elimination forest of G, still satisfies properties 1 and 2, and additionally satisfies that for each $u \in V(F')$ we have that $G[\mathsf{desc}_{F'}[u]]$ is connected.

The proof above is algorithmic: The decomposition (T, bag) can be computed in time $\mathcal{O}(w \cdot n)$. Afterwards, the margins $\mu(t)$ for $t \in V(T)$ can be found in total time $\mathcal{O}(w \cdot n)$ by determining, for each $v \in V(G)$, the shallowest node $t \in V(T)$ whose bag contains v. The rest of the construction of F' is trivial to implement in time $\mathcal{O}(n \log n \cdot w)$. □

2.4 Rankwidth and cliquewidth

Rankwidth. We now introduce rankwidth formally. Let G be an undirected graph. A rank decomposition of G is a pair $\mathcal{T} = (T, \lambda)$, where T is a cubic tree (where all nonleaf vertices have degree exactly 3) with at least two leaves, and $\lambda : V(G) \to \vec{L}(T)$ is a bijection between the set of vertices of G and the set of leaf edges of T.[9] We may equivalently consider T to be a rooted binary tree by subdividing an arbitrary edge of T and rooting the tree at a newly created vertex.

For an oriented edge $\vec{xy} \in \vec{E}(T)$, define $\mathcal{L}(\mathcal{T})[\vec{xy}] \subseteq V(G)$ as the set of vertices assigned to the leaf edges of T that are closer to x than y: $\mathcal{L}(\mathcal{T})[\vec{xy}] := \{v \in V(G) \mid \lambda(v) \text{ is a predecessor of } \vec{xy}\}$. Then, for an edge $xy \in E(T)$, we define the width of xy in $\mathcal{T}$ as the GF(2)-rank of the adjacency matrix of the bipartite induced subgraph

$$G[\mathcal{L}(\mathcal{T})[\vec{xy}], \mathcal{L}(\mathcal{T})[\vec{yx}]].$$

[9] Note that this definition precludes the existence of rank decompositions of graphs containing at most one vertex; in this thesis, we will not be concerned about extending the definition to such graphs.

Now, the width of a rank decomposition of G is the maximum width of any edge of the decomposition, and then the rankwidth of G is the minimum possible width of a rank decomposition of G.

The following lemma is an analog of the Bodlaender–Hagerup lemma for rankwidth; that is, we prove that each rank decomposition can be made into a logarithmic-depth rank decomposition of the same graph without worsening the width of the decomposition too much.

Lemma 2.4.1 ([CK07]). *Let (T, λ) be a rank decomposition of a graph G of width k. Then there exists a rooted rank decomposition of G of height $\mathcal{O}(\log |V(T)|)$ and width at most $2k$.*

We prove a slightly stronger result in Section 4.2: First, we restate Lemma 2.4.1 in terms of rank decompositions of *partitioned graphs*, which we also define in Section 4.2. Next, for our applications we require the proof to be efficient – in fact, we construct the logarithmic-depth decomposition in $\mathcal{O}(|V(T)| \log |V(T)|)$ time.

Cliquewidth. We move to *cliquewidth* – a graph parameter functionally equivalent to rankwidth. As mentioned before, the notion of cliquewidth actually predates rankwidth – it was introduced by Courcelle, Engelfriet, and Rozenberg [CER93] and defined in its modern form by Courcelle in [Cou95]. In the description below, we follow the treatment of cliquewidth in [KS24], which defines cliquewidth similarly to [CMR00].

Let $k \in \mathbb{N}$. A tuple $\mathcal{G} = (G, V_1, \ldots, V_k)$ is a *k-graph* if G is a graph and $V_1, \ldots, V_k$ are disjoint subsets of $V(G)$ whose union equals $V(G)$ (they are not a partition because they are indexed by $[k]$ and allowed to be empty). We define three types of operations for constructing k-graphs. First, the disjoint union of two k-graphs $\mathcal{G}^1 = (G^1, V_1^1, \ldots, V_k^1)$ and $\mathcal{G}^2 = (G^2, V_1^2, \ldots, V_k^2)$ where G^1 and G^2 are disjoint is defined as

$$\mathcal{G}_1 \oplus \mathcal{G}_2 = (G^1 \cup G^2, V_1^1 \cup V_1^2, \ldots, V_k^1 \cup V_k^2), \text{ where } G^1 \cup G^2 \text{ is the union of } G^1 \text{ and } G^2.$$

Then, $\eta(i,j)(\mathcal{G})$ for $i, j \in [k]$ with $i \neq j$ denotes the k-graph obtained from $\mathcal{G} = (G, V_1, \ldots, V_k)$ by adding all possible edges between V_i and V_j, i.e.,

$$\eta(i,j)(\mathcal{G}) = (G', V_1, \ldots, V_k), \text{ where } V(G') = V(G) \text{ and } E(G') = E(G) \cup \{uv \mid u \in V_i \wedge v \in V_j\}.$$

Then, $\pi(i,j)(\mathcal{G})$ for $i, j \in [k]$ with $i \neq j$ denotes the k-graph obtained from $\mathcal{G}$ by renaming i into j, i.e.,

$$\pi(i,j)(\mathcal{G}) = (G, V_1', \ldots, V_k'), \text{ where } V_i' = \emptyset,\ V_j' = V_i \cup V_j, \text{ and } V_l' = V_l \text{ for } l \in [k] \setminus \{i, j\}.$$

A graph has cliquewidth at most k if it can be constructed from single-vertex k-graphs by using those operations.

It was observed by Oum and Seymour that cliquewidth and rankwidth are functionally tied to each other, so every class of graphs of bounded cliquewidth has bounded rankwidth and vice versa:

Theorem 2.4.2 ([OS06]). *If a graph has rankwidth k, then its cliquewidth is at least k and at most $2^{k+1} - 1$.*

While the proof of Oum and Seymour is effective, in Section 4.10 we will give an optimized counterpart of their proof: Given a rank decomposition of a graph G of width k decorated with suitable annotations of the edges and nodes of the decomposition, we can produce a $(2^{k+1} - 1)$-expression of G in time $\mathcal{O}_k(|V(G)|)$.

2.5 Twin-width

We begin by giving a definition of twin-width of graphs, before discussing the generalizations of twin-width to the setting of matrices.

Let G be a graph and assume G has $n \geqslant 1$ vertices. A *contraction sequence* of G is a sequence of partitions $\mathcal{P}_n, \mathcal{P}_{n-1}, \ldots, \mathcal{P}_1$ of $V(G)$ such that:

- $\mathcal{P}_n = \{\{v\} \mid v \in V(G)\}$ is the finest partition of $V(G)$ where every vertex is in a separate part;
- $\mathcal{P}_1 = \{V(G)\}$ is the coarsest partition of $V(G)$ consisting of just one set;
- for each $i \in [p-1]$, $\mathcal{P}_i$ is a *contraction* of $\mathcal{P}_{i+1}$: that is, $\mathcal{P}_i$ is created from $\mathcal{P}_{i+1}$ by merging two sets into one.

The *error value* of a partition $\mathcal{P}$ of $V(G)$ is the least integer d such that for every part $P \in \mathcal{P}$, there are at most d parts $Q \in \mathcal{P}$ other than P such that (P, Q) is not pure. Then a contraction sequence $\mathcal{P}_n, \mathcal{P}_{n-1}, \ldots, \mathcal{P}_1$ is a *d-sequence* if all partitions in the sequence have error values at most d. Finally, the *twin-width* of a graph G is the minimum value of d such that G admits a d-sequence.

2.5.1 Twin-width of matrices

We now show a generalization of twin-width to matrices. The description below is sourced from [PSZ22] and defines the concepts of *d-twin-ordered* matrices and twin-width of matrices that were first defined in [BKTW20].

Let M be a matrix. If $(\mathcal{R},\mathcal{C})$ is a division of M, then a *contraction* of $(\mathcal{R},\mathcal{C})$ is any division $(\mathcal{R}',\mathcal{C}')$ obtained from $(\mathcal{R},\mathcal{C})$ by either merging two consecutive row blocks $R_1,R_2\in\mathcal{R}$ into a single row block $R_1\cup R_2$, or merging two consecutive column blocks $C_1,C_2\in\mathcal{C}$ into a single column block $C_1\cup C_2$. A *contraction sequence* for M is a sequence of divisions

$$(\mathcal{R}_p,\mathcal{C}_p),(\mathcal{R}_{p-1},\mathcal{C}_{p-1}),\ldots,(\mathcal{R}_1,\mathcal{C}_1),$$

such that

- $(\mathcal{R}_p,\mathcal{C}_p)$ is the finest division where every row and every column is in a separate block;
- $(\mathcal{R}_1,\mathcal{C}_1)$ is the coarsest partition where all rows are in a single row block and all columns are in a single column block; and
- for each $i\in[p-1]$, $(\mathcal{R}_{i+1},\mathcal{C}_{i+1})$ is a contraction of $(\mathcal{R}_i,\mathcal{C}_i)$.

Note that thus, p has to be equal to the sum of the dimension of M, minus 1. Finally, for a division $(\mathcal{R},\mathcal{C})$ of M, the *error value* of $(\mathcal{R},\mathcal{C})$ is the least d such that in $(\mathcal{R},\mathcal{C})$, every row block and every column block contains at most d nonconstant zones. Then M is said to be *d-twin-ordered* if it admits a *d-sequence*, that is, a contraction sequence where every division has error value at most d. The *twin-width* of a binary matrix M is the least d such that one can permute the rows and columns of M so that the obtained matrix is d-twin-ordered.

The twin-width of graphs and matrices are closely tied to each other:

Observation 2.5.1. *The following properties hold:*

- *If a graph G has twin-width d, then there is a total order $<$ of $V(G)$ such that the adjacency matrix of G, with rows and columns ordered according to $<$, is $(d+2)$-twin-ordered.*
- *If a graph G has a total order $<$ such that the adjacency matrix of G, with rows and columns ordered according to $<$, is t-twin-ordered, then the twin-width of G is at most* $\max(0,2t-1)$.

Even though these relations should be considered folklore, we could not find their proofs in the literature. Hence we give the proof below.

Proof. Let $\mathcal{P}_n,\mathcal{P}_{n-1},\ldots,\mathcal{P}_1$ be a d-sequence of G. Without loss of generality assume that $V(G)$ has a total order $<$ so that every part $P\in\mathcal{P}_i$ for every $i\in[n]$ is an interval with respect of $<$. Let M be the adjacency matrix of G, with rows and columns sorted according to $<$. We claim that M is $(d+2)$-twin-ordered. Indeed, consider the contraction sequence $(\mathcal{R}_{2n-1},\mathcal{C}_{2n-1}),\ldots,(\mathcal{R}_1,\mathcal{C}_1)$ of M constructed as follows: Initialize $\mathcal{R}_{2n-1}=\mathcal{C}_{2n-1}=V(G)$. Then whenever $\mathcal{P}_i$ is constructed from $\mathcal{P}_{i+1}$ by merging two sets $X,Y\in\mathcal{P}_{i+1}$ into $X\cup Y\in\mathcal{P}_i$, we define $(\mathcal{R}_{2i},\mathcal{C}_{2i})$ and $(\mathcal{R}_{2i-1},\mathcal{C}_{2i-1})$ from $(\mathcal{R}_{2i+1},\mathcal{C}_{2i+1})$ by first contracting the row blocks X and Y, and then contracting the column blocks X and Y, so that $\mathcal{R}_{2i-1}=\mathcal{C}_{2i-1}=\mathcal{P}_i$ for all $i\in[n]$.

Observe that a single contraction can decrease the error value of a partition by at most 1 (though it may increase the error value unboundedly); hence it is enough to show that for each $i\in[n]$, the partition $(\mathcal{R}_{2i-1},\mathcal{C}_{2i-1})=(\mathcal{P}_i,\mathcal{P}_i)$ of M has error value at most $d+1$. Indeed, whenever $R,C\in\mathcal{P}_i$ are so that the zone $M[R,C]$ is nonconstant, then either $R=C$, or $R\neq C$ and the pair (R,C) is not pure in G (and for each R, there are at most d such parts C in $\mathcal{P}_i$). Hence for each $R\in\mathcal{P}_i$, there are at most $d+1$ parts $C\in\mathcal{P}_i$ such that $M[R,C]$ is nonconstant. And, symmetrically, for each $C\in\mathcal{P}_i$, there are at most $d+1$ parts $R\in\mathcal{P}_i$ with nonconstant zone $M[R,C]$.

Conversely, assume that G has a t-twin-ordered adjacency matrix M, where the rows and columns of M are ordered according to the total order $<$ on $V(G)$. Hence there exists a t-sequence $(\mathcal{R}_{2n-1},\mathcal{C}_{2n-1})$, $\ldots,(\mathcal{R}_1,\mathcal{C}_1)$ of M witnessing that M is t-twin-ordered. Note here that each set in each $\mathcal{R}_i$ and $\mathcal{C}_i$ is an interval of $V(G)$ with respect to $<$. Consider now the sequence $\mathcal{P}_{2n-1},\mathcal{P}_{2n-2},\ldots,\mathcal{P}_1$ of partitions of $V(G)$ defined as follows:

$$\mathcal{P}_i=\{R\cap C\mid R\in\mathcal{R}_i,\,C\in\mathcal{C}_i,\,R\cap C\neq\emptyset\}.$$

It can be easily verified that each $\mathcal{P}_i$ is a partition of $V(G)$. Moreover, since the sets in all partitions $\mathcal{R}_i$ and $\mathcal{C}_i$ are intervals with respect to $<$, so are the sets in $\mathcal{P}_i$; and for each $i\in[2n-2]$, we have that $\mathcal{P}_i=\mathcal{P}_{i+1}$

or that $\mathcal{P}_i$ is constructed from $\mathcal{P}_{i+1}$ by merging two sets $X, Y \in \mathcal{P}_{i+1}$. Finally, $\mathcal{P}_{2n-1} = \{\{v\} \mid v \in V(G)\}$ and $\mathcal{P}_1 = \{V(G)\}$. So a contraction sequence of G can be constructed from the sequence $\mathcal{P}_{2n-1}, \dots, \mathcal{P}_1$ by filtering out the duplicate partitions. It remains to show that the error value of each partition $\mathcal{P}_i$ is at most $\max(0, 2t-1)$. So let $R \cap C \in \mathcal{P}_i$ with $R \in \mathcal{R}_i$ and $C \in \mathcal{C}_i$ and suppose there are at least $u := \max(1, 2t)$ other sets $R_1 \cap C_1, \dots, R_u \cap C_u \in \mathcal{P}_i$ (with $R_j \in \mathcal{R}_i$ and $C_j \in \mathcal{C}_i$ for $j \in [u]$) such that $(R \cap C, R_j \cap C_j)$ is not pure for all $j \in [u]$. Without loss of generality, assume that for all $j \in [u-1]$, the set $R_j \cap C_j$ is an interval of $V(G)$ that is earlier in $<$ than $R_{j+1} \cap C_{j+1}$. Then either $R_j \neq R_{j+1}$ and R_j is earlier in $<$ than R_{j+1}, or $C_j \neq C_{j+1}$ and C_j is earlier in $<$ than C_{j+1}. We infer that either there are at least $t+1$ distinct sets in the family $\{R_1, \dots, R_u\}$, or there are at least $t+1$ distinct sets in $\{C_1, \dots, C_u\}$; without loss of generality assume the former. But then the zone $M[R_j, C]$ is nonconstant for each $j \in [u]$, implying that the column block $C \in \mathcal{C}_i$ contains at least $t+1$ nonconstant zones in the division $(\mathcal{R}_i, \mathcal{C}_i)$ – a contradiction. □

We remark here that several similar – but slightly different – generalizations of twin-width to matrices have appeared in the literature, for instance *symmetric twin-width* [BKTW20] (defined so that the twin-width of any graph is exactly equal to the symmetric twin-width of its adjacency matrix), and twin-width of so-called *ordered matrices* [BGdM+21], related to the setting of *ordered graphs* presented in Section 1.1.3. However, in this thesis we will not use these alternative definitions.

We give another observation about binary twin-ordered matrices: The set of entries 1 can be decomposed into a small number of rectangles. Formally, for a binary matrix M, a *rectangle decomposition* of M is a set $\mathcal{K}$ of pairwise disjoint rectangular submatrices (i.e., zones induced by some row block and some column block) such that every submatrix in $\mathcal{K}$ is entirely filled with 1s and there is no entry 1 outside the submatrices in $\mathcal{K}$. The following lemma is stated and proved in the graph setting in [BGK+21b], and then adapted to the matrix setting below in [PSZ22]; we adapt the proof here to the matrix setting.

Lemma 2.5.2 (Statement and proof from [PSZ22])**.** *Let M be an $n \times n$ binary matrix that is d-twin-ordered. Then M admits a rectangle decomposition $\mathcal{K}$ with $|\mathcal{K}| \leqslant \max(1, d(2n-2))$.*

Proof. Let $(\mathcal{R}_0, \mathcal{C}_0), \dots, (\mathcal{R}_{2n-2}, \mathcal{C}_{2n-2})$ be a contraction sequence for M with error value at most d. Let $\mathcal{S}_i$ be the set of zones of $(\mathcal{R}_i, \mathcal{C}_i)$, and let

$$\mathcal{S} = \bigcup_{i=0}^{2n-2} \mathcal{S}_i.$$

Note that $\mathcal{S}$ is a *laminar* family, that is, every two submatrices in $\mathcal{S}$ are either disjoint or one is contained in the other.

Let $\mathcal{K}$ be the subfamily of $\mathcal{S}$ consisting of those submatrices that are entirely filled with 1s, and are inclusion-wise maximal in $\mathcal{S}$ subject to this property. Note that every entry 1 in M is contained in some member of $\mathcal{K}$, for the zone of $(\mathcal{R}_0, \mathcal{C}_0)$ in which this entry is contained is a 1×1 submatrix entirely filled with 1s. Since $\mathcal{S}$ is laminar, it follows that $\mathcal{K}$ is a rectangle decomposition of M. So it remains to argue that $|\mathcal{K}| \leqslant \max(1, d(2n-2))$.

Consider any $A \in \mathcal{K}$ and let i be the largest index such that $A \in \mathcal{S}_i$. We may assume that $i < 2n-2$, for otherwise the matrix M is entirely filled with 1s and then $|\mathcal{K}| = 1$. By maximality, A is contained in a nonconstant zone $B \in \mathcal{S}_{i+1}$ that resulted from merging A with another adjacent zone $A' \in \mathcal{S}_i$, which is not entirely filled with 1s. In particular, B lies in the unique row block or column block of $(\mathcal{R}_{i+1}, \mathcal{C}_{i+1})$ that resulted from merging two row blocks or two column blocks of $(\mathcal{R}_i, \mathcal{C}_i)$. There can be at most d nonconstant zones in this row/column block of $(\mathcal{R}_{i+1}, \mathcal{C}_{i+1})$, and B is one of them. We infer that i can be the largest index satisfying $A \in \mathcal{S}_i$ for at most d different submatrices $A \in \mathcal{K}$. Since this applies to every index $i \in \{0, 1, \dots, 2n-3\}$, we conclude that $|\mathcal{K}| \leqslant d(2n-2)$. □

2.5.2 Mixed minors and matrix obstructions for twin-width

We now formally introduce the obstructions for bounded twin-width in the form of *mixed minors*. In the following exposition, we mostly follow [PSZ22, PS23].

A matrix is *constant* if it contains only entries 0 or only entries 1; *horizontal* if all its columns equal each other (or equivalently, each row is constant); *vertical* if all its rows equal each other (or equivalently, each column is constant); and *mixed* if it is neither horizontal nor vertical (or equivalently, it contains at least two different rows and at least two different columns). The following fact about mixed matrices is standard.

Lemma 2.5.3 ([BKTW20]). *A matrix is mixed if and only if it contains a* corner *– a 2×2 mixed zone – as a submatrix.*

We now introduce the following definition of *grid* and *mixed minors* in matrices – intuitively, "complicated substructures" in matrices that preclude the matrix from having small twin-width. Let M be a matrix with entries 0 and 1. Recall that a t-division of M is a partition of rows and columns of M into t row blocks and t column blocks. A *t-grid minor* in M is a t-division of M where every zone contains at least one entry 1. A *t-mixed minor* in M is a t-division of M where every zone is mixed. We say that M is *t-grid-free* if M does not contain a t-grid minor, and *t-mixed-free* if M does not contain a t-mixed minor.

The following result of Marcus and Tardos asserts that if a matrix has a large density of entries 1, then it contains a large grid minor:

Theorem 2.5.4 ([MT04]). *For every $t \in \mathbb{N}$ there exists $c_t \in \mathbb{N}$ such that the following holds. Suppose M is an $n \times m$ binary matrix with at least $c_t \cdot \max(n, m)$ entries* 1*. Then M has a t-grid minor.*

The currently best upper bound on c_t is $\frac{8}{3}(t+1)^2 2^{4t}$, due to Cibulka and Kynčl [CK19]. We adopt the constant c_t in the notation for the remainder of this section.

Note that there exist matrices with a large density of entries 1 that exclude large mixed minors since the constant-1 matrix is 1-mixed-free. However, in [BKTW20], Bonnet et al. used the result of Marcus and Tardos above to show that, intuitively, large mixed minors are the canonical obstacles for having bounded twin-width.

Theorem 2.5.5 ([BKTW20]). *Let M be a binary matrix. Then the following implications hold:*

- *If M is d-twin-ordered, then M is $(2d+2)$-mixed-free.*
- *If M is t-mixed-free, then M has twin-width at most k_t, where $k_t = 2^{2^{O(t)}}$ is a constant depending only on t.*

Note that the conclusion of the second implication of Theorem 2.5.5 is only a bound on the twin-width of a matrix since the rows and columns of the matrix might still need to be permuted to be k_t-twin-ordered. In this thesis we will only rely on the first implication of Theorem 2.5.5 – being d-twin-ordered implies $(2d+2)$-mixed-freeness.

From the Marcus–Tardos theorem we infer that in t-mixed-free matrices, all ℓ-divisions have only $\mathcal{O}_t(\ell)$ mixed zones:

Lemma 2.5.6. *Let M be a t-mixed free matrix, and let $(\mathcal{R}, \mathcal{C})$ be an ℓ-division of M, for some integer ℓ. Then $(\mathcal{R}, \mathcal{C})$ has at most $c_t \cdot \ell$ mixed zones.*

Proof. Construct an $\ell \times \ell$ matrix A by taking the division $(\mathcal{R}, \mathcal{C})$ and substituting each mixed zone with a single entry 1, and each nonmixed zone with a single entry 0. Observe that A may have at most $c_t \cdot \ell$ entries 1, for otherwise, by Theorem 2.5.4, A would contain a t-grid minor, which would correspond to a t-mixed minor in M. Hence $(\mathcal{R}, \mathcal{C})$ may have at most $c_t \cdot \ell$ mixed zones. □

The next lemma is essentially proven in [BKTW20] but never stated explicitly. So we include a proof from [PSZ22] for completeness.

Lemma 2.5.7 (Implicit in [BKTW20], proof from [PSZ22]). *A t-mixed-free $n \times n$ matrix contains at most $2c_t(n+2)$ corners.*

Proof. Let M be a t-mixed-free $n \times n$ matrix. Consider the $\lceil n/2 \rceil$-division $(\mathcal{R}, \mathcal{C})$ of M, in which every row block consists of rows with indices $2i-1$ and $2i$ for some $i \in \{1, \ldots, \lfloor n/2 \rfloor\}$, possibly except the last block that consists only of row n in case n is odd, and similarly for column blocks. By Lemma 2.5.6, $(\mathcal{R}, \mathcal{C})$ has at most $c_t \lceil n/2 \rceil \leqslant c_t(n/2+1)$ mixed zones, which implies that M has at most $c_t(n/2+1)$ corners in which the bottom-right entry is in the intersection of an even-indexed row and an even-indexed column. Call such corners of *type* 00; corners of types 01, 10, and 11 are defined analogously. By suitably modifying the pairing of rows and columns in $(\mathcal{R}, \mathcal{C})$, we can analogously prove that the number of corners of each of the remaining three types is also bounded by $c_t(n/2+1)$. Hence, in total there are at most $4c_t(n/2+1) = 2c_t(n+2)$ corners in M. □

For Chapter 6 (compact oracle for twin-width), we will need a variant of Lemma 2.5.6 that focuses on mixed borders between neighboring zones. Here, two different zones in a division $(\mathcal{R}, \mathcal{C})$ are called *adjacent* if they are either in the same row block and consecutive column blocks, or in the same column

block and consecutive row blocks. A *mixed cut* in $(\mathcal{R},\mathcal{C})$ is a pair of adjacent zones such that there is a corner that crosses the boundary between them, i.e., has two entries in each of them. A *split corner* in $(\mathcal{R},\mathcal{C})$ is a corner intersecting four different zones, i.e., it has an entry in four different zones.

The proof of the following observation is again essentially present in [BKTW20].

Lemma 2.5.8 (Implicit in [BKTW20], proof from [PSZ22])**.** *Let M be a t-mixed free matrix, and let $(\mathcal{R},\mathcal{C})$ be an ℓ-division of M, for some integer ℓ. Then $(\mathcal{R},\mathcal{C})$ has at most $c_t \cdot (\ell+2)$ mixed cuts and at most $2c_t \cdot (\ell+1)$ split corners.*

Proof. Let $(\mathcal{R}^{00},\mathcal{C}^{00})$ be the division obtained from $(\mathcal{R},\mathcal{C})$ by merging the row blocks indexed $2i-1$ and $2i$ into a single row block, and merging the column blocks indexed $2i-1$ and $2i$ into a single column block, for each $i \in \{1,\ldots,\lfloor \ell/2 \rfloor\}$. Obtain divisions $(\mathcal{R}^{10},\mathcal{C}^{10})$, $(\mathcal{R}^{01},\mathcal{C}^{01})$, and $(\mathcal{R}^{11},\mathcal{C}^{11})$ in a similar manner, where if the first number in the superscript is 1 then we merge row blocks $2i$ and $2i+1$ for each $i \in \{1,\ldots,\lceil \ell/2 \rceil - 1\}$ instead, and if the second number in the superscript is 1 then we merge column blocks $2i$ and $2i+1$ for each $i \in \{1,\ldots,\lceil \ell/2 \rceil - 1\}$ instead.

Observe that for every mixed cut of $(\mathcal{R},\mathcal{C})$, the two zones in the mixed cut end up in the same zone in either $(\mathcal{R}^{00},\mathcal{C}^{00})$ or in $(\mathcal{R}^{11},\mathcal{C}^{11})$, rendering this zone mixed. However, by Lemma 2.5.6, $(\mathcal{R}^{00},\mathcal{C}^{00})$ and $(\mathcal{R}^{11},\mathcal{C}^{11})$ have at most $c_t \cdot (\ell/2+1)$ mixed zones. It follows that $(\mathcal{R},\mathcal{C})$ has at most $2c_t \cdot (\ell/2+1) = c_t \cdot (\ell+2)$ mixed cuts. The bound on the number of split corners follows from the same argument combined with the observation that every split corner in $(\mathcal{R},\mathcal{C})$ is entirely contained in a single zone of exactly one of divisions $(\mathcal{R}^{00},\mathcal{C}^{00})$, $(\mathcal{R}^{10},\mathcal{C}^{10})$, $(\mathcal{R}^{01},\mathcal{C}^{01})$, and $(\mathcal{R}^{11},\mathcal{C}^{11})$. □

2.6 Logic

In this thesis we will rely on several results pertaining to the properties of formal logic systems in finite graphs. In this section, we will formally introduce several variants of first- and second-order logic in the setting of graphs and show how they are related to the classes of graphs of bounded tree-, rank-, and twin-width. We begin with the most expressive of the logic formalisms used in the thesis – CMSO_2.

2.6.1 CMSO_2

CMSO_2 is the monadic second-order logic on graphs with quantification over edge subsets and modular counting predicates. This logic is typically associated with graphs of bounded treewidth; see [CFK+15, Section 7.4] for an introduction suited for an algorithm designer. Formulas of CMSO_2 are evaluated in graphs and there are variables of four different sorts: for single vertices, for single edges, for vertex subsets, and for edge subsets. The latter two sorts are called *monadic*. The atomic formulas of CMSO_2 are of the following forms:

- *Equality:* $x = y$, where x, y are both either single vertex/edge variables.
- *Membership:* $x \in X$, where x is a single vertex/edge variable and X is a monadic vertex/edge variable.
- *Incidence:* $\mathsf{inc}(x,e)$, where x is a single vertex variable and e is a single edge variable.
- *Modular counting:* $|X| \equiv a \pmod m$, where a and m are integers, $m > 0$.

The semantics of the above is as expected. Then CMSO_2 consists of all formulas that can be obtained from atomic formulas using the following constructs: standard boolean connectives, negation, and quantification over all sorts of variables, both existential and universal. Thus, a formula of CMSO_2 may contain variables that are not bound by any quantifier; these are called *free variables*. A formula without quantifiers is called *quantifier-free*, while a formula without free variables is a *sentence*. For a sentence φ and a graph G, we write $G \models \varphi$ to signify that φ is satisfied in G (read G *is a model of* φ). When $\varphi(X_1,\ldots,X_k)$ is a formula with k free variables, then we similarly write $G \models \varphi(A_1,\ldots,A_k)$ or $(G,A_1,\ldots,A_k) \models G$ to say that G satisfies the formula φ with $X_1 = A_1,\ldots,X_k = A_k$.

Then we extend CMSO_2 to capture *optimization problems* by defining a variant of CMSO_2 called $\mathsf{LinCMSO}_2$; this is a straightforward adaptation of the definition of $\mathsf{LinCMSO}_1$ from [KS24]. A $\mathsf{LinCMSO}_2$ formula with p free (vertex or edge) set variables is a pair (φ, f), where $\varphi = \varphi(X_1, X_2, \ldots, X_{p+q})$ is a CMSO_2 formula with $p+q$ free variables for $q \geqslant 0$, and $f : \mathbb{Z}^q \to \mathbb{Z}$ is an affine integer function defined by $q+1$ integers $c_0,\ldots,c_q$ so that $f(x_1,\ldots,x_q) = c_0 + c_1x_1 + \ldots + c_qx_q$. The value of (φ, f) on a tuple $(G,A_1,\ldots,A_p)$ is the maximum possible value of $f(|A_{p+1}|,\ldots,|A_{p+q}|)$, where $(G,A_1,\ldots,A_{p+q}) \models \varphi$;

and, for each $i \in [q]$, $A_{p+i} \subseteq V(G)$ if X_{p+i} is a vertex set variable, or $A_{p+i} \subseteq E(G)$ is X_{p+i} is an edge set variable. If no such sets $A_{p+1}, \ldots, A_{p+q}$ exist, we put $\perp$ as the value of (φ, f). We note that even though this naturally defines only maximization problems, we can define minimization problems by using negative coefficients. We define the length $|(\varphi, f)|$ of (φ, f) to be $|\varphi| + \sum_{i=0}^{p} |c_i|$.

In the fragment MSO_2 of CMSO_2, we only consider the formulas of CMSO_2 without modular counting predicates.

Recall that Courcelle's theorem [Cou90] states that given a graph G of treewidth k and a CMSO_2 formula φ, it can be decided whether $G \models \varphi$ in time $f(k, \varphi) \cdot n$, where n is the vertex count of G and f is a computable function. Also, in a sense, bounded treewidth forms a *"boundary of tractability"* for the logic CMSO_2 [See91, KT10, Kre12].

2.6.2 CMSO_1

We now turn to CMSO_1 – the monadic second-order logic on graphs *without* quantification over edge subsets, but including modular counting predicates. Contrary to CMSO_2, the CMSO_1 logic only permits variables of two sorts: for single vertices and for vertex subsets. The atomic formulas support testing for equality, membership, and modular counting, analogously and with the same semantics as in the definition of CMSO_2. Additionally, such formulas can verify *adjacency* through an atomic formula $\mathsf{adj}(x, y)$ for two single-vertex variables x, y.

As before, in MSO_1, we only allow formulas of CMSO_1 that exclude modular counting predicates.

Next we define an *optimization* variant of CMSO_1, which we call $\mathsf{LinCMSO}_1$. The following definition follows [KS24] and is inspired by the analogous extensions by Courcelle, Makowsky and Rotics [CMR00] and Courcelle and Engelfriet [CE12].

A $\mathsf{LinCMSO}_1$ formula with p free set variables is a pair (φ, f), where φ is a CMSO_1 formula with $p + q$ free variables for $q \geqslant 0$, and $f \colon \mathbb{Z}^q \to \mathbb{Z}$ an affine integer function defined by $q + 1$ integers $c_0, \ldots, c_q$ so that $f(x_1, \ldots, x_q) = c_0 + c_1 x_1 + \ldots + c_q x_q$. Then, the value of (φ, f) on a tuple $(G, X_1, \ldots, X_p)$ is the maximum value of $f(|X_{p+1}|, \ldots, |X_{p+q}|)$, where $X_{p+1}, \ldots, X_{p+q} \subseteq V(G)$ and $(G, X_1, \ldots, X_{p+q}) \models \varphi$. As in the case of $\mathsf{LinCMSO}_2$, if no such sets $X_{p+1}, \ldots, X_{p+q}$ exist, then the value is $\perp$. The length $|(\varphi, f)|$ of (φ, f) is $|\varphi| + \sum_{i=0}^{p} |c_i|$.

Similarly to how CMSO_2 is usually associated with treewidth, CMSO_1 is deeply linked to the notion of rankwidth (or equivalently, cliquewidth). Courcelle, Makowsky, and Rotics generalized Courcelle's theorem to cliquewidth by showing that, given a graph G together with its k-expression and a CMSO_1 formula φ, we can decide whether $G \models \varphi$ in time $\mathcal{O}_{k,\varphi}(n)$, where again n is the number of vertices of G [CMR00]. An analogous result for the optimization variant $\mathsf{LinCMSO}_1$ also holds [CMR00, CMR01]: When a graph G is supplied together with its k-expression and a $\mathsf{LinCMSO}_1$ formula (φ, f), the value of (φ, f) on G can be computed in time $\mathcal{O}_{k,|(\varphi,f)|}(n)$.

2.6.3 FO

Finally, we define the first-order logic (FO) for graphs as follows. Variables of formulas of first-order logic represent single vertices of the graphs. The atomic formulas of FO can test for *equality* of variables as well as the *adjacency* of the vertices represented by the variables. The remainder of the definition follows their monadic second-order counterparts.

As a side note, an ongoing project aims to complete the classification of the hereditary classes of graphs with *efficient* FO *model checking* – i.e., those hereditary classes $\mathcal{C}$ for which there exists an algorithm that, given a graph $G \in \mathcal{C}$ and an FO sentence φ, verifies whether $G \models \varphi$ in time $\mathcal{O}_{\varphi}(|G|^{\mathcal{O}(1)})$. Currently, it is known that such an algorithm exists for classes of graphs of bounded twin-width, as long as a contraction sequence of bounded width of an input graph is provided [BKTW20]. Also, a recent result [DEM+23] proves that *monadically stable* classes of graphs admit efficient FO model checking. Both graph class properties are incomparable: There exist classes of graphs of bounded twin-width that are not monadically stable, and there exist monadically stable classes of graphs that do not have bounded twin-width. It is conjectured [BGdM+21, DMS23] that *monadically dependent* classes of graphs[10] – a property of graphs that generalizes both bounded twin-width and monadic stability – are precisely the classes of graphs that admit efficient FO model checking (under standard hardness assumptions).

[10] Also called *monadically NIP* classes of graphs, where *NIP* stands for the *non-independence property*.

2.6.4 Additional logic preliminaries

It will sometimes be convenient to replace a sequence of variables with a named tuple of vertices; for instance, if we define that $\overline{x}$ is a tuple of variables comprising variables $x_1, \ldots, x_k$, then we can say that $\varphi(\overline{x})$ is a formula with variables $x_1, \ldots, x_k$, and $\varphi(\overline{x}, y)$ is a formula with variables $x_1, \ldots, x_k, y$.

We will sometimes assume that the vertices of the graphs considered in this thesis can be assigned colors. Formally, we can define a finite palette Γ of colors, and then each vertex of the graph can be assigned an arbitrary subset of the colors in Γ. In this setting, the definition of formulas of the logics defined above extends so that the formulas can additionally verify the colors of vertices. Formally, for every color $C \in \Gamma$, we introduce a unary relation symbol C. Then for any graph G and vertex $v \in V(G)$, we define that $C(v)$ is true if and only if v has color C.

Part I

Treewidth and rankwidth

Chapter 3

Dynamic treewidth

In this chapter we give a resolution to the dynamic treewidth problem with subpolynomial amortized time complexity of the updates (Theorem 1.3.1). That is, we present a data structure that for a fully dynamic graph G of treewidth k, maintains a constant-factor-approximate tree decomposition of G. The amortized update time is subpolynomial in n for every fixed k. As a consequence, we prove the dynamic variant of Courcelle's Theorem for treewidth: The satisfaction of any fixed CMSO_2 property φ can be maintained within the same complexity bounds. We recall the statement of the theorem below for convenience.

Theorem 1.3.1 ([KMN+23]). *There is a data structure that for an integer $k \in \mathbb{N}$, fixed upon initialization, and a dynamic graph G, updated by edge insertions and deletions, maintains a tree decomposition of G of width at most $6k+5$ whenever G has treewidth at most k. More precisely, at every point in time the data structure either contains a tree decomposition of G of width at most $6k+5$, or a marker "Treewidth too large", in which case it is guaranteed that the treewidth of G is larger than k. The data structure can be initialized on k and an edgeless n-vertex graph G in time $g(k) \cdot n$, and then every update takes amortized time $2^{f(k)\cdot\sqrt{\log n \log\log n}}$, where $g(k) \in 2^{k^{\mathcal{O}(1)}}$ and $f(k) \in k^{\mathcal{O}(1)}$ are computable functions.*

Moreover, upon initialization the data structure can be also provided a CMSO_2 sentence φ, and it can maintain the information whether φ is satisfied in G whenever the marker "Treewidth too large" is not present. In this case, the initialization time is $g(k,\varphi) \cdot n$ and the amortized update time is $h(k,\varphi) \cdot 2^{f(k)\cdot\sqrt{\log n \log\log n}}$, where g, h, and f are computable functions.

We also restate the analog of Theorem 1.3.1 for optimization problems expressible in monadic second order logic:

Theorem 1.3.3. *The data structure of Theorem 1.3.1 can be also provided a $\mathsf{LinCMSO}_2$ sentence φ upon initialization and can maintain the value of φ in G whenever the marker "Treewidth too large" is not present. The initialization time is $g(k,\varphi) \cdot n$ and the amortized update time is $h(k,\varphi) \cdot 2^{f(k)\cdot\sqrt{\log n \log\log n}}$ for computable functions g, h, f.*

As discussed in the Introduction, the data structure persists even at times when the treewidth grows above k, instead of working under the assumption that this never happens. In fact, throughout this chapter we work in the latter weaker setting, as it can be lifted to the stronger setting discussed in Theorem 1.3.1 in a generic way using the technique of delaying invariant-breaking updates, proposed by Eppstein et al. [EGIS96]; see also [CCD+20, Section 11]. We discuss how this technique applies to our specific problem in Section 3.8.

Within the data structure of Theorem 1.3.1 we modify the maintained tree decomposition $\mathcal{T}$ only in a restricted fashion: through *prefix-rebuilding updates*. These amount to rebuilding a prefix $\mathcal{S}$ of $\mathcal{T}$ into a new prefix $\mathcal{S}'$, and reattaching all trees of $\mathcal{T} - \mathcal{S}$ to $\mathcal{S}'$ without modifying them. As a consequence, while in Theorem 1.3.1 we only discuss maintenance of CMSO_2 properties, in fact we can maintain the run of any standard dynamic programming procedure on $\mathcal{T}$. In Section 3.7 we present a general automata-based framework for dynamic programming on tree decompositions that can be combined with our data structure. Automata verifying CMSO_2-expressible properties are just one instantiation of this framework.

We conjecture that the update time of the data structure of Theorem 1.3.1 can be improved to polylogarithmic in n, or even close to the $\mathcal{O}(\log n)$ bound achieved by Bodlaender for graphs of treewidth 2.

Organization of the chapter. In Section 3.1 we give an overview of our algorithm and present the key ideas behind the result. In Section 3.2 we present our framework of "prefix-rebuilding data structures" and

the statement of the main lemma (Lemma 3.2.5) that will imply Theorem 1.3.1. In Section 3.3 we prove combinatorial results on objects called "closures", which will then be leveraged in Section 3.4 to build our main algorithmic tool called the "refinement operation". Then, in Section 3.5 we use the refinement operation to build a height reduction operation for dynamic tree decompositions, and in Section 3.6 we put these results together and finish the proof of Lemma 3.2.5. We discuss conclusions and future research directions in Section 3.9. In Section 3.7 we present our framework for dynamic maintenance of dynamic programming on tree decompositions and in Section 3.8 we complete the proof of Theorems 1.3.1 and 1.3.3 from Lemma 3.2.5.

3.1 Overview

In this section we give an overview of our algorithm. We first give a high-level description of the whole algorithm in Section 3.1.1, and then in Sections 3.1.2 and 3.1.3 we sketch the proofs of the most important technical ingredients.

3.1.1 High-level description

Let n be the vertex count and k a given parameter that bounds the treewidth of the considered dynamic graph G. Our goal is to maintain a rooted tree decomposition $\mathcal{T}$ of height $2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$ and width at most $6k+5$, and at the same time any dynamic programming scheme, or more formally, a tree decomposition automaton with $\mathcal{O}_k(1)$ evaluation time, on $\mathcal{T}$. We will also require $\mathcal{T}$ to be binary, i.e., that every node has at most two children.

The goal of maintaining such a tree decomposition is reasonable because of a well-known lemma of Bodlaender and Hagerup [BH98]: For every graph of treewidth k, there exists a binary tree decomposition of height $\mathcal{O}(\log n)$ and width at most $3k+2$. Then, assuming $\mathcal{T}$ is a binary tree decomposition of height h and width $\mathcal{O}(k)$, the operations of adding an edge or deleting an edge can be implemented in time $\mathcal{O}_k(h)$ as follows. Let us assume that we store the existence of an edge uv in the highest node whose bag contains both u and v. Now, when deleting the edge uv, it suffices to find the highest node of $\mathcal{T}$ whose bag contains both u and v, update information about the existence of this edge stored in this node, and then update dynamic programming tables of the nodes on the path from this node to the root, taking $\mathcal{O}_k(h)$ time. In the edge addition operation between vertices u and v, we let P_u (resp. P_v) be the path in $\mathcal{T}$ from the highest node containing u (resp. v) to the root, add u and v to all bags on $P_u \cup P_v$, add the information about the existence of the edge uv to the root node, and update dynamic programming tables on $P_u \cup P_v$, again taking in total $\mathcal{O}_k(h)$ time. Let us emphasize that only the highest bag containing both u and v is "aware" of the existence of the edge uv, as opposed to the more intuitive alternative of all the bags containing both u and v being "aware" of uv. This is crucial for the fact that the dynamic programming tables of only $\mathcal{O}(h)$ nodes have to be recomputed after an edge addition/deletion.

Now, the only issue is that the edge addition operation could cause the width of $\mathcal{T}$ to increase to more than $6k+5$. By maintaining Bodlaender-Kloks dynamic programming [BK96] on $\mathcal{T}$, we can detect if the treewidth of the graph actually increased to more than k and terminate the algorithm in that case.[11] The more interesting case is when the treewidth of the graph is still at most k, in which case we have to modify $\mathcal{T}$ in order to make its width smaller while still maintaining small height. The main technical contribution of our result is to show that such changes to tree decompositions can indeed be implemented efficiently.

Let us introduce some notation. We denote a rooted tree decomposition by a pair $\mathcal{T} = (T, \mathsf{bag})$, where T is a rooted tree and $\mathsf{bag}\colon V(T) \to 2^{V(G)}$ is a function specifying the bag $\mathsf{bag}(t)$ of each node t. For a set of nodes $W \subseteq V(T)$, we denote by $\mathsf{bags}(W) = \bigcup_{t\in W} \mathsf{bag}(t)$. A *prefix* of a rooted tree T is a a set of nodes $T_{\mathrm{pref}} \subseteq V(T)$ that contains the root and induces a connected subtree, and a prefix of a rooted tree decomposition $\mathcal{T} = (T, \mathsf{bag})$ is a prefix of T. For a rooted tree T, we denote by $\mathsf{height}(T)$ the maximum number of nodes on a root-leaf path, and for a node $t \in V(T)$, $\mathsf{height}(t)$ is the height of the subtree rooted at t.

Recall that in the edge addition operation, we increased the sizes of bags in a subtree consisting of the union $P_u \cup P_v$ of two paths, each between a node and the root. In particular, all of the nodes with too large bags are contained in the prefix $P_u \cup P_v$ of size at most $2h$, where h is the height of our tree decomposition. Now, a natural idea for improving the width would be to replace the prefix $P_u \cup P_v$ by a tree decomposition of $G[\mathsf{bags}(P_u \cup P_v)]$ with height $\mathcal{O}(\log n)$ and width $3k+2$ given by the Bodlaender-Hagerup lemma.

[11] By the standard technique of delaying updates we actually do not need to terminate the algorithm, but in this overview let us assume for simplicity that we are allowed to just terminate the algorithm if the width becomes more than k.

While this form of the idea is too naive, we show that surprisingly, something that is similar in the spirit can be achieved.

The main tool we develop for maintaining tree decompositions in the dynamic setting is the *refinement operation*. The definition and properties of the operation are technical and will be described in Section 3.1.2, but let us give here an informal description of what is achieved by the operation. The refinement operation takes as an input a prefix T_{pref} of the tree decomposition $\mathcal{T}$ that we are maintaining, and informally stated, replaces T_{pref} by a tree decomposition of width at most $6k+5$ and height at most $\mathcal{O}(\log n)$. The operation also edits other parts of $\mathcal{T}$, but in a way that makes them only better in terms of sizes of bags. In particular, if we use the refinement operation on $T_{\mathrm{pref}} = P_u \cup P_v$ after an edge addition operation that made the width exceed $6k+5$ in the nodes in $P_u \cup P_v$, the operation brings the width of $\mathcal{T}$ back to at most $6k+5$. The amortized time complexity of the refinement operation is $\mathcal{O}_k(|T_{\mathrm{pref}}|)$, and it can increase the height of $\mathcal{T}$ by at most $\mathcal{O}(\log n)$.

With the refinement operation, we have a tool for keeping the width of the maintained tree decomposition $\mathcal{T}$ bounded by $6k+5$. However, each application of the refinement operation can increase the height of $\mathcal{T}$ by $\log n$, so we need a tool also for decreasing the height. We develop such a tool by a combination of a carefully chosen potential function and a strategy to decrease the potential function "for free" by using the refinement operation if the height is too large. In particular, the potential function we use is

$$\Phi(\mathcal{T}) = \sum_{t \in V(T)} (\gamma \cdot k)^{|\mathsf{bag}(t)|} \cdot \mathsf{height}(t),$$

where $\gamma < 1000$ is a fixed constant defined in Section 3.4.1. This function has the properties that it does not increase too much in the edge addition operation (the increase is at most $\mathcal{O}_k(\mathsf{height}(T)^2)$), it plays well together with the details of the amortized analysis of the refinement operation (the factor $(\gamma \cdot k)^{|\mathsf{bag}(t)|}$ comes from there), and because of the factor $\mathsf{height}(t)$, it naturally admits smaller values on trees of smaller height. In Section 3.1.3 we outline a strategy that, provided the height of $\mathcal{T}$ exceeds $2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$, selects a prefix T_{pref} so that applying the refinement operation to T_{pref} decreases the value of $\Phi(\mathcal{T})$, and moreover the running time of the refinement operation can be bounded by this decrease. In particular, this means that as long as the height is more than $2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$, we can apply such a refinement operation "for free", in terms of amortized running time, and moreover decrease the value of the potential. As the potential cannot keep decreasing forever, repeated applications of such an operation eventually lead to improving the height to at most $2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$.

3.1.2 The refinement operation

In this subsection we overview the refinement operation. First, let us note that our refinement operation builds on the tree decomposition improvement operation introduced by Korhonen and Lokshtanov for improving static fixed-parameter algorithms for treewidth [KL22], which in turn builds on the 2-approximation algorithm for treewidth of Korhonen [Kor21] and in particular on the use of the techniques of Thomas [Tho90] and Bellenbaum and Diestel [BD02] in computing treewidth. Our refinement operation generalizes the improvement operation of [KL22] by allowing to refine a continuous subtree instead of only a single bag, which is crucial for controlling the height of the tree decomposition. We also adapt the operation from being suitable to compute treewidth exactly into a more approximative version (the $6k+5$ width comes from the combination of this and the Bodlaender-Hagerup lemma [BH98]) in order to attain structural properties that are needed for efficient running time. For this, the Dealternation Lemma of Bojańczyk and Pilipczuk [BP22] is used. In the rest of this section we do not assume knowledge of these previous results.

Recall the goal of the refinement operation: Given a prefix T_{pref} of the tree decomposition $\mathcal{T} = (T, \mathsf{bag})$ that we are maintaining, we would like to, in some sense, recompute the tree decomposition on this prefix. Observe that we cannot simply select an induced subgraph like $G[\mathsf{bags}(T_{\mathrm{pref}})]$ and hope to replace T_{pref} by any tree decomposition of it, because the tree decomposition needs to take into account also the connectivity provided by vertices outside of $\mathsf{bags}(T_{\mathrm{pref}})$. For this, the correct notion will be the *torso* of a set of vertices. Recall from Preliminaries that for a graph G and a set X of vertices, the graph $\mathsf{torso}_G(X)$ has the vertex set X and has an edge uv if there is a u-v-path in G whose internal vertices are outside of X. In other words, $\mathsf{torso}_G(X)$ is the supergraph of $G[X]$ obtained by making for each connected component C of $G - X$ the neighborhood $N(C)$ into a clique.

We then outline the refinement operation (see Figure 3.1 for an illustration of it). Given T_{pref}, we find a set of vertices $X \supseteq \mathsf{bags}(T_{\mathrm{pref}})$ so that $\mathsf{torso}_G(X)$ has treewidth at most $2k+1$ (here we have $2k+1$ instead

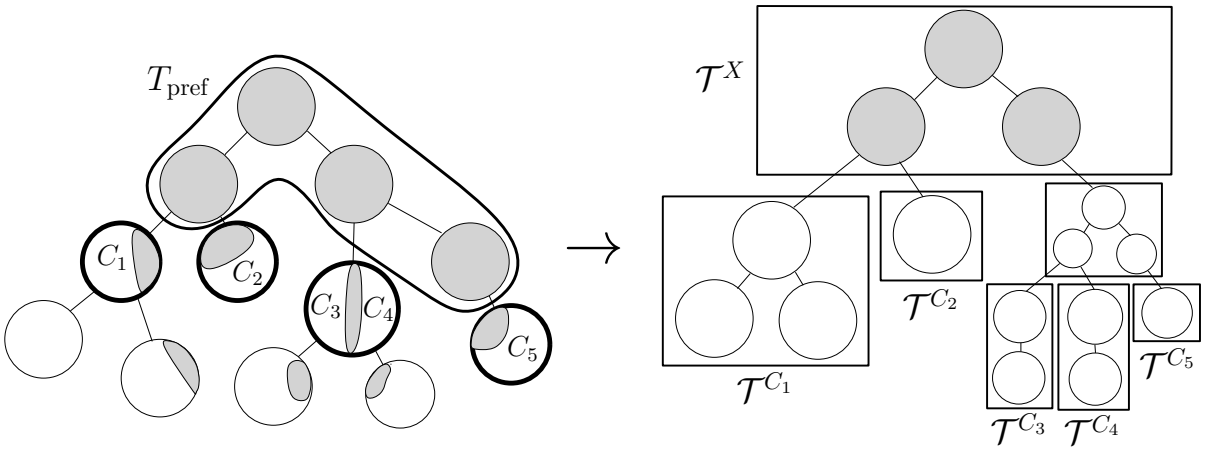


Figure 3.1: The refinement operation. The left picture illustrates the tree decomposition $\mathcal{T}$, with the prefix T_{pref} encircled and the vertices in $X \supseteq \mathsf{bags}(T_{\text{pref}})$ depicted in gray. The appendices of T_{pref} are circled by boldface, and the components of $G - X$ are denoted by $C_1, \ldots, C_5$. The right picture illustrates the tree decomposition constructed from $\mathcal{T}$ by the refinement using X, in particular, by taking the tree decomposition $\mathcal{T}^X$ of $\mathsf{torso}_G(X)$, and gluing the tree decompositions $\mathcal{T}^{C_i}$ for components C_i of $G - X$ to it. The subtree consisting of the three nodes above $\mathcal{T}^{C_3}$, $\mathcal{T}^{C_4}$, $\mathcal{T}^{C_5}$ is constructed in order to keep the tree binary after reattaching $\mathcal{T}^{C_3}$, $\mathcal{T}^{C_4}$, $\mathcal{T}^{C_5}$.

of k for a technical reason we will explain). Then, we compute an optimum-width tree decomposition $\mathcal{T}^X$ of $\mathsf{torso}_G(X)$ and use the Bodlaender-Hagerup lemma [BH98] to make its height $\mathcal{O}(\log n)$, resulting in $\mathcal{T}^X$ having width at most $6k + 5$. We root $\mathcal{T}^X$ at an arbitrary node, and it will form a prefix of the new refined tree decomposition. What remains, is to construct tree decompositions $\mathcal{T}^C$ for each connected component C of $G - X$ and attach them into $\mathcal{T}^X$.

Next, for each appendix a of T_{pref}, denote by $\mathcal{T}_a = (T_a, \mathsf{bag}_a)$ the restriction of $\mathcal{T}$ to the subtree rooted at a. Now, note that because $X \supseteq \mathsf{bags}(T_{\text{pref}})$, for each connected component C of $G - X$ there exists a unique appendix a of T_{pref} such that C is contained in the bags of $\mathcal{T}_a$. Moreover, the restriction $(T_a, \mathsf{bag}_a|_{N[C]})$ to the closed neighborhood $N[C]$ of C is a tree decomposition of the induced subgraph $G[N[C]]$ minus the edges inside $N(C)$. Then, observe that because $\mathcal{T}^X$ is a tree decomposition of $\mathsf{torso}_G(X)$, it must have a bag that contains $N(C)$. Now, our goal is to attach $(T_a, \mathsf{bag}_a|_{N[C]})$ into this bag. In order to achieve this while satisfying the connectedness condition of tree decompositions, we need to have the set $N(C)$ in the root of $(T_a, \mathsf{bag}_a|_{N[C]})$. We denote by $\mathcal{T}^C = (T^C, \mathsf{bag}^C)$ the tree decomposition obtained from $(T_a, \mathsf{bag}_a|_{N[C]})$ by "forcing" $N(C)$ to be in the bag $\mathsf{bag}^C(a)$ of the root node a, in particular, by inserting $N(C)$ to $\mathsf{bag}^C(a)$ and then fixing the connectedness condition by inserting each vertex $v \in N(C)$ to all bags on the unique path from the root to the subtree of the other bags containing v. Then, $\mathcal{T}^C$ is a tree decomposition of $G[N[C]]$ whose root bag contains $N(C)$, and therefore it can be attached to the bag of $\mathcal{T}^X$ that contains $N(C)$. These attachments may make the degree of the resulting tree decomposition higher than 2, so finally these high-degree nodes need to be expanded into binary trees. This concludes the informal description of the refinement operation. The actual definition is a bit more involved, as it is necessary for obtaining efficient running time to (1) treat in some cases multiple different components C in $\mathcal{T}_a$ as one component, and (2) prune out some unnecessary bags of $\mathcal{T}^C$.

From the description of the refinement operation sketched above, it should be clear that the resulting tree decomposition is indeed a tree decomposition of G. It is also easy to see that the height of the refined tree decomposition is at most $\mathsf{height}(\mathcal{T}) + \mathcal{O}(\log n)$: This is because $\mathcal{T}^X$ has height at most $\mathcal{O}(\log n)$, and each of the attached decompositions $\mathcal{T}^C$ has height at most $\mathsf{height}(\mathcal{T})$. Recall that our goal is that if all of the bags of width more than $6k + 5$ of $\mathcal{T}$ are contained in T_{pref}, then the width of refined tree decomposition is at most $6k + 5$. The widths of the bags in $\mathcal{T}^X$ are clearly at most $6k + 5$. However, because of the additional insertions of vertices in $N(C)$ to bags in $\mathcal{T}^C$, it is not clear why those bags would have width at most $6k + 5$. In fact, we cannot guarantee this without additional properties of X we outline next.

Let us call a set of vertices $X \subseteq V(G)$ a k-closure of T_{pref} if $X \supseteq \mathsf{bags}(T_{\text{pref}})$ and the treewidth of $\mathsf{torso}_G(X)$ is at most $2k+1$. In particular, the set X in the refinement operation is a k-closure of T_{pref}. We say that a k-closure X is *linked* into T_{pref} if for each component C of $G - X$, the set $N(C)$ is linked into

$\mathsf{bags}(T_{\mathrm{pref}})$ in the sense that there are no separators of size $< |N(C)|$ separating $N(C)$ from $\mathsf{bags}(T_{\mathrm{pref}})$. The key property for controlling the width of $\mathcal{T}^C$ is that if X is linked into T_{pref}, then each bag of $\mathcal{T}^C$ has width at most the width of the corresponding bag in $\mathcal{T}$. In particular, let $\mathcal{T}_a$ be a subtree of $\mathcal{T}$ hanging on an appendix a of T_{pref}, and C be a component of $G - X$ contained in $\mathcal{T}_a$. Recall that we construct the tree decomposition $\mathcal{T}^C = (T^C, \mathsf{bag}^C)$ by taking $T^C = T_a$, and then for each $t \in V(T^C)$ setting

$$\mathsf{bag}^C(t) = (\mathsf{bag}_a(t) \cap N[C]) \cup \{v \in N(C) \mid \text{the highest bag containing } v \text{ is below } t\}.$$

Then, we can bound the size of $\mathsf{bag}^C(t)$ as follows.

Lemma 3.1.1. *If X is linked into T_{pref}, then $|\mathsf{bag}^C(t)| \leqslant |\mathsf{bag}_a(t)|$.*

Proof sketch. Let $N_t = \{v \in N(C) \mid \text{the highest bag containing } v \text{ is below } t\}$ and note that it suffices to prove $|N_t| \leqslant |\mathsf{bag}_a(t) \setminus N[C]|$. By linkedness and Menger's theorem, there are $N(C)$ vertex-disjoint paths from $N(C)$ to $\mathsf{bags}(T_{\mathrm{pref}})$, and because $\mathsf{bags}(T_{\mathrm{pref}})$ is disjoint from C, all internal vertices of these paths are in $V(G) \setminus N[C]$. Moreover, $|N_t|$ of these paths are from N_t to $\mathsf{bags}(T_{\mathrm{pref}})$, and because $\mathsf{bag}_a(t)$ separates N_t from $\mathsf{bags}(T_{\mathrm{pref}})$ and is disjoint from N_t, each such path contains an internal vertex in $\mathsf{bag}_a(t)$, implying $|\mathsf{bag}_a(t) \setminus N[C]| \geqslant |N_t|$. □

Lemma 3.1.1 shows that if T_{pref} contains all bags of width more than $6k+5$ and X is linked into T_{pref}, the resulting tree decomposition will have width at most $6k+5$. We note that the existence of a k-closure of T_{pref} that is linked into T_{pref} is nontrivial, but before going into that let us immediately generalize the notion of linkedness in order to obtain a stronger form of Lemma 3.1.1 that will be useful for analyzing the potential function. Recall that *depth* $d_{\mathcal{T}}(v)$ of a vertex $v \in V(G)$ in $\mathcal{T} = (T, \mathsf{bag})$ is the depth of the highest node of T whose bag contains v (i.e., the distance from this node to root). For a set of vertices S, we denote $d_{\mathcal{T}}(S) = \sum_{v \in S} d_{\mathcal{T}}(v)$. Then, we say that a k-closure X is *$d_{\mathcal{T}}$-linked* into T_{pref} if it is linked into T_{pref}, and additionally for each neighborhood $N(C)$, there are no separators S with $|S| = |N(C)|$ and $d_{\mathcal{T}}(S) < d_{\mathcal{T}}(N(C))$ separating $N(C)$ from $\mathsf{bags}(T_{\mathrm{pref}})$.

Recall our potential $\Phi(\mathcal{T}) = \sum_{t \in V(T)} (\gamma \cdot k)^{|\mathsf{bag}(t)|} \cdot \mathsf{height}(t)$. Using $d_{\mathcal{T}}$-linkedness we are able to prove that the actual definition of the refinement operation satisfies the following properties.

Lemma 3.1.2 (Informal)**.** *Let X be a $d_{\mathcal{T}}$-linked k-closure of T_{pref}, a an appendix of T_{pref}, and $C_1, \ldots, C_\ell$ the connected components of $G - X$ that are contained in $\mathcal{T}_a$. It holds that $\sum_{i=1}^{\ell} \Phi(\mathcal{T}^{C_i}) \leqslant \Phi(\mathcal{T}_a)$, and moreover, the tree decompositions $\mathcal{T}^{C_i}$ for all i, together with their updated dynamic programming tables, can be constructed in time $\mathcal{O}_k(\Phi(\mathcal{T}_a) - \sum_{i=1}^{\ell} \Phi(\mathcal{T}^{C_i}))$.*

Let us note that Lemma 3.1.2 would hold also for a potential function without the $\mathsf{height}(t)$ factor; this factor is included in the potential only for the purposes of the height reduction scheme that will be outlined in Section 3.1.3. Also, in the actual refinement operation each C_i in Lemma 3.1.2 can actually be the union of multiple different components with the same neighborhood $N(C_i)$, and we can actually charge a bit extra from the potential for each of these "connected components"; this extra potential will be used for constructing the binary trees for the high-degree attachment points.

After ignoring these numerous technical details, the main takeaway of Lemma 3.1.2 is that constructing the decompositions $\mathcal{T}^C$ is "free" in terms of the potential. The only place where we could use a lot of time or increase the potential a lot is finding the set X and constructing the tree decomposition $\mathcal{T}^X$. For bounding this, we give a lemma asserting that we can assume X to have size at most $\mathcal{O}_k(|T_{\mathrm{pref}}|)$, and in addition to have an even stronger structural property that will be useful in the height reduction scheme. For a node a of $\mathcal{T}$, denote by $\mathsf{cmp}(a) \subseteq V(G)$ the vertices that occur in the bags of the subtree rooted at a, but not in bag of the parent of a. We prove the following statement using the Dealternation Lemma of Bojańczyk and Pilipczuk [BP22]. We note that the bound $2k+1$ in the definition of k-closure comes from this proof.

Lemma 3.1.3. *Let G be a graph of treewidth at most k, and $\mathcal{T}$ a tree decomposition of G of width $\mathcal{O}(k)$. For any prefix T_{pref} of $\mathcal{T}$, there exists a k-closure X of T_{pref} so that for each appendix a of T_{pref} it holds that $|X \cap \mathsf{cmp}(a)| \leqslant \mathcal{O}(k^4)$.*

In particular, as $\mathcal{T}$ is a binary tree, T_{pref} has at most $|T_{\mathrm{pref}}| + 1$ appendices. So by Lemma 3.1.3, T_{pref} admits a k-closure with at most $|T_{\mathrm{pref}}| \cdot (k+1) + \mathcal{O}(|T_{\mathrm{pref}}| \cdot k^4) = \mathcal{O}(|T_{\mathrm{pref}}| \cdot k^4)$ vertices. By using such a k-closure, we can bound the size of $\mathcal{T}^X$ by $\mathcal{O}_k(|T_{\mathrm{pref}}|)$. As each node in $\mathcal{T}^X$ has potential at most $\mathcal{O}_k(\mathsf{height}(\mathcal{T}) + \log n)$ in the resulting decomposition, we get that the refinement operation increases the potential function by at most $\mathcal{O}_k(|T_{\mathrm{pref}}| \cdot (\mathsf{height}(\mathcal{T}) + \log n))$. In particular, in the refinement operation

applied directly after edge insertion, we have that $|T_{\text{pref}}| \leqslant 2|\mathsf{height}(\mathcal{T})|$, so the potential function increases by at most $\mathcal{O}_k(\mathsf{height}(\mathcal{T})^2)$ (note that $\mathsf{height}(\mathcal{T}) \geqslant \log n$).

Let us now turn to two issues that we have delayed for some time: How to guarantee that the k-closure X is $d_{\mathcal{T}}$-linked, and how to actually find such an X. Let us say that a closure is *c-small* if it satisfies the condition of Lemma 3.1.3 for some specific bound $c \in \mathcal{O}(k^4)$ that can be obtained from the proof of Lemma 3.1.3. The following lemma, which is proved similarly to proofs of Korhonen and Lokshtanov [KL22, Section 5], gives a simple condition that guarantees $d_{\mathcal{T}}$-linkedness.

Lemma 3.1.4. *Let T_{pref} be a prefix of a tree decomposition. If X is a c-small k-closure of T_{pref} that among all c-small k-closures of T_{pref} primarily minimizes $|X|$, and secondarily minimizes $d(X)$, then X is $d_{\mathcal{T}}$-linked into T_{pref}.*

With Lemma 3.1.4, we can use dynamic programming for finding c-small k-closures that are $d_{\mathcal{T}}$-linked. In particular, we adapt the dynamic programming of Bodlaender and Kloks [BK96] for computing treewidth into computing c-small k-closures that optimize for the conditions in the lemma. This adaptation uses quite standard techniques, but let us note that one complication is that even a small change to a tree decomposition can change the depths of all vertices, so we cannot just store the value $d(X)$ in the dynamic programming tables. Instead, we have to make use of the definition of the function d and the fact that we are primarily minimizing $|X|$. By maintaining these dynamic programming tables throughout the algorithm, we get that given T_{pref}, we can in time $\mathcal{O}_k(|T_{\text{pref}}|)$ find an $\mathcal{O}(k^4)$-small k-closure X of T_{pref} that is $d_{\mathcal{T}}$-linked, and also the graph $\mathsf{torso}_G(X)$.

3.1.3 Height reduction

In this subsection we sketch the height reduction scheme, in particular, the following lemma.

Lemma 3.1.5 (Height reduction)**.** *Let $\mathcal{T}$ be the tree decomposition we are maintaining. There is a function $f(n,k) \in 2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$ so that if $\mathsf{height}(\mathcal{T}) > f(n,k)$, then there exists a prefix T_{pref} of $\mathcal{T}$ so that the refinement operation on T_{pref} results in a tree decomposition $\mathcal{T}'$ with $\Phi(\mathcal{T}') < \Phi(\mathcal{T})$ and runs in time $\mathcal{O}_k(\Phi(\mathcal{T}) - \Phi(\mathcal{T}'))$.*

To sketch the proof of Lemma 3.1.5, let us build a certain model of accounting how the potential changes in a refinement operation with a prefix T_{pref}. First, recall that Lemma 3.1.2 takes care of the potential in the tree decompositions $\mathcal{T}^C$ for components C of $G - X$, and the only place we need to worry about increasing the potential are the nodes of $\mathcal{T}^X$. In the previous section we bounded this potential increase by $\mathcal{O}_k(|T_{\text{pref}}| \cdot (\mathsf{height}(\mathcal{T}) + \log n))$, where in particular the factor $\mathsf{height}(\mathcal{T}) + \log n$ comes from the fact that after attaching the tree decomposition $\mathcal{T}^C$, the height of a node in $\mathcal{T}^X$ could be as large as $\mathsf{height}(\mathcal{T}) + \log n$. This upper bound was sufficient for the refinement operation performed after an edge addition, but to prove Lemma 3.1.5 we need a more fine-grained view.

Consider the following model: We start with the tree decomposition of $\mathcal{T}^X$ of height $\mathcal{O}(\log n)$, and attach the tree decompositions of the components $\mathcal{T}^C$ to it one by one. Each time we attach a tree decomposition $\mathcal{T}^C$, we increase the height of at most $\mathcal{O}(\log n)$ nodes in $\mathcal{T}^X$ (because $\mathsf{height}(\mathcal{T}^X) \leqslant \mathcal{O}(\log n)$), and this height is increased by at most $\mathsf{height}(\mathcal{T}^C)$, which is at most $\mathsf{height}(a)$, where a is the appendix of T_{pref} whose subtree contains C. While there can be many components C contained in $\mathcal{T}_a$, observe that Lemma 3.1.3 implies that in fact such components can have only at most $k^{\mathcal{O}(k)}$ different neighborhoods $N(C)$, and therefore at most $k^{\mathcal{O}(k)}$ different attachment points in $\mathcal{T}^X$. In particular, after handling technical details about the binary trees used to flatten high-degree attachment points, we can assume that an appendix a of T_{pref} is responsible for increasing the height of at most $\mathcal{O}_k(\log n)$ nodes in $\mathcal{T}^X$. Moreover, it increases the height of those nodes by at most $\mathsf{height}(a)$ each, so in total, it is responsible for increasing the potential by $\mathcal{O}_k(\mathsf{height}(a) \cdot \log n)$.

Denote $\Phi(T_{\text{pref}}) = \sum_{t \in T_{\text{pref}}} (\gamma \cdot k)^{|\mathsf{bag}(t)|} \cdot \mathsf{height}(t)$, i.e., the value of the potential on the nodes in T_{pref}. The above discussion, combined with Lemma 3.1.2, leads to the following lemma.

Lemma 3.1.6. *Let T_{pref} be a prefix of a tree decomposition $\mathcal{T}$, $A := \mathsf{App}(T_{\text{pref}}) \subseteq V(T)$ the appendices of T_{pref}, and $\mathcal{T}'$ the tree decomposition resulting from refining $\mathcal{T}$ with T_{pref}. It holds that*

$$\Phi(\mathcal{T}') \leqslant \Phi(\mathcal{T}) - \Phi(T_{\text{pref}}) + \mathcal{O}_k(|T_{\text{pref}}| \cdot \log n) + \sum_{a \in A} \mathcal{O}_k(\mathsf{height}(a) \cdot \log n).$$

Now, in order to prove Lemma 3.1.5, it is sufficient to prove that if $\mathcal{T}$ has too large height, then there exists a prefix T_{pref} with appendices A so that $\Phi(T_{\text{pref}}) > c_k \log n \left(|T_{\text{pref}}| + \sum_{a \in A} \mathsf{height}(a)\right)$, where c_k is

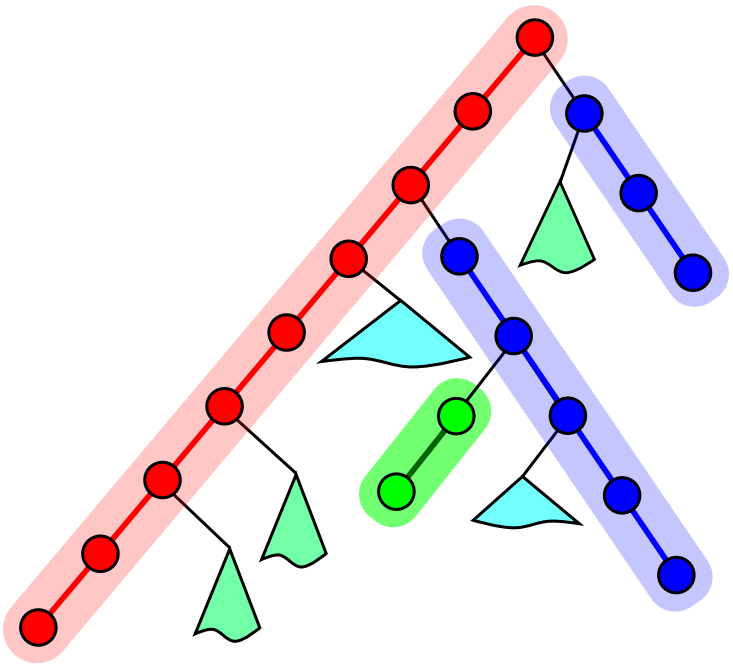


Figure 3.2: Construction of T_{pref} in height reduction. The consecutive paths extracted by the construction procedure are depicted in red, blue, and green. Their union constitutes T_{pref}. Big trees are depicted in sea-green, shallow trees are depicted in cyan.

some large enough number depending on k. (Here, the required c_k comes from the number of attachment points and the potential function, so some $c_k = k^{\mathcal{O}(k)}$ is sufficient.) In our proof, the value $\Phi(T_{\text{pref}})$ in fact will be larger than $c_k \log n \left(|T_{\text{pref}}| + \sum_{a \in A} \mathsf{height}(a)\right)$ by some arbitrary constant factor, which gives also the required property that the refinement operation on such a prefix T_{pref} will run in time $\mathcal{O}_k(\Phi(\mathcal{T}) - \Phi(\mathcal{T}'))$. We will find the prefix T_{pref} using the following lemma:

Lemma 3.1.7. *Let $c \geqslant 2$ and T be a binary tree with n nodes. If the height of T is at least $2^{\Omega(\sqrt{\log n \log c})}$, then there exists a nonempty prefix T_{pref} of T so that*

$$c \cdot \left(|T_{\text{pref}}| + \sum_{a \in \mathsf{App}(T_{\text{pref}})} \mathsf{height}_T(a) \right) \leqslant \sum_{x \in T_{\text{pref}}} \mathsf{height}_T(x).$$

Moreover, if we can access the height of each node of T in constant time, then such T_{pref} can be computed in time $\mathcal{O}(|T_{\text{pref}}|)$.

In our algorithm, it will be enough to invoke Lemma 3.1.7 with $c = \Theta(c_k \log n)$ with a large enough constant hidden in the Θ notation, and then the prefix T_{pref} received from applying Lemma 3.1.7 will satisfy the requirements of Lemma 3.1.5. It remains to sketch the proof of Lemma 3.1.7. The remaining part of this section is dedicated to this sketch.

We first sketch how to select T_{pref} when $\mathsf{height}(T) > n^\varepsilon$ for some $\varepsilon > 0$ and c is small compared to n. The natural strategy is to start by setting T_{pref} to be the path from the root to the deepest leaf in T. Then we have $\sum_{x \in T_{\text{pref}}} \mathsf{height}_T(x) \geqslant n^{2\varepsilon}/2$. However, T_{pref} may have $n^\varepsilon/2$ appendices that each have height $n^\varepsilon/2$, so it is possible that $c \cdot \sum_{a \in \mathsf{App}(T_{\text{pref}})} \geqslant n^{2\varepsilon} \cdot c/4$. The key observation is that in this case, many subtrees of appendices $a \in A$ must be even more unbalanced than T is, having height at least $n^\varepsilon/2$ while containing at most $4 \cdot n^{1-\varepsilon}$ nodes (simply by a counting argument). In particular, let us say that a subtree rooted on an appendix a is *big* if it contains more than $10c \cdot n^{1-\varepsilon}$ nodes, and *shallow* if $\mathsf{height}_T(a) \leqslant n^\varepsilon/(10c)$. Now, there can be at most $n/(10c \cdot n^{1-\varepsilon}) = n^\varepsilon/(10c)$ big subtrees, so they contribute at most $n^{2\varepsilon}/10$ to the sum $\sum_{a \in \mathsf{App}(T_{\text{pref}})} c \cdot \mathsf{height}_T(a)$. Similarly, the shallow subtrees also contribute at most $n^{2\varepsilon}/10$ to the sum, so the sum coming from the roots of the subtrees of T_{pref} that are either big or shallow (or both) is only a small fraction of $\sum_{x \in T_{\text{pref}}} \mathsf{height}_T(x)$.

Then, there are subtrees of appendices that are neither big or shallow; these subtrees are both small and deep, hence they seem even more unbalanced than $\mathcal{T}$. We apply the same strategy to those trees recursively.

For each appendix a whose subtree is small and deep, we insert to T_{pref} the path P_a from a to its deepest descendant. As the subtree is deep (of height at least $n^{\varepsilon}/(10c)$), we have that $\sum_{x \in T_{\mathrm{pref}}} \mathsf{height}_T(x)$ increases by $\sum_{x \in P_a} \mathsf{height}_T(x) = \Omega(n^{2\varepsilon}/(10c)^2)$. Now, when analyzing the appendices of P_a, we again apply the strategy to handle subtrees that are big or shallow by charging them from $\sum_{x \in P_a} \mathsf{height}_T(x)$, and then handling subtrees that are both small and deep recursively. This time, the right definition of big will be to have at least $n^{1-2\varepsilon}(10c)^3$ nodes, and the right definition of shallow will be to have height at most $n^{\varepsilon}/(10c)^2$. More generally, on the ith level of such recursion we can call a subtree big if it contains more than $n^{1-i\varepsilon}(10c)^{i\cdot(i+1)/2}$ nodes, and shallow if its height is at most $n^{\varepsilon}/(10c)^i$. When ε is a constant, this recursion can continue only for a constant number of levels before no subtree can be both small and deep, simply because it would require the subtree to have larger height than the number of nodes. Therefore, in the end we are able to find a prefix T_{pref} that satisfies the requirements of Lemma 3.1.7.

It is not surprising that selecting the height limit to be n^{ε} is not optimal. In particular, the same strategy as outlined above will work if we select the initial height to be of the form $2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$, resulting in showing that we can obtain the amortized running time of $2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$ per query.

3.2 Dynamic tree decompositions

In this section we present a general design of a data structure that will operate on a dynamically changing binary tree decomposition of a dynamically changing graph.

Define an *annotated tree decomposition* of a graph G as a triple $(T, \mathsf{bag}, \mathsf{edges})$ where:

- T and bag are defined as in the standard definition of the tree decomposition;
- $\mathsf{edges}\colon V(T) \to 2^{\binom{V(G)}{2}}$ is defined as follows: For a node $t \in V(T)$, $\mathsf{edges}(t)$ is a subset of $\binom{\mathsf{bag}(t)}{2}$ consisting of all edges $uv \in E(G) \cap \binom{\mathsf{bag}(t)}{2}$ for which t is the shallowest node containing both u and v. Note that thus, every edge of G belongs to exactly one set $\mathsf{edges}(t)$.

Note that edges is uniquely determined from G and (T, bag); and conversely, G is uniquely determined from $(T, \mathsf{bag}, \mathsf{edges})$. Given a set $A \subseteq V(T)$, the restriction of $(T, \mathsf{bag}, \mathsf{edges})$ to A, denoted $(T, \mathsf{bag}, \mathsf{edges})|_A$, is the tuple $(T[A], \mathsf{bag}|_A, \mathsf{edges}|_A)$ where $\mathsf{bag}|_A$ and $\mathsf{edges}|_A$ are restrictions of the functions bag, edges to A, respectively.

Next, consider an update changing an annotated binary tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ to another annotated binary tree decomposition $(T', \mathsf{bag}', \mathsf{edges}')$. This update can also change the underlying graph G, in particular, it changes G to be the graph uniquely determined from $(T', \mathsf{bag}', \mathsf{edges}')$. We say that the update is *prefix-rebuilding* if $(T', \mathsf{bag}', \mathsf{edges}')$ is created from $(T, \mathsf{bag}, \mathsf{edges})$ by replacing a prefix T_{pref} of T with a new rooted tree T'_{pref} and then "reattaching" some subtrees of T rooted at the appendices of T_{pref} below the nodes of T'_{pref}. Formally, a prefix-rebuilding update is described by a tuple $\overline{u} \coloneqq (T_{\mathrm{pref}}, T'_{\mathrm{pref}}, T^\star, \mathsf{bag}^\star, \mathsf{edges}^\star, \pi)$ where:

- $T_{\mathrm{pref}} \subseteq V(T)$ is a prefix of T;
- $T'_{\mathrm{pref}} \subseteq V(T')$ is a prefix of T' satisfying
$$(T, \mathsf{bag}, \mathsf{edges})|_{V(T)\setminus T_{\mathrm{pref}}} = (T', \mathsf{bag}', \mathsf{edges}')|_{V(T')\setminus T'_{\mathrm{pref}}};$$
- $(T^\star, \mathsf{bag}^\star, \mathsf{edges}^\star) = (T', \mathsf{bag}', \mathsf{edges}')|_{T'_{\mathrm{pref}}}$;
- $\pi\colon \mathsf{App}(T_{\mathrm{pref}}) \rightharpoonup T'_{\mathrm{pref}}$ is the partial function that maps appendices of T_{pref} to nodes of T'_{pref} such that for each appendix t of T_{pref} for which $\pi(t)$ is defined, the parent of t in T' is $\pi(t)$.

It is straightforward that $(T', \mathsf{bag}', \mathsf{edges}')$ can be uniquely determined from $(T, \mathsf{bag}, \mathsf{edges})$ and the tuple $\overline{u}$ as above. The *size* of $\overline{u}$, denoted $|\overline{u}|$, is defined as $|T_{\mathrm{pref}}| + |T'_{\mathrm{pref}}|$. It is also straightforward that given $\overline{u}$, a representation of $(T, \mathsf{bag}, \mathsf{edges})$ can be turned into a representation of $(T', \mathsf{bag}', \mathsf{edges}')$ in time $\ell^{\mathcal{O}(1)} \cdot |\overline{u}|$, where ℓ is the maximum of the widths of $(T, \mathsf{bag}, \mathsf{edges})$ and $(T', \mathsf{bag}', \mathsf{edges}')$.

Finally, we say that a dynamic data structure is *ℓ-prefix-rebuilding with overhead τ* if it stores an annotated binary tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of width at most ℓ and supports the following operations:

- $\mathsf{Init}(T, \mathsf{bag}, \mathsf{edges})$: Initializes the annotated binary tree decomposition with $(T, \mathsf{bag}, \mathsf{edges})$. Runs in worst-case time $\mathcal{O}(\tau \cdot |V(T)| \cdot \ell^{\mathcal{O}(1)})$;

- $\mathsf{Update}(\overline{u})$: Applies a prefix-rebuilding update $\overline{u}$ to the decomposition $(T, \mathsf{bag}, \mathsf{edges})$. It can be assumed that the resulting tree decomposition is binary and has width at most ℓ. Runs in worst-case time $\mathcal{O}(\tau \cdot |\overline{u}| \cdot \ell^{\mathcal{O}(1)})$.

Usually, the overhead τ will correspond to the time necessary to recompute any auxiliary information associated with each node of the decomposition undergoing the update. For example, the height of a node s in the tree decomposition can be inferred in $\mathcal{O}(1)$ time from the heights of the (at most two) children of s, so the overhead required to recompute the heights of the nodes after the update is $\tau = \mathcal{O}(1)$ per affected node.

Prefix-rebuilding data structures will usually implement an additional operation allowing to efficiently query the current state of the data structure. For example, next we state a data structure that allows us to access various auxiliary information about the tree decomposition:

Lemma 3.2.1. *For every $\ell \in \mathbb{N}$, there exists an ℓ-prefix-rebuilding data structure with overhead $\mathcal{O}(1)$ that additionally implements the following operations:*

- $\mathsf{Height}(s)$*: Given a node $s \in V(T)$, returns $\mathsf{height}_T(s)$. Runs in worst-case time $\mathcal{O}(1)$.*
- $\mathsf{Size}(s)$*: Given a node $s \in V(T)$, returns the number of nodes in the subtree of T rooted at s. Runs in worst-case time $\mathcal{O}(1)$.*
- $\mathsf{Cmpsize}(s)$*: Given a node $s \in V(T)$, returns the size $|\mathsf{cmp}(s)|$. Runs in worst-case time $\mathcal{O}(1)$.*
- $\mathsf{Top}(v)$*: Given a vertex $v \in V(G)$, returns the unique highest node t of T so that $v \in \mathsf{bag}(t)$. Runs in worst-case time $\mathcal{O}(1)$.*

The proof of Lemma 3.2.1 uses standard arguments on dynamic programming on tree decompositions. It will be proved in Section 3.7.

More generally, any typical dynamic programming scheme on tree decompositions can be turned into a prefix-rebuilding data structure. Here is a statement that we present informally at the moment.

Lemma 3.2.2 (informal)**.** *Fix $\ell \in \mathbb{N}$. Assume that there exists a dynamic programming scheme operating on binary tree decompositions of width at most ℓ, where the state of a node t of the tree decomposition depends only on $\mathsf{bag}(t)$, $\mathsf{edges}(t)$, and the states of the children of t in the tree; and that this state can be computed in time τ from these information. Then, there exists an ℓ-prefix-rebuilding data structure with overhead τ that additionally implements the following operation:*

- $\mathsf{State}(s)$*: Given a node $s \in V(T)$, returns the state of the node s. Runs in worst-case time $\mathcal{O}(1)$.*

In Section 3.7 we formalize what we mean by a "dynamic programming scheme" on tree decompositions through a suitable automaton model. Then Lemma 3.2.2 is formalized by a statement (Lemma 3.7.7) saying that the run of an automaton on a tree decomposition can be maintained under prefix-rebuilding updates, while the first three bullet points of Lemma 3.2.1 are formally proved by applying this statement to specific (very simple) automata. In several places in the sequel, we will need to maintain more complicated dynamic programming schemes on tree decompositions under prefix-rebuilding updates. In every case, we state a suitable lemma about the existence of a prefix-rebuilding data structure, and this lemma is then proved in Section 3.7 using a suitable automaton construction.

Finally, we show that the assumption that the function $\mathsf{edges}^\star$ is given in the description of a prefix-rebuilding update can be lifted in prefix-rebuilding updates that do not change the underlying graph G. Consider a prefix-rebuilding update that does not change the graph G, and let us say that a *weak description* of the update is a tuple $\widehat{u} := (T_{\mathrm{pref}}, T'_{\mathrm{pref}}, T^\star, \mathsf{bag}^\star, \pi)$ that is required to satisfy the same properties as a description of a prefix-rebuilding update except for the edges function. Because the graph G is not changed, the new annotated binary tree decomposition $(T', \mathsf{bag}', \mathsf{edges}')$ can be determined uniquely from $(T, \mathsf{bag}, \mathsf{edges})$ and $\widehat{u}$. We again denote $|\widehat{u}| = |T_{\mathrm{pref}}| + |T'_{\mathrm{pref}}|$.

We show that a weak description $\widehat{u}$ of a prefix-rebuilding update can be turned into a description $\overline{u}$ of a prefix-rebuilding update such that $|\overline{u}| = \mathcal{O}(|\widehat{u}|)$ and the annotated binary tree decomposition $(T', \mathsf{bag}', \mathsf{edges}')$ resulting from applying $\overline{u}$ is the same as the one resulting from applying $\widehat{u}$. We note that this operation can make the sets T_{pref} and T'_{pref} larger, but this is bounded by $\mathcal{O}(|\widehat{u}|)$.

Lemma 3.2.3. *For every $\ell \in \mathbb{N}$, there exists an ℓ-prefix-rebuilding data structure with overhead $\mathcal{O}(1)$ that additionally implements the following operations:*

- $\mathsf{Strengthen}(\widehat{u})$: *Given a weak description $\widehat{u}$ of a prefix-rebuilding operation, returns a description $\overline{u}$ of a prefix-rebuilding operation such that $|\overline{u}| = \mathcal{O}(\widehat{u})$ and applying $\widehat{u}$ and $\overline{u}$ result in the same annotated tree decomposition $(T', \mathsf{bag}', \mathsf{edges}')$. Runs in worst-case time $|\widehat{u}| \cdot \ell^{\mathcal{O}(1)}$.*

Proof. Let $\widehat{u} = (\widehat{T}_{\mathrm{pref}}, \widehat{T'}_{\mathrm{pref}}, \widehat{T}^\star, \widehat{\mathsf{bag}}^\star, \widehat{\pi})$ and $(T', \mathsf{bag}', \mathsf{edges}')$ be the resulting annotated tree decomposition. We observe that the topmost bag containing an edge $uv \in E(G)$ can change only if both $u, v \in \mathsf{bags}_{T'}(\widehat{T'}_{\mathrm{pref}})$. However, if $u, v \in \mathsf{bags}_{T'}(\widehat{T'}_{\mathrm{pref}})$, then because of the vertex condition of (T', bag') it must hold that $u, v \in \mathsf{bags}_T(\widehat{T}_{\mathrm{pref}} \cup \mathsf{App}(\widehat{T}_{\mathrm{pref}}))$, and in particular, the vertex condition in (T, bag) implies that uv must be stored in $\mathsf{edges}(\widehat{T}_{\mathrm{pref}} \cup \mathsf{App}(\widehat{T}_{\mathrm{pref}}))$. Therefore, the changes to the edges function are limited to the subtree of T consisting of $\widehat{T}_{\mathrm{pref}} \cup \mathsf{App}(\widehat{T}_{\mathrm{pref}})$, and therefore for constructing $\overline{u} = (T_{\mathrm{pref}}, T'_{\mathrm{pref}}, T^\star, \mathsf{bag}^\star, \mathsf{edges}^\star, \pi)$ it suffices to take $T_{\mathrm{pref}} = \widehat{T}_{\mathrm{pref}} \cup \mathsf{App}(\widehat{T}_{\mathrm{pref}})$ and analogously construct T'_{pref}, $T^\star$, $\mathsf{bag}^\star$, and π from $\widehat{T'}_{\mathrm{pref}}$, $\widehat{T}^\star$, $\widehat{\mathsf{bag}}^\star$, and $\widehat{\pi}$. Then, the $\mathsf{edges}^\star$ function can be determined from $(T^\star, \mathsf{bag}^\star)$ and the edges function restricted to $\widehat{T}_{\mathrm{pref}} \cup \mathsf{App}(\widehat{T}_{\mathrm{pref}})$. The running time and the bound on $|\overline{u}|$ follow from the fact that the tree decompositions are binary. □

Now, by using the data structure from Lemma 3.2.3, we can assume when constructing prefix-rebuilding operations that do not change G that it is sufficient to construct a weak description, but when implementing prefix-rebuilding data structures that the update method receives a (not weak) description. In the rest of this chapter, we assume that we are always maintaining the data structure from Lemma 3.2.3, in particular, usually first using it to turn a weak description $\widehat{u}$ into a description $\overline{u}$, and then immediately applying $\mathsf{update}(\overline{u})$ to it.

Logic. Recall that Courcelle's theorem states that given a graph G of treewidth k and a CMSO_2 formula φ, it can be decided whether $G \models \varphi$ in time $f(k, \varphi) \cdot n$, where n is the vertex count of G and f is a computable function. In the proof of Courcelle's theorem, one typically first computes a tree decomposition of G of width at most k, for instance using the algorithm of Bodlaender [Bod96], and then applies a dynamic programming procedure (*aka* automaton) suitably constructed from φ to verify the satisfaction of φ. We show that this dynamic programming procedure can be maintained under prefix-rebuilding updates. More formally, in Section 3.7 we prove the following statement.

Lemma 3.2.4. *Fix $\ell \in \mathbb{N}$ and a CMSO_2 sentence φ. Then there exists an ℓ-prefix-rebuilding data structure with overhead $\mathcal{O}_{\ell,\varphi}(1)$ that additionally implements the following operation:*

- $\mathsf{Query}()$: *Returns whether $G \models \varphi$. Runs in worst-case time $\mathcal{O}_{\ell,\varphi}(1)$.*

Dynamic tree decompositions under the promise of small treewidth. With all the definitions in place, we can finally state the core result that will be leveraged to prove Theorem 1.3.1.

Lemma 3.2.5. *There is a data structure that for an integer $k \in \mathbb{N}$, fixed upon initialization, and a dynamic graph G, updated by edge insertions and deletions, maintains an annotated tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of G of width at most $6k+5$ using prefix-rebuilding updates under the promise that $\mathrm{tw}(G) \leqslant k$ at all times. More precisely, at every point in time the graph is guaranteed to have treewidth at most k and the data structure contains an annotated tree decomposition of G of width at most $6k+5$. The data structure can be initialized on k and an edgeless n-vertex graph G in time $2^{\mathcal{O}(k^8)} \cdot n$, and then every update:*

- *returns the sequence of prefix-rebuilding updates used to modify the tree decomposition; and*
- *takes amortized time $2^{\mathcal{O}(k^9 + k \log k \cdot \sqrt{\log n \log\log n})}$.*

Note that as a direct consequence of Lemma 3.2.5, the total size of all prefix-rebuilding updates returned by the data structure over first q edge insertions/deletions is bounded by

$$2^{\mathcal{O}(k^8)} \cdot n + 2^{\mathcal{O}(k^9 + k \log k \cdot \sqrt{\log n \log\log n})} \cdot q.$$

Lemma 3.2.5 is proved in Section 3.6 using the results of Sections 3.3 to 3.5. We remark that the statement of the lemma is essentially a weaker version of Theorem 1.3.1: First, we assume that no update increasing $\mathrm{tw}(G)$ above k may ever arrive to the data structure; next, we do not support dynamic CMSO_2 model checking. We fix these issues in Section 3.8 by means of, respectively: a straightforward application of the technique of postponing invariant-breaking insertions of Eppstein et al. [EGIS96], and Lemma 3.2.4.

3.3 Closures

In this section, we introduce a graph-theoretical notion of a *closure*, which will be used in the presentation of our algorithm later in the chapter. For the rest of the section, fix an integer $k \in \mathbb{N}$ and let G be a graph of treewidth at most k.

Intuitively, when one tries to maintain a tree decomposition of a graph dynamically, one inevitably reaches a situation where some of the bags of the maintained tree decomposition are too large. Consider the following naive approach of improving such a tree decomposition: Let W be the union of the bags that are deemed too large. Construct a (rooted) tree decomposition T_W of $\mathsf{torso}_G(W)$. Then, for each connected component $C \in \mathrm{cc}(G - W)$, the set $N(C)$ is a clique in $\mathsf{torso}_G(W)$; hence, the entire set $N(C)$ resides in a single bag t_C of T_W. Therefore, C can be incorporated into T_W by constructing a tree decomposition T_C of $G[N[C]]$ whose root bag contains $N(C)$ entirely, and then attaching the root of T_C to the bag $t_C \in V(T_W)$. It can be straightforwardly verified that this is a valid construction of a tree decomposition of G.

However, it is not clear why this construction would improve the width of the maintained decomposition. This owes to the fact that the treewidth of $\mathsf{torso}_G(W)$ might be in principle much larger than k. One might, however, hope that the set W can be covered by an only slightly larger set $X \supseteq W$ such that the treewidth of $\mathsf{torso}_G(X)$ is small. This is, indeed, the case, leading to the definition of a *closure* of W:

Definition 1. *Let $k \in \mathbb{N}$ and G be such that $\mathrm{tw}(G) \leqslant k$. Let also $W \subseteq V(G)$. Then, the set $X \subseteq V(G)$ is called the* k-closure *of W in G if*

$$X \supseteq W \qquad \text{and} \qquad \mathrm{tw}(\mathsf{torso}_G(X)) \leqslant 2k+1.$$

Note that each set W admits a trivial closure $X = V(G)$, as $\mathsf{torso}_G(V(G)) = G$. Obviously, such a closure might be much larger than W. Fortunately, the following lemma shows how to construct closures of more manageable size:

Lemma 3.3.1. *Let (T, bag) be a tree decomposition of G of width at most k. Let also S be an lca-closed set of nodes of T. Then,*

$$\mathrm{tw}(\mathsf{torso}_G(\mathsf{bags}(S))) \leqslant 2k+1.$$

Proof. We construct a rooted tree U in the following way: Let $V(U) = S$ and let t_1 be a parent of t_2 in U if and only if t_1 is a strict ancestor of t_2 and the simple path between t_1 and t_2 in T does not contain any other vertices of S. Since S is lca-closed, it can be easily verified that U is indeed a rooted tree. We also construct a tree decomposition (U, bag') as follows:

$$\mathsf{bag}'(t) = \begin{cases} \mathsf{bag}(t) & \text{if } t \text{ is the root of } U, \\ \mathsf{bag}(t) \cup \mathsf{bag}(\mathsf{parent}_U(t)) & \text{otherwise.} \end{cases}$$

We claim that (U, bag') is a tree decomposition of $\mathsf{torso}_G(\mathsf{bags}(S))$ of width at most $2k+1$. The vertex condition is straightforward to verify. For the edge condition, consider an edge uv of $\mathsf{torso}_G(\mathsf{bags}(S))$. We have that $u, v \in \mathsf{bags}(S)$ and there exists a simple path P between u and v in G that is internally disjoint with $\mathsf{bags}(S)$. Pick two nodes t_u, t_v of S such that $u \in \mathsf{bag}(t_u)$, $v \in \mathsf{bag}(t_v)$. For each node t on the path between t_u and t_v in U, the set $\mathsf{bag}(t)$ must contain either u or v; otherwise, $\mathsf{bag}(t)$ would be a separator between u and v in G disjoint with $\{u, v\}$, contradicting the existence of P. Hence, one of the following must hold:

- Both u and v belong to $\mathsf{bag}(t)$ for some $t \in S$. Then $u, v \in \mathsf{bag}'(t)$, so the edge condition is satisfied for the edge uv.
- We have $u \in \mathsf{bag}(t_1)$, $v \in \mathsf{bag}(t_2)$ for some nodes $t_1, t_2 \in S$ that are adjacent in U. Without loss of generality, assume that t_1 is the parent of t_2. Then, since $\mathsf{bag}'(t_2) = \mathsf{bag}(t_1) \cup \mathsf{bag}(t_2)$, we infer that $u, v \in \mathsf{bag}'(t_2)$.

We conclude that (U, bag') is indeed a tree decomposition of $\mathsf{torso}_G(\mathsf{bags}(S))$. Since each bag of (U, bag') has size at most $2k+2$, the proof is finished. □

Lemma 3.3.1 already shows that each set $W \subseteq V(G)$ admits a closure X of cardinality at most $\mathcal{O}(k) \cdot |W|$. Indeed, consider a tree decomposition $(T_{\mathrm{opt}}, \mathsf{bag})$ of G of minimum width. For each vertex $v \in W$, select into S a node $t \in V(T_{\mathrm{opt}})$ such that $v \in \mathsf{bag}(t)$. Then, take the lca-closure of S (which increases $|S|$ by a factor of at most 2) and apply Lemma 3.3.1.

Unfortunately, this will not be sufficient in our setting. In our algorithm, as we maintain a tree decomposition (T, bag), the set W will be chosen as the union of bags in a prefix T_{pref} of T. In this setup, another condition on the closure $X \supseteq W$ will be required: For every appendix t of T_{pref}, we require that the entire component $\mathsf{cmp}(t)$ contains only a bounded number of vertices of X. Such closures will be called *small*. The existence of such a closure will be proved as Lemma 3.3.2 (Small Closure Lemma) in Section 3.3.1. This is followed in Section 3.3.2 by proving a structural result about small closures (Lemma 3.3.7, Closure Linkedness Lemma). Next, in Section 3.3.3, we will define objects related to closures that will be central to the tree decomposition improvement algorithm – *blockages*, *explorations* and *collected components* – as well as prove several structural properties of these notions. Finally, Section 3.3.4 sketches how to find small closures efficiently in a dynamically changing tree decomposition.

3.3.1 Small Closure Lemma

We now formally define the notion of small closures.

Definition 2. *Let (T, bag) be a tree decomposition of G and T_{pref} be a prefix of T. Let also $c \in \mathbb{N}$ be an integer. Then we say that a set $X \supseteq \mathsf{bags}(T_{\mathrm{pref}})$ is* c-small *with respect to (T, bag) if for every appendix t of T_{pref}, it holds that $|X \cap \mathsf{cmp}(t)| \leqslant c$.*

With this definition in place, we are ready to state the Small Closure Lemma, asserting the existence of c-small closures for c large enough:

Lemma 3.3.2 (Small Closure Lemma)**.** *There exists a function $g(\ell) \in \mathcal{O}(\ell^4)$ such that the following holds. Let $k, \ell \in \mathbb{N}$ with $k \leqslant \ell$ and G be a graph with $\mathrm{tw}(G) \leqslant k$. Let (T, bag) be a tree decomposition of G of width at most ℓ and $T_{\mathrm{pref}} \subseteq V(T)$ be a prefix of T. Then there exists a $g(\ell)$-small k-closure of $\mathsf{bags}(T_{\mathrm{pref}})$ with respect to (T, bag).*

The proof of Lemma 3.3.2 uses the machinery of Bojańczyk and Pilipczuk [BP22] in the form of the Dealternation Lemma: Intuitively, since T is a bounded-width decomposition of G, there exists a well-structured tree decomposition U of G such that for each appendix t of T_{pref}, the component $\mathsf{cmp}(t)$ can be partitioned into a bounded number of well-structured "chunks" of U. The closure X will be constructed so that each chunk of U contains only a bounded number of vertices from X. Thus, X will include a bounded number of vertices from each $\mathsf{cmp}(t)$.

We now present a formal version of the Dealternation Lemma. The description follows the exposition in [BP22], with some details irrelevant to us omitted.

Elimination forests. The output of the Dealternation Lemma is a tree decomposition of G presented as the so-called *elimination forest*:

Definition 3. *An* elimination forest *of G is a rooted forest F on vertex set $V(G)$ with the following property: if $uv \in E(G)$, then u and v are in the ancestor-descendant relationship in F.*

The following definition shows how to turn an elimination forest of G into a tree decomposition:

Definition 4. *Assume that F is an elimination forest of G (so $V(F) = V(G)$). A tree decomposition* induced *by F is the tree decomposition (F, bag), where for each $v \in V(G)$, we set $\mathsf{bag}(v)$ to contain v and each ancestor of v connected by an edge of G to any descendant of v.*

In the definition above, we slightly abuse the notation and allow the shape of a tree decomposition to be a rooted forest, rather than a rooted tree; all other conditions remain the same. Note that such a forest decomposition can be always turned into a tree decomposition of same width by selecting one root r and making all other roots children of r.

That (F, bag) constructed as in Definition 4 is indeed a tree decomposition of G is argued in [BP22, Section 3]. It is now natural to define the *width* of an elimination forest F as the width of the tree decomposition induced by F. Clearly, each elimination forest has width lower-bounded by $\mathrm{tw}(G)$. On the other hand, every graph G has an elimination forest of width exactly $\mathrm{tw}(G)$ [BP22, Lemma 3.6].

Factors. Intuitively, *factors* are well-structured "chunks" of a forest F. Formally, a factor is a subset of $V(F)$ that is either:

- a *forest factor*: a union of a nonempty set of rooted subtrees of F, whose roots are all siblings to each other; or

- a *context factor*: a nonempty set of the form $\Phi_1 \setminus \Phi_2$, where Φ_1 is a rooted subtree and $\Phi_2 \subseteq \Phi_1$ is a forest factor; the root of a context factor is the root of Φ_1, while the roots of the tree factors in Φ_2 are called the *appendices*.

Dealternation Lemma. We can now state the Dealternation Lemma.

Lemma 3.3.3 (Dealternation Lemma, [BP22])**.** *There exists a function $f(\ell) \in \mathcal{O}(\ell^3)$ such that the following holds. Let (T, bag) be a tree decomposition of G of width at most ℓ. Then there exists an elimination forest F of G of width $\mathrm{tw}(G)$ such that for every node $t \in V(T)$, the set $\mathsf{cmp}(t)$ is a disjoint union of at most $f(\ell)$ factors of F.*

Proof of the Small Closure Lemma. We now show how the Dealternation Lemma implies the Small Closure Lemma (Lemma 3.3.2).

Proof of Lemma 3.3.2. Recall that we are given: a graph G of treewidth at most k; a tree decomposition (T, bag) of G of width at most ℓ, where $\ell \geqslant k$; and a prefix T_{pref} of T. We are supposed to find a small k-closure X of $\mathsf{bags}(T_{\mathrm{pref}})$. We stress that this requires that $\mathrm{tw}(\mathsf{torso}_G(X)) \leqslant 2k+1$.

We begin by applying Lemma 3.3.3 to (T, bag) and getting an elimination forest F of G of width $\mathrm{tw}(G)$, for which each $\mathsf{cmp}(t)$ for $t \in V(T)$ can be decomposed into at most $f(\ell) \in \mathcal{O}(\ell^3)$ factors of F. We remark that $V(F) = V(G)$.

As argued in the paragraph following Definition 4, there exists a tree decomposition (F, bag') of G of width $\mathrm{tw}(G)$, where $\mathsf{bag}'(v)$ for $v \in V(G)$ is defined as the set containing v and every ancestor of v in F incident to an edge whose other endpoint is a descendant of v. Let $W \coloneqq \mathsf{bags}_T(T_{\mathrm{pref}})$ and W' be the lca-closure of W in F. We claim that the set

$$X \coloneqq \bigcup_{v \in W'} \mathsf{bag}'(v)$$

is an $((\ell+1) \cdot f(\ell))$-small k-closure of W with respect to T. This will conclude the proof.

Claim 3.3.4. *X is a k-closure of $\mathsf{bags}_T(T_{\mathrm{pref}})$.*

Proof of the claim. Since $\mathsf{bags}_T(T_{\mathrm{pref}}) \subseteq W'$ and $v \in \mathsf{bag}'(v)$ for each $v \in W'$, we have that $\mathsf{bags}_T(T_{\mathrm{pref}}) \subseteq X$, as required. It remains to show that $\mathrm{tw}(\mathsf{torso}_G(X)) \leqslant 2k+1$. However, as W' is lca-closed in F, Lemma 3.3.1 applies to W' and (every component of) tree decomposition (F, bag'), finishing the proof. ◁

Claim 3.3.5. *Let t be an appendix of T_{pref} and let Φ be a factor of F with $\Phi \subseteq \mathsf{cmp}(t)$. Then*

$$|\Phi \cap X| \leqslant \ell + 1.$$

Proof of the claim. Since t is an appendix of T_{pref}, it follows from the definition of $\mathsf{cmp}(t)$ that $\mathsf{cmp}(t)$ is disjoint with $\mathsf{bags}_T(T_{\mathrm{pref}})$. Thus, Φ is also disjoint with $\mathsf{bags}_T(T_{\mathrm{pref}})$.

First, assume that Φ is a forest factor. Then Φ is downwards closed: if a vertex belongs to Φ, then all its descendants also belong to Φ. Since W' consists of $\mathsf{bags}_T(T_{\mathrm{pref}})$ and a subset of ancestors of $\mathsf{bags}_T(T_{\mathrm{pref}})$, it follows that Φ is disjoint with W'. Now, for each $v \in W'$, the set $\mathsf{bag}'(v)$ comprises v and some ancestors of v. So again, Φ is disjoint with each set $\mathsf{bag}'(v)$ for $v \in W'$ and thus disjoint with X.

Now assume that Φ is a context factor. Recall that $\Phi = \Phi_1 \setminus \Phi_2$, where Φ_1 is a subtree of F rooted at some vertex r, and $\Phi_2 \subseteq \Phi_1$ is a forest factor. The appendices of Φ have a common parent, which we call s. By the disjointness of $\mathsf{bags}_T(T_{\mathrm{pref}})$ with Φ, we see that each vertex of $\mathsf{bags}_T(T_{\mathrm{pref}})$ is either outside of Φ_1 or inside some subtree of Φ_2.

Since W' is the lca-closure of $\mathsf{bags}_T(T_{\mathrm{pref}})$, we have that $W' = \{\mathrm{lca}(u,v) \mid u, v \in \mathsf{bags}_T(T_{\mathrm{pref}})\}$. Consider $u, v \in \mathsf{bags}_T(T_{\mathrm{pref}})$ such that $\mathrm{lca}(u,v) \in \Phi_1$. If either u or v is outside of Φ_1, then $\mathrm{lca}(u,v)$ is also outside of Φ_1. Therefore, both u and v belong to Φ_2. In this case, it can be easily seen that $\mathrm{lca}(u,v)$ either belongs to Φ_2 (if both u and v are from the same rooted tree of Φ_2) or is equal to s (otherwise). Thus,

$$\Phi_1 \cap W' \subseteq \Phi_2 \cup \{s\}.$$

Note that $\Phi_2 \cup \{s\}$ is a connected subgraph of F containing s and some rooted subtrees attached to s.

Again, for each $v \in W'$, the set $\mathsf{bag}'(v)$ comprises v and some ancestors of v. Hence, for $v \in W' \setminus \Phi_1$, the set $\mathsf{bag}'(v)$ is disjoint with Φ. Therefore,

$$\Phi \cap X \subseteq \bigcup \{\Phi \cap \mathsf{bag}'(v) \mid v \in \Phi_2 \cup \{s\}\}.$$

Now let $v \in \Phi_2 \cup \{s\}$ and $x \in \Phi \cap \mathsf{bag}'(v)$. By the definition of bag', x is either:

- equal to v. Then we have that $x \in \Phi_2 \cup \{s\}$ and $x \in \Phi$, so necessarily $x = s$; or
- an ancestor of v that is connected by an edge to a descendant y of v. But since also $x \in \Phi$, we get that x is also an ancestor of s. Also, y is a descendant of s (as y is a descendant of v and v is a descendant of s). We conclude that $x \in \mathsf{bag}'(s)$.

In both cases we have $x \in \mathsf{bag}'(s)$ and therefore

$$\Phi \cap X \subseteq \mathsf{bag}'(s).$$

As (F, bag') is a decomposition of width $\mathrm{tw}(G) \leqslant k \leqslant \ell$, the statement of the claim follows immediately. ◁

Claim 3.3.6. *Let t be an appendix of T_{pref}. Then*

$$|\mathsf{cmp}(t) \cap X| \leqslant (\ell+1)f(\ell).$$

Proof of the claim. By the Dealternation Lemma (Lemma 3.3.3), $\mathsf{cmp}(t)$ can be partitioned into at most $f(\ell)$ disjoint factors of F. By Claim 3.3.5, each such factor intersects X in at most $\ell+1$ elements. ◁

The proof of the Small Closure Lemma follows immediately from Claims 3.3.4 and 3.3.6. □

3.3.2 Minimum-weight closures and Closure Linkedness Lemma

Having established the existence of c-small closures for sufficiently large c, we now show a structural result about such closures: the Closure Linkedness Lemma. Intuitively, we prove that if $X \supseteq W$ is a c-small closure of W optimal with respect to some measure, then each connected component C of $G - X$ is well-connected to W. This property can be thought of as an analog of a similar result in a work of Korhonen and Lokshtanov [KL22, Lemma 5.1]; the difference is that we work with optimal c-small closures, compared to just optimal closures in [KL22].

From now on, let $\omega\colon V(G) \to \mathbb{Z}$ be an arbitrary weight function. For any subset $A \subseteq V(G)$, let $\omega(A) \coloneqq \sum_{v \in A} \omega(v)$. First, let us define a few notions involving weight functions:

Definition 5. *Fix $k, c \in \mathbb{N}$ and a weight function $\omega\colon V(G) \to \mathbb{Z}$. Let G be a graph of treewidth at most k, $W \subseteq V(G)$, and X be a c-small k-closure of W. We say that X is* ω-minimal *if for every c-small k-closure X' of W, one of the following conditions holds:*

- $|X'| > |X|$, *or*
- $|X'| = |X|$ *and* $\omega(X') \geqslant \omega(X)$.

Definition 6 ([KL22])**.** *Let G be a graph, $A, B \subseteq V(G)$, and $\omega : V(G) \to \mathbb{Z}$ be a weight function. We say that a set A is* ω-linked *into B if each (A, B)-separator S satisfies either of the following conditions:*

- $|S| > |A|$,
- $|S| = |A|$ *and* $\omega(S) \geqslant \omega(A)$.

By definition, if A is ω-linked into B, then A is also linked into B.
We can now state and prove the Closure Linkedness Lemma:

Lemma 3.3.7 (Closure Linkedness Lemma)**.** *Fix $k, c \in \mathbb{N}$ and a weight function $\omega : V(G) \to \mathbb{Z}$. Let G be a graph of treewidth at most k, (T, bag) be a tree decomposition of G (of any width), T_{pref} be a prefix of T, and X be an ω-minimal c-small k-closure of $\mathsf{bags}(T_{\mathrm{pref}})$. Then for each $C \in \mathrm{cc}(G - X)$, the set $N(C)$ is ω-linked into $\mathsf{bags}(T_{\mathrm{pref}})$.*

We remark that it follows from Lemma 3.3.7 and the Small Closure Lemma that if the width of the decomposition (T, bag) is $\ell \geqslant k$, then for some large enough constant $c_\ell \in \mathcal{O}(\ell^4)$ there exists a c_ℓ-small k-closure X of $\mathsf{bags}(T_{\mathrm{pref}})$ such that the neighborhood of each connected component of $G - X$ is ω-linked into $\mathsf{bags}(T_{\mathrm{pref}})$. In fact, any such ω-minimal closure will have this property.

The proof of the Closure Linkedness Lemma proceeds by assuming that some set $N(C)$ is not ω-linked into $\mathsf{bags}(T_{\mathrm{pref}})$; then, a small $(N(C), \mathsf{bags}(T_{\mathrm{pref}}))$-separator will exist. This separator will be used to construct a new k-closure X' of $\mathsf{bags}(T_{\mathrm{pref}})$ with smaller weight, thus contradicting the ω-minimality of X. However, in order to prove that X' is a k-closure of $\mathsf{bags}(T_{\mathrm{pref}})$, one needs to show that $\mathsf{torso}_G(X')$ has sufficiently small treewidth. In order to facilitate this argument, we will use a useful technical tool from the work of Korhonen and Lokshtanov [KL22]:

Lemma 3.3.8 (Pulling Lemma, [KL22, Lemma 4.8])**.** *Let G be a graph, $X \subseteq V(G)$ and (T, bag) be a tree decomposition of $\mathsf{torso}_G(X)$. Let (A, S, B) be a separation of G satisfying the following: There exists a node $r \in V(T)$ such that S is linked into $\mathsf{bag}(r) \cap (S \cup B)$. Let also $X' \coloneqq (X \cap A) \cup S$. Then, there exists a tree decomposition (T, bag') of $\mathsf{torso}_G(X')$ of width not exceeding the width of (T, bag).*

We also need a simple helper lemma:

Lemma 3.3.9. *Let G be a graph, $X \subseteq V(G)$ and $C \in \mathrm{cc}(G - X)$. Then $N(C)$ is a clique in $\mathsf{torso}_G(X)$.*

Proof. For every $u, v \in N(C)$, there exists a path connecting u and v whose all internal vertices are contained in C. □

We are now ready to prove the Closure Linkedness Lemma.

Proof of Lemma 3.3.7. Let $W \coloneqq \mathsf{bags}(T_{\mathrm{pref}})$. For the sake of contradiction, assume we have a component $C \in \mathrm{cc}(G - X)$ such that its neighborhood $N(C)$ is *not* ω-linked into W. By Definition 6, there exists an $(N(C), W)$-separator S such that either $|S| < |N(C)|$, or $|S| = |N(C)|$ and $\omega(S) < \omega(N(C))$. Without loss of generality, assume that S is such a separator with minimum possible size. Naturally, S induces a separation (A, S, B) such that $W \subseteq A \cup S$ and $N(C) \subseteq S \cup B$.

Now, construct a new set X' from X as follows:

$$X' \coloneqq (X \cap A) \cup S.$$

We claim that X' is also a c-small k-closure of W. Note that since $N(C)$ is contained in X and disjoint from $X \cap A$, we have that $X' \subseteq (X \setminus N(C)) \cup S$. Therefore, $|X'| \leqslant |X| - |N(C)| + |S|$ and $\omega(X') \leqslant \omega(X) - \omega(N(C)) + \omega(S)$. So if $|S| < |N(C)|$, we have $|X'| < |X|$, and if $|S| = |N(C)|$ and $\omega(S) < \omega(N(C))$, then $|X'| \leqslant |X|$ and $\omega(X') < \omega(X)$. So provided X' is indeed a c-small k-closure of W, X cannot be ω-minimal, contradicting our assumption.

Claim 3.3.10. *X' is a k-closure of W.*

Proof of the claim. Since W is disjoint with B and X is a k-closure of W, we get that $X' \supseteq W$.

Aiming to use the Pulling Lemma, we let (T_X, bag_X) to be a tree decomposition of $\mathsf{torso}_G(X)$ of width at most $2k + 1$. By Lemma 3.3.9, $N(C)$ is a clique in $\mathsf{torso}_G(X)$, so there exists a node $r \in V(T_X)$ with $N(C) \subseteq \mathsf{bag}_X(r)$. It remains to verify that S is linked into $\mathsf{bag}_X(r) \cap (S \cup B)$.

Observe that $N(C) \subseteq \mathsf{bag}_X(r) \cap (S \cup B)$. Hence, it is enough to check that S is linked into $N(C)$. However, if it was not the case, then there would be an $(N(C), S)$-separator S' of size $|S'| < |S|$. But then S' would be also an $(N(C), W)$-separator of size smaller than $|S|$, contradicting the minimality of S. Hence, Lemma 3.3.8 applies to the tree decomposition (T_X, bag_X) and the separation (A, S, B), producing a tree decomposition of $\mathsf{torso}_G(X')$ of width not exceeding $2k + 1$. ◁

Claim 3.3.11. *X' is c-small.*

Proof of the claim. Recall that $X \supseteq W$, where $W = \mathsf{bags}(T_{\mathrm{pref}})$. Therefore, as C is a connected component of $G - X$, the entire connected component C must be contained within $\mathsf{cmp}(t)$ for some appendix t of T_{pref}. Hence, $N(C) \subseteq \mathsf{cmp}(t) \cup \mathsf{adh}(t)$. Also, $\mathsf{adh}(t) \subseteq W$. As $(\mathsf{cmp}(t), \mathsf{adh}(t), V(G) \setminus (\mathsf{cmp}(t) \cup \mathsf{adh}(t)))$ is a separation of G, and S is a minimum-size $(N(C), W)$-separator, it follows that $S \subseteq \mathsf{cmp}(t) \cup \mathsf{adh}(t)$; in other words, vertices outside of $\mathsf{cmp}(t) \cup \mathsf{adh}(t)$ are not useful towards the separation of $N(C)$ from W.

Now, for an appendix $t' \neq t$ of T_{pref}, $\mathsf{cmp}(t')$ is disjoint from $\mathsf{cmp}(t) \cup \mathsf{adh}(t)$: This is because $\mathsf{cmp}(t')$ is disjoint from W (containing $\mathsf{adh}(t)$ in its entirety) and from $\mathsf{cmp}(t)$ (as t, t' do not remain in the ancestor-descendant relationship in T). Therefore, S is disjoint with $\mathsf{cmp}(t')$ and thus, by the definition of X',

$$|X' \cap \mathsf{cmp}(t')| = |(X \cap A) \cap \mathsf{cmp}(t')| \leqslant |X \cap \mathsf{cmp}(t')| \leqslant c.$$

Hence, the smallness condition is not violated for the appendix t'.

In order to prove that the same condition is satisfied for the appendix t, we observe that $N(C) \cap W \subseteq S$ (as S separates $N(C)$ from W), so also $N(C) \cap \mathsf{adh}(t) \subseteq S$. Therefore, $|N(C) \cap \mathsf{adh}(t)| \leqslant |S \cap \mathsf{adh}(t)|$, and we get that

$$|N(C) \cap \mathsf{cmp}(t)| = |N(C)| - |N(C) \cap \mathsf{adh}(t)| \geqslant |S| - |S \cap \mathsf{adh}(t)| = |S \cap \mathsf{cmp}(t)|.$$

Now, as $X' \subseteq (X \setminus N(C)) \cup S$ and $N(C) \subseteq X$, we infer that

$$|X' \cap \mathsf{cmp}(t)| \leqslant |X \cap \mathsf{cmp}(t)| - |N(C) \cap \mathsf{cmp}(t)| + |S \cap \mathsf{cmp}(t)| \leqslant |X \cap \mathsf{cmp}(t)| \leqslant c. \qquad \triangleleft$$

By Claims 3.3.10 and 3.3.11 we infer that X' is a c-small k-closure of W. This is a contradiction to the ω-minimality of X. □

3.3.3 Closure exploration, blockages and collected components

In this section, we define several auxiliary objects that will be constructed in the tree decomposition improvement algorithm after a suitable closure is found: explorations, blockages and collected components. For the remainder of the section, let us fix a tree decomposition (T, bag) of G of width at most ℓ, a nonempty prefix T_{pref} of T and a ω-minimal c-small k-closure $X \supseteq \mathsf{bags}(T_{\mathrm{pref}})$ for some weight function $\omega\colon \mathbb{N} \to \mathbb{N}$.

Blockages. We start with the definition of a blockage:

Definition 7. *We say that a node $t \in V(T) \setminus T_{\mathrm{pref}}$ is a* blockage *in T with respect to T_{pref} and X if one of the following cases holds:*

- $\mathsf{bag}(t) \subseteq N_G[C]$ *for some component $C \in \mathrm{cc}(G - X)$ that intersects* $\mathsf{bag}(t)$ (component blockage)*;*
- $\mathsf{bag}(t) \subseteq X$ *and* $\mathsf{bag}(t)$ *is a clique in* $\mathsf{torso}_G(X)$ (clique blockage)*;*

and no strict ancestor of t is a blockage.

Note that a component blockage intersects with exactly one component $C \in \mathrm{cc}(G - X)$.

Since T, T_{pref} and X will usually be known from the context, we will usually just say that t is a blockage. Then, we introduce the following notation: let $\mathsf{Blockages}(T_{\mathrm{pref}}, X)$ be the set of blockages.

Properties of blockages. We now prove a few lemmas relating blockages to X.

Lemma 3.3.12. *If $t \in \mathsf{Blockages}(T_{\mathrm{pref}}, X)$, then for every pair of vertices $u, v \in \mathsf{bag}(t)$ there exists a uv-path in G internally disjoint with X.*

Proof. If t is a component blockage for component $C \in \mathrm{cc}(G - X)$, then the statement of the lemma immediately follows from $\mathsf{bag}(t) \subseteq N_G[C]$ – in fact, for all $u, v \in \mathsf{bag}(t)$, there exists a uv-path whose all internal vertices belong to C. On the other hand, if t is a clique blockage, then the statement is equivalent to the assertion that $\mathsf{bag}(t)$ is a clique in $\mathsf{torso}_G(X)$. □

Note that it immediately follows from Lemma 3.3.12 that if t is a blockage, then $\mathsf{bag}(t) \cap X$ is a clique in $\mathsf{torso}_G(X)$.

Lemma 3.3.13. *If $t \in \mathsf{Blockages}(T_{\mathrm{pref}}, X)$, then $\mathsf{cmp}(t) \cap X = \emptyset$.*

Proof. Assume otherwise. Then, we claim that the set $X' \coloneqq X \setminus \mathsf{cmp}(t)$ is also a c-small k-closure of $\mathsf{bags}(T_{\mathrm{pref}})$, contradicting the minimality of X. In fact, we will show that $\mathsf{torso}_G(X')$ is a subgraph of $\mathsf{torso}_G(X)$. Towards this goal, choose vertices $u, v \in X'$ and assume that $uv \in E(\mathsf{torso}_G(X'))$, i.e., there exists a path $P = v_1, v_2, \ldots, v_p$ from $u = v_1$ to $v = v_p$ internally disjoint with X'. If P is internally disjoint with X, then also $uv \in E(\mathsf{torso}_G(X))$, finishing the proof. In the opposite case, P must intersect $X \setminus X'$ and thus intersect $\mathsf{cmp}(t)$. Note that by the definition of X', we necessarily have $u, v \notin \mathsf{cmp}(t)$. Let v_a, v_b $(1 \leqslant a \leqslant b \leqslant p)$ be the first and the last intersection of P with $\mathsf{adh}(t)$, respectively; such vertices exist since $(\mathsf{cmp}(t), \mathsf{adh}(t), (V(G) \setminus (\mathsf{cmp}(t) \cup \mathsf{adh}(t))))$ is a separation of G. The subpaths $v_1, v_2, \ldots, v_a$ and $v_b, v_{b+1}, \ldots, v_p$ are disjoint with $\mathsf{cmp}(t)$ and thus are internally disjoint with X. Construct a new path P' from u to v by concatenating:

- the subpath $v_1, v_2, \ldots, v_a$,
- a path from v_a to v_b internally disjoint with X (its existence is asserted by Lemma 3.3.12),
- the subpath $v_b, v_{b+1}, \ldots, v_p$.

Naturally, P' is again internally disjoint with X'. We claim that P' is internally disjoint with X, thus witnessing that $uv \in E(\mathsf{torso}_G(X))$. Indeed, each of the three segments of P' is internally disjoint with X, so P' can internally intersect X only if $v_a \in X$ (or $v_b \in X$). However, in this case, we have that $v_a \in X'$ (respectively, $v_b \in X'$), as $v_a, v_b \notin \mathsf{cmp}(t)$. But P' is internally disjoint with X', hence v_a (resp., v_b) must be the first (resp., the last) vertex of P' and thus not an internal vertex of P'. Hence, P' is internally disjoint with X. □

We remark that if t is a clique blockage, then from Lemma 3.3.13 it follows that $\mathsf{bag}(t) = \mathsf{adh}(t)$, so in particular $\mathsf{bag}(t) \subseteq \mathsf{bag}(\mathsf{parent}(t))$.

Exploration and exploration graph. For a given weight function $\omega\colon V(G) \to \mathbb{Z}$, prefix T_{pref} of T and a ω-minimal c-small k-closure X of $\mathsf{bags}(T_{\mathrm{pref}})$, we define the *exploration* as the prefix $\mathsf{Exploration}(T_{\mathrm{pref}}, X)$ of T whose set of appendices is given by $\mathsf{Blockages}(T_{\mathrm{pref}}, X)$. A node $t \in V(T)$ is deemed *explored* if $t \in \mathsf{Exploration}(T_{\mathrm{pref}}, X)$, otherwise it is *unexplored*. Then, a vertex $v \in V(G)$ is explored if it belongs to $\mathsf{bag}(t)$ for some explored node t, and unexplored otherwise. Observe that $v \in V(G)$ is unexplored if and only if it belongs to $\mathsf{cmp}(t)$ for some blockage t. In particular, by Lemma 3.3.13, every vertex of X is explored.

Clearly, we have $T_{\mathrm{pref}} \subseteq \mathsf{Exploration}(T_{\mathrm{pref}}, X)$. Also, if T is a binary tree decomposition, then $|\mathsf{Blockages}(T_{\mathrm{pref}}, X)| \leqslant |\mathsf{Exploration}(T_{\mathrm{pref}}, X)| + 1$ (this follows immediately from the fact that the number of appendices of any prefix of T exceeds the cardinality of the prefix by at most 1).

Next, we define the *exploration graph* $H \coloneqq \mathsf{ExplorationGraph}(T_{\mathrm{pref}}, X)$ by compressing the components $\mathsf{cmp}(t)$ of blockages t to single vertices. The purpose of this definition will be to present the connected components of $G - X$ in a way that can be bounded by $|\mathsf{Exploration}(T_{\mathrm{pref}}, X)|$, even if the number of such components can be much larger. Formally:

- $V(H)$ comprises: *explored vertices*, that is, the set of vertices $\bigcup_{t \in \mathsf{Exploration}(T_{\mathrm{pref}}, X)} \mathsf{bag}(t)$; and *blockage vertices*, that is, the set $\mathsf{Blockages}(T_{\mathrm{pref}}, X)$ of nodes of T.
- The set of edges $E(H)$ is constructed by taking the subgraph of G induced by the explored vertices and adding edges tv for each $t \in \mathsf{Blockages}(T_{\mathrm{pref}}, X)$ and $v \in \mathsf{adh}(t)$.
- The *compression mapping* $\xi\colon V(G) \to V(H)$ is an identity mapping on $V(G) \cap V(H)$; and for $t \in \mathsf{Blockages}(T_{\mathrm{pref}}, X)$, we set $\xi^{-1}(t) \coloneqq \mathsf{cmp}(t)$.

Note that Lemma 3.3.13 implies that $X \subseteq V(H)$. Also, if T is a binary tree decomposition, then

$$|V(H)| \leqslant (\ell + 1) \cdot |\mathsf{Exploration}(T_{\mathrm{pref}}, X)| + |\mathsf{Exploration}(T_{\mathrm{pref}}, X)| + 1.$$

We also observe the following fact:

Lemma 3.3.14. *If $uv \in E(G)$, then $\xi(u) = \xi(v)$ or $\xi(u)\xi(v) \in E(H)$.*

Proof. If both u and v are explored, then $\xi(u) = u$, $\xi(v) = v$ and $\xi(u)\xi(v) \in E(H)$. Otherwise, assume without loss of generality that $\xi(v) = t$ is a blockage vertex. Then $v \in \mathsf{cmp}(t)$, so $u \in \mathsf{cmp}(t) \cup \mathsf{adh}(t)$. If $u \in \mathsf{cmp}(t)$, then $\xi(u) = \xi(v) = t$. Otherwise $u \in \mathsf{adh}(t)$, so u is an explored vertex. Then $\xi(u) = u$ and $\xi(v) = t$, so $\xi(u)\xi(v) \in E(H)$ by the construction. □

Collected components. Using the notation above, we partition $V(G) \setminus X$ into subsets called *collected components* as follows. For each $C \in \mathrm{cc}(H - X)$, let $\mathfrak{C} \coloneqq \xi^{-1}(C)$ (so $\mathfrak{C} \subseteq V(G) \setminus X$) and if $\mathfrak{C}$ is nonempty, then include $\mathfrak{C}$ in the set $\mathsf{Coll}(T_{\mathrm{pref}}, X)$ of collected components of G. In other words, we uncompress each connected component of $H - X$ to form a collected component – a subset of $V(G) \setminus X$. It can be easily verified that the set $\mathsf{Coll}(T_{\mathrm{pref}}, X)$ is, indeed, a partition of $V(G) \setminus X$.

Let $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$. We say that $\mathfrak{C}$ is:

- *unblocked* if $\xi(\mathfrak{C})$ contains at least one explored vertex of $V(H)$; or
- *blocked* if $\xi(\mathfrak{C})$ only contains blockage vertices of $V(H)$.

Note that if t is a clique blockage, then t is an isolated vertex of $H - X$; hence, the collected component $\xi^{-1}(\{t\})$ (if it is nonempty) is necessarily a blocked component. However, the converse implication is not true in general: It could be that a component blockage t' satisfies $\mathsf{adh}(t') \subseteq X$, again causing t' to be an isolated vertex of $H - X$. Then, the collected component $\xi^{-1}(\{t'\})$ (if it is nonempty) will, too, be a blocked component.

It turns out that $\mathsf{Coll}(T_{\mathrm{pref}}, X)$ is a coarser partition of $V(G) \setminus X$ than $\mathrm{cc}(G - X)$:

Lemma 3.3.15. *Each collected component $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$ is the union of a collection of connected components of $G - X$.*

Proof. It is enough to prove the following: If $u, v \in V(G) \setminus X$ are adjacent in G, then $\xi(u)$ and $\xi(v)$ belong to the same connected component of H. But by Lemma 3.3.14, $\xi(u)$ and $\xi(v)$ are either equal or adjacent in H; this proves the claim. □

Next, we prove a characterization of each type of collected component.

Lemma 3.3.16. *If $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$ is:*

- *unblocked, then the set of explored vertices belonging to $\xi(\mathfrak{C})$ is contained in a single connected component $C \in \mathrm{cc}(G - X)$. Moreover, $C \subseteq \mathfrak{C}$ and $N(\mathfrak{C}) = N(C)$;*
- *blocked, then $\xi(\mathfrak{C})$ consists of a single blockage vertex t. Moreover, $N(\mathfrak{C}) \subseteq \mathsf{adh}(t) \cap X$.*

Proof. We start by proving the following claim:

Claim 3.3.17. *If $u, v \in \mathfrak{C}$ are explored, then u and v belong to the same connected component of $G - X$.*

Proof of the claim. Let $P = w_1 w_2 \dots w_p$ ($w_1 = \xi(u)$, $w_p = \xi(v)$) be a simple path from $\xi(u)$ to $\xi(v)$ in $H - X$ (such a path exists since u, v are in the same collected component). We will prove inductively that each explored vertex of P is in the same connected component of $G - X$ as w_1. This is trivially true for w_1. For an inductive step, choose an explored vertex w_i ($i \geqslant 2$). If w_{i-1} is also explored, then $w_{i-1}w_i$ is an edge of $G - X$, so w_i is in the same connected component of $G - X$ as w_{i-1} (and thus in the same connected component of $G - X$ as w_1).

If w_{i-1} is, on the other hand, a blockage vertex (i.e., $w_{i-1} \in V(T)$), then note that $i \geqslant 3$ (since w_1 is an explored vertex). Moreover, by the definition of $E(H)$, w_{i-1} is adjacent in H only to explored vertices in $\mathsf{adh}(w_{i-1})$. Therefore, w_{i-2} is also an explored vertex and $w_{i-2}, w_i \in \mathsf{adh}(w_{i-1})$. Since $w_{i-2}, w_i \notin X$, it follows that w_{i-1} is a component blockage, i.e., $\mathsf{bag}(w_{i-1}) \subseteq N_G[C]$ for some component $C \in \mathrm{cc}(G - X)$ intersecting $\mathsf{bag}(t)$. Therefore, both w_{i-2} and w_i belong to C; and by the inductive assumption, so does w_1. ◁

Also, the following simple fact will be useful:

Claim 3.3.18. *If D is a connected component of $G - X$ such that $D \subseteq \mathsf{cmp}(t)$ for a blockage t, then $N(D) \subseteq \mathsf{adh}(t) \cap X$.*

Proof of the claim. Obviously, $N(D) \subseteq X$. However, by Lemma 3.3.13, $\mathsf{cmp}(t) \cap X = \emptyset$. As $N[D] \subseteq \mathsf{cmp}(t) \cup \mathsf{adh}(t)$, it follows that $N(D) \subseteq \mathsf{adh}(t)$. ◁

Now, assume that $\mathfrak{C}$ is unblocked. By Claim 3.3.17, the set of explored vertices belonging to $\xi(\mathfrak{C})$ is contained in a single connected component $C \in \mathrm{cc}(G - X)$. From Lemma 3.3.15, we have that $C \subseteq \mathfrak{C}$. It remains to show that $N(\mathfrak{C}) = N(C)$. To this end, we will argue the following:

Claim 3.3.19. *Let $D \in \mathrm{cc}(G - X)$ be a connected component with $D \subseteq \mathfrak{C}$. Then $N(D) \subseteq N(C)$.*

Proof of the claim. We can assume that $D \neq C$. Then the set $\xi(D)$ only contains blockage vertices; and since blockage vertices form an independent set in H, it follows from Lemma 3.3.14 that $\xi(D)$ comprises just one blockage vertex, say t. Hence $D \subseteq \mathsf{cmp}(t)$, and it also follows from Claim 3.3.18 that $N(D) \subseteq \mathsf{adh}(t) \cap X$.

Let $u \in C$ be any explored vertex (so $\xi(u) = u$). Now, since $C, D \subseteq \mathfrak{C}$ and $\xi(D) = \{t\}$, there must exist a path P from t to u in $H - X$. Let v be the (explored) vertex immediately after t in P. By Claim 3.3.17, $v \in C$; and $v \in \mathsf{adh}(t)$ by construction. It follows that t is a component blockage intersecting $C \in \mathrm{cc}(G - X)$, so $\mathsf{bag}(t) \subseteq N[C]$; in particular, $\mathsf{adh}(t) \cap X \subseteq N(C)$. ◁

As $N(\mathfrak{C}) = \bigcup_{D \in \mathrm{cc}(G-X),\, D \subseteq \mathfrak{C}} N(D)$, we get from Claim 3.3.19 that $N(\mathfrak{C}) = N(C)$, as required.

Finally, assume that $\mathfrak{C}$ is blocked. Since blockage vertices of H form an independent set, it follows from Lemma 3.3.14 that $|\xi(\mathfrak{C})| = 1$. Let t be the blockage vertex that is the only element of $\xi(\mathfrak{C})$. Then Claim 3.3.18 applies to every $D \in \mathrm{cc}(G - X)$ with $D \subseteq \mathfrak{C}$ and implies that $N(D) \subseteq \mathsf{adh}(t) \cap X$. Therefore, $N(\mathfrak{C}) \subseteq \mathsf{adh}(t) \cap X$. □

The next lemma follows easily from the previous lemma:

Lemma 3.3.20. *For each $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$, the set $N(\mathfrak{C})$ is a clique in $\mathsf{torso}_G(X)$.*

Proof. If $\mathfrak{C}$ is unblocked, then $N(\mathfrak{C}) = N(C)$ for some $C \in \mathrm{cc}(G - X)$. By Lemma 3.3.9, $N(C)$ is a clique in $\mathsf{torso}_G(X)$. On the other hand, if $\mathfrak{C}$ is blocked, then $N(\mathfrak{C}) \subseteq \mathsf{adh}(t) \cap X$ for some blockage t. But it follows immediately from the definition of a blockage that $\mathsf{adh}(t) \cap X$ is a clique in $\mathsf{torso}_G(X)$. □

Finally, we define the *home bag* of a collected component $\mathfrak{C}$ to be a node of T selected as follows:

- If $\mathfrak{C}$ is unblocked, then the home bag of $\mathfrak{C}$ is the shallowest node t of T such that $\mathsf{bag}(t)$ contains an explored vertex in $\mathfrak{C}$. Note that this bag is uniquely defined, as the explored vertices in $\mathfrak{C}$ induce a connected subgraph of G and hence their occurrences in (T, bag) induce a connected subtree of T.
- If $\mathfrak{C}$ is blocked, then the home bag of $\mathfrak{C}$ is the unique blockage t such that $\xi(\mathfrak{C}) = \{t\}$.

We conclude by proving some properties of the home bags of collected components.

Lemma 3.3.21. *Let $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$ be unblocked and let t be the home bag of $\mathfrak{C}$. Then*

$$t \in \mathsf{Exploration}(T_{\mathrm{pref}}, X) \setminus T_{\mathrm{pref}} \qquad \text{and} \qquad |\mathsf{bag}(t)| > |N(\mathfrak{C})|.$$

Proof. First note that t is an explored node: Each explored vertex of $\mathfrak{C}$ is contained in some bag $\mathsf{bag}(u)$ for an explored node u, and the set of explored nodes is ancestor-closed. Also, $t \notin T_{\mathrm{pref}}$ is immediate: We have $\mathsf{bags}(T_{\mathrm{pref}}) \subseteq X$ by the definition, but $\mathfrak{C}$ is disjoint from X. We conclude that $t \in \mathsf{Exploration}(T_{\mathrm{pref}}, X) \setminus T_{\mathrm{pref}}$.

Next we show that $|\mathsf{bag}(t)| > |N(\mathfrak{C})|$. Recall from Lemma 3.3.16 that there is some $D \in \mathrm{cc}(G - X)$ that contains all explored vertices of $\mathfrak{C}$ and such that $N(\mathfrak{C}) = N(D)$. Since t is an explored node of T, $\mathsf{bag}(t) \setminus X$ consists only of explored vertices. Hence, t is the shallowest bag of T intersecting D. Because $G[D]$ is connected, all bags of T intersecting D must be in the subtree of T rooted at t. Additionally, by the Closure Linkedness Lemma (Lemma 3.3.7), $N(D)$ is ω-linked into $\mathsf{bags}(T_{\mathrm{pref}})$. As $t \notin T_{\mathrm{pref}}$, we find that $\mathsf{adh}(t)$ is an $(N(D), \mathsf{bags}(T_{\mathrm{pref}}))$-separator, from which it follows that $|\mathsf{adh}(t)| \geqslant |N(D)|$. However, it cannot be that $\mathsf{adh}(t) = \mathsf{bag}(t)$ since $\mathsf{adh}(t) \cap D = \emptyset$ and $\mathsf{bag}(t) \cap D \neq \emptyset$. Thus, $\mathsf{adh}(t) \subsetneq \mathsf{bag}(t)$ and so $|\mathsf{bag}(t)| > |N(D)|$. □

Lemma 3.3.22. *Let $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$ be blocked and let t be the home bag of $\mathfrak{C}$. Assume that $\mathsf{parent}(t) \notin T_{\mathrm{pref}}$. Then $|\mathsf{bag}(t) \cap X| < |\mathsf{bag}(\mathsf{parent}(t))|$.*

Proof. Since $\mathfrak{C}$ is blocked, t is a blockage. From Lemma 3.3.13 we have $\mathsf{cmp}(t) \cap X = \emptyset$, so $\mathsf{bag}(t) \cap X = \mathsf{adh}(t) \cap X$. But it cannot be that $\mathsf{adh}(t) \cap X = \mathsf{bag}(\mathsf{parent}(t))$: Otherwise, as $\mathsf{adh}(t) \cap X$ is a clique in $\mathsf{torso}_G(X)$ (Lemma 3.3.12), $\mathsf{parent}(t)$ would be a clique blockage and then t could not be a blockage itself. Since $\mathsf{adh}(t) \subseteq \mathsf{bag}(\mathsf{parent}(t))$, we conclude that $|\mathsf{bag}(t) \cap X| < |\mathsf{bag}(\mathsf{parent}(t))|$. □

3.3.4 Computing closures and auxiliary objects

We finally show that the objects defined in the previous sections can be computed efficiently in a tree decomposition that is changing dynamically under prefix-rebuilding updates. Recall that such decompositions are modeled by the annotated tree decompositions, and data structures maintaining such dynamic decompositions are called prefix-rebuilding data structures. In this section, we fix a specific weight function: For an annotated tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of a graph G, as the weight function we use the *depth function* $d_{\mathcal{T}}$ of $\mathcal{T}$: the function

$$d_{\mathcal{T}} : V(G) \to \mathbb{Z}_{\geqslant 0}$$

that maps each vertex $v \in V(G)$ to the depth of the shallowest node $t \in V(T)$ such that $v \in \mathsf{bag}(t)$. Henceforth, whenever the tree decomposition $\mathcal{T}$ is known from the context, we will write d instead of $d_{\mathcal{T}}$.

We are now ready to state the promised result.

Lemma 3.3.23. *For every $c, \ell, k \in \mathbb{N}$ with $\ell \geqslant k$, there exists an ℓ-prefix-rebuilding data structure with overhead $2^{\mathcal{O}((c+\ell)^2)}$, additionally supporting the following operation:*

- $\mathsf{Query}(T_{\mathrm{pref}})$*: Given a prefix T_{pref} of T, returns either* No closure *if there is no c-small k-closure of $\mathsf{bags}(T_{\mathrm{pref}})$, or*
  - *a $d_{\mathcal{T}}$-minimal c-small k-closure X of $\mathsf{bags}(T_{\mathrm{pref}})$,*
  - *the graph $\mathsf{torso}_G(X)$, and*
  - *the set $\mathsf{Blockages}(T_{\mathrm{pref}}, X) \subseteq V(T)$.*

  This operation runs in worst-case time $2^{\mathcal{O}(k(c+\ell)^2)} \cdot (|T_{\mathrm{pref}}| + |\mathsf{Exploration}(T_{\mathrm{pref}}, X)|)$.

The proof of Lemma 3.3.23 is provided in Section 3.7.4.

3.4 Refinement data structure

In this section we define a data structure, called the *refinement data structure*, which will be used in Sections 3.5 and 3.6 to improve a tree decomposition of a given dynamic graph.

Let G be a graph of treewidth at most k, which is changing over time by edge insertions and deletions. Our ultimate goal is to maintain an annotated binary tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of G of width at most $6k+5$ (we usually write $\mathcal{T} = (T, \mathsf{bag})$ when we are not using the edges function). The aim of the refinement operation is twofold. First, it is used to improve the width of the decomposition, that is, when given a prefix T_{pref} of (T, bag) that contains all the bags of size more than $6k+6$, the refinement operation outputs a tree decomposition (T', bag') of width at most $6k+5$. Second, it is called when the tree of the decomposition becomes *too unbalanced*. This shall be clarified later in Section 3.5.

Before proceeding with the description of the refinement data structure, let us introduce the potential function which plays a major role in both the analysis of the amortized complexity of the refinement and in the height-reduction scheme of Section 3.5.

3.4.1 Potential function

Let $k \in \mathbb{N}$ be the upper bound on the treewidth of the dynamic graph, and $\ell = 6k+5$, which is the desired width of the tree decomposition we are maintaining (in particular, the actual width can be at most $6k+6$ when the refinement operation is called). For a node t of a tree decomposition $\mathcal{T} = (T, \mathsf{bag})$, we define its potential by the formula:

$$\Phi_{\ell,\mathcal{T}}(t) := g_\ell(|\mathsf{bag}(t)|) \cdot \mathsf{height}_T(t), \tag{3.1}$$

where

$$g_\ell(x) := (53(\ell+4))^x \qquad \text{for every } x \in \mathbb{N}.$$

Observe that since we maintain a decomposition of width $\mathcal{O}(k)$, we have

$$\Phi_{\ell,\mathcal{T}}(t) \leqslant 2^{\mathcal{O}(k \log k)} \cdot \mathsf{height}_T(t) \qquad \text{for every node } t.$$

Intuitively, the term $g_\ell(|\mathsf{bag}(t)|)$ in the potential function allows us to update the tree decomposition by replacing the node t with $\mathcal{O}(\ell)$ copies of t, where each copy t' has the same height as t but the bag of t' is strictly smaller than that of t. Then, we can pay for such a transformation from the decrease in the potential function. The second term $\mathsf{height}_T(t)$ will turn out to be essential in Section 3.5: It will allow to argue that the potential function can be decreased significantly if the current tree decomposition is too unbalanced.

For a subset $W \subseteq V(T)$, we denote

$$\Phi_{\ell,\mathcal{T}}(W) := \sum_{t \in W} \Phi_{\ell,\mathcal{T}}(t),$$

and similarly, for the whole tree decomposition $\mathcal{T} = (T, \mathsf{bag})$, we set

$$\Phi_\ell(\mathcal{T}) := \sum_{t \in V(T)} \Phi_{\ell,\mathcal{T}}(t).$$

3.4.2 Data structure

The next lemma describes the interface of our data structure.

Lemma 3.4.1 (Refinement data structure)**.** *Fix $k \in \mathbb{N}$ and let $\ell = 6k+5$. There exists an $(\ell+1)$-prefix-rebuilding data structure with overhead $2^{\mathcal{O}(k^8)}$, that maintains a tree decomposition $\mathcal{T} = (T, \mathsf{bag})$ with $N := |V(T)|$ nodes, and additionally supports the following operation:*

- $\mathsf{Refine}(T_{\mathrm{pref}})$*: Given a prefix T_{pref} of T so that T_{pref} contains all nodes of $\mathcal{T}$ with bags of size $\ell+2$, returns a description $\overline{u}$ of a prefix-rebuilding update, so that the tree decomposition $\mathcal{T}'$ obtained by applying $\overline{u}$ has the following properties:*

(WIDTH) $\mathcal{T}'$ has width at most ℓ and

(POT) the following inequality holds:

$$\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}') \geqslant \sum_{t \in T_{\mathrm{pref}}} \mathsf{height}_T(t) - 2^{\mathcal{O}(k \log k)} \cdot \left(|T_{\mathrm{pref}}| + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(t) \right) \cdot \log N.$$

Moreover, it holds that

(RT1) the worst-case running time of $\mathsf{Refine}(T_{\mathrm{pref}})$*, and therefore also* $|\overline{u}|$*, is upper-bounded by*

$$2^{\mathcal{O}(k^9)} \cdot \left(\left(|T_{\mathrm{pref}}| + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(t) \right) \cdot \log N + \max(\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}'), 0) \right).$$

In the remainder we present the data structure of Lemma 3.4.1 and prove some key properties used later. This section ends with a proof of the correctness of the refinement operation, including the proof of property (WIDTH). The amortized analysis of the data structure, in particular the proofs of properties (RT1) and (POT) will follow in Section 3.4.3.

Fix positive integers k, ℓ such that $\ell = 6k + 5$. Let G be a dynamic n-vertex graph of treewidth at most k. In our data structure we store:

- a binary annotated tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of G of width at most $\ell + 1$;
- an instance $\mathbb{D}^{\mathrm{aux}}$ of a data structure from Lemma 3.2.1 for maintaining various auxiliary information about $\mathcal{T}$; and
- an instance $\mathbb{D}^{\mathrm{explore}}$ of a data structure from Lemma 3.3.23 used to compute necessary objects for the refinement operation.

Implementation of the update operations on our data structure is simple. Upon receiving a prefix-rebuilding update $\overline{u}$ with a prefix T_{pref}, we recompute the decomposition $(T, \mathsf{bag}, \mathsf{edges})$ according to $\overline{u}$, and pass $\overline{u}$ to the inner data structures $\mathbb{D}^{\mathrm{aux}}$ and $\mathbb{D}^{\mathrm{explore}}$. The initialization of the data structure is similarly easy.

From now on, we focus on the refinement operation. Let $T_{\mathrm{pref}} \subseteq V(T)$ be the given prefix of T which includes all bags of $\mathcal{T}$ of size $\ell + 2$.

For the ease of presentation, instead of constructing a description $\overline{u}$ of a prefix-rebuilding update, we shall show how to construct a tree decomposition (T', bag') obtained from (T, bag) after applying $\overline{u}$. The suitable description $\overline{u}$ can be then easily extracted from (T', bag') and from the objects computed along the way, in particular, Lemma 3.2.3 will be implicitly used here.

The refinement operation proceeds in five steps.

Step (1) (Compute auxiliary objects)

Given a prefix T_{pref} of the decomposition $\mathcal{T} = (T, \mathsf{bag})$ we use the data structure $\mathbb{D}^{\mathrm{explore}}$ from Lemma 3.3.23 to compute the following objects:

- a $d_{\mathcal{T}}$-minimal c-small k-closure X of $\mathsf{bags}(T_{\mathrm{pref}})$, where $d_{\mathcal{T}}$ is the depth function of $\mathcal{T}$ defined in Section 2.3 and $c \in \mathcal{O}(k^4)$ is the bound given Lemma 3.3.2,
- the graph $\mathsf{torso}_G(X)$, and
- the set $\mathsf{Blockages}(T_{\mathrm{pref}}, X) \subseteq V(T)$.

Then, we immediately apply the following Lemma 3.4.2, provided below, to compute also

- the explored prefix $F := \mathsf{Exploration}(T_{\mathrm{pref}}, X)$ and
- the exploration graph $H := \mathsf{ExplorationGraph}(T_{\mathrm{pref}}, X)$.

Lemma 3.4.2. *Let* T_{pref} *be a prefix of* T *and* X *a* $d_{\mathcal{T}}$*-minimal* c*-small* k*-closure of* $\mathsf{bags}(T_{\mathrm{pref}})$*. Given* T_{pref}*,* X*, and* $\mathsf{Blockages}(T_{\mathrm{pref}}, X)$*, the explored prefix* F *and the exploration graph* H *can be computed in time* $k^{\mathcal{O}(1)} \cdot |F|$*.*

Proof. First, F can be computed in time $\mathcal{O}(|F|)$ by simply a depth-first search on T that stops on blockages. Then, the vertices $V(H)$ of the exploration graph can be computed in time $k^{\mathcal{O}(1)} \cdot |F|$ by taking the union of $\mathsf{bags}(F)$ and $\mathsf{Blockages}(T_{\mathrm{pref}}, X)$. Having access to $(T, \mathsf{bag}, \mathsf{edges})$, the induced subgraph $G[\mathsf{bags}(F)]$ can be computed in time $k^{\mathcal{O}(1)} \cdot |F|$, because all of its edges are stored in $\mathsf{edges}(F)$. Also, the edges between blockage vertices and vertices in $\mathsf{bags}(F)$ can be directly computed from the definition. □

The running time of this step is clearly dominated by the call to $\mathbb{D}^{\mathrm{explore}}$, and by substituting $c = \mathcal{O}(k^4)$ in the bound on the running time of Query, we conclude the following.

(RT2) The running time of Step (1) is $2^{\mathcal{O}(k^9)} \cdot |F|$.

In further analysis of the refinement data structure, the following definitions will be useful.

Definition 8. *For a collected component $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$, we define its* interface, weight, *and* height, *denoted respectively by* $\mathsf{Interface}(\mathfrak{C})$, $\mathsf{weight}(\mathfrak{C})$ *and* $\mathsf{height}(\mathfrak{C})$, *as follows.*

- *If $\mathfrak{C}$ is unblocked, then set $\mathsf{Interface}(\mathfrak{C}) := N(\mathfrak{C})$.*
- *If $\mathfrak{C}$ is blocked, then set $\mathsf{Interface}(\mathfrak{C}) := \mathsf{bag}(t) \cap X$, where $t \in \mathsf{Blockages}(T_{\mathrm{pref}}, X)$ is the only element of $\xi(\mathfrak{C})$ (i.e., t is the home bag of $\mathfrak{C}$).*

In both cases, we set

$$\mathsf{weight}(\mathfrak{C}) := |\mathsf{Interface}(\mathfrak{C})| \qquad \text{and} \qquad \mathsf{height}(\mathfrak{C}) := \mathsf{height}(t),$$

where t is the home bag of $\mathfrak{C}$.

Also, we introduce the notation mapping each set $B \subseteq V(G)$ to the set of collected components with interface B:

$$\mathsf{Interface}^{-1}(B) := \{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X) \mid \mathsf{Interface}(\mathfrak{C}) = B\}.$$

We remark that it follows from Lemma 3.3.16 that $N(\mathfrak{C}) \subseteq \mathsf{Interface}(\mathfrak{C})$.

Step (2) (Build a new prefix)

At this step, we intend to construct a prefix of a new decomposition $\mathcal{T}' = (T', \mathsf{bag}')$. As it turns out, the prefix will be a tree decomposition $\mathcal{T}^X$ of $\mathsf{torso}_G(X)$ with some additional properties. The reason for using here $\mathsf{torso}_G(X)$ is simple: For every connected component C of $G - X$, the neighborhood $N(C) \subseteq X$ is a clique in $\mathsf{torso}_G(X)$. Therefore, by Observation 2.3.1, for every tree decomposition (T^X, bag^X) of $\mathsf{torso}_G(X)$, there is a node $t \in T^X$ whose bag $\mathsf{bag}(t)$ contains $N(C)$. This will allow us to combine the tree decompositions of $\mathsf{torso}_G(X)$ with tree decompositions of the components of $G - X$.

Let us give a detailed description of the constructed prefix $\mathcal{T}^X$. We recall the following result, due to Bodlaender, that an optimum-width tree decomposition of a graph can be computed in parameterized linear time:

Theorem 3.4.3 ([Bod96, Theorem 1])**.** *Given a graph G, where $k := \mathrm{tw}(G)$, one can compute in time $2^{\mathcal{O}(k^3)} \cdot |V(G)|$ a tree decomposition (T, bag) of G of width k and with $|V(T)| = \mathcal{O}(|V(G)|)$.*

We also restate the following classical result of Bodlaender and Hagerup.

Theorem 2.3.3 ([BH98, Lemma 2.2])**.** *Given a graph G and its tree decomposition (T, bag) of width k, one can compute in time $\mathcal{O}(k \cdot |V(T)|)$ a binary tree decomposition (T', bag') of G of height $\mathcal{O}(\log |V(T)|)$, width at most $3k + 2$, and with $|V(T')| = \mathcal{O}(|V(T)|)$.*

Moreover, we will use another auxiliary operation modifying a binary tree decomposition.

Lemma 3.4.4. *Given a graph G and its binary tree decomposition (T, bag) of height h and width k', one can compute in time $\mathcal{O}(2^{k'} \cdot |V(T)|)$ a binary tree decomposition (T', bag') of G of height $\mathcal{O}(h + k')$, width k', and with $|V(T')| = \mathcal{O}(2^{k'} \cdot |V(T)|)$, such that for every node $t \in V(T)$ and every subset $B \subseteq \mathsf{bag}(t)$, there is a leaf node t_B of T' such that $\mathsf{bag}'(t_B) = B$.*

Proof. Let us iterate through all the nodes $t \in V(T)$. Let $\mathsf{bag}(t) = \{v_1, v_2, \ldots, v_p\}$, where $p \leqslant k' + 1$.

We construct a binary tree decomposition (T_t, bag_t) of the induced subgraph $\mathsf{bag}(t)$ as follows. The root of T_t is a new node p with $\mathsf{bag}_t(p) = \mathsf{bag}(t)$. The height of T_t is p. Each vertex $u \in V(T_t)$ at depth i, where $i = 0, 1, \ldots, p-1$, has two children u_{yes} and u_{no} with $\mathsf{bag}_t(u_{\mathrm{yes}}) = \mathsf{bag}_t(u)$ and $\mathsf{bag}_t(u_{\mathrm{no}}) = \mathsf{bag}_t(u) \setminus \{v_{i+1}\}$. These properties define the whole decomposition (T_t, bag_t). This is a valid tree decomposition of $\mathsf{bag}(t)$ since whenever we exclude a vertex v from a bag of $u \in V(T_t)$, then v does not appear in any of the bags of descendants of u in T_t. Moreover, observe that for every subset $B \subseteq \mathsf{bag}(t)$, there is a leaf node $t_B \in V(T)$ such that $\mathsf{bag}_t(t_B) = B$.

It remains to attach such trees T_t to the original tree T – this will form the desired decomposition (T', bag'), where $\mathsf{bag}'|_T = \mathsf{bag}$ and $\mathsf{bag}'|_{T_t} = \mathsf{bag}_t$ for every t. If a vertex t has at most one child, then we can attach T_t to T by simply making the root of T_t a child of t. If t has two children, then we can subdivide one of its edges to its children with a vertex t_{copy}, where $\mathsf{bag}'(t_{\mathrm{copy}}) = \mathsf{bag}(t)$, and apply the previous case to t_{copy}.

It is easy to see that the width of (T', bag') is still k', and the height of T' is at most $\mathcal{O}(h + k')$ as every attached tree T_t has height at most $k' + 1$. □

We are ready to define the desired decomposition $\mathcal{T}^X$ of $\mathsf{torso}_G(X)$. We perform the following operations.

1. Apply Theorem 3.4.3 to obtain a tree decomposition $\mathcal{T}^X_{\mathrm{opt}}$ of $\mathsf{torso}_G(X)$.
2. Use the algorithm from Theorem 2.3.3 on the decomposition $\mathcal{T}^X_{\mathrm{opt}}$ and call the resulting decomposition $\mathcal{T}^X_{\mathrm{shallow}}$.
3. Run the algorithm from Lemma 3.4.4 to transform the decomposition $\mathcal{T}^X_{\mathrm{shallow}}$ into the desired tree decomposition $\mathcal{T}^X = (T^X, \mathsf{bag}^X)$.

Let us list all the properties of this step that will be required in further parts of the algorithm. The purpose of property (P3) might be unclear at this point, but it will play an important role in the analysis of the amortized running time.

Lemma 3.4.5. *The following properties hold.*

(P1) $\mathcal{T}^X$ *is a binary tree decomposition of* $\mathsf{torso}_G(X)$ *of width at most* $6k+5$ *and height at most* $\mathcal{O}(k+\log N)$.

(P2) $V(T^X) \leqslant 2^{\mathcal{O}(k)} \cdot |T_{\mathrm{pref}}|$.

(P3) For every collected component $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$, *there exists a leaf* $t_{\mathfrak{C}} \in V(T^X)$ *such that* $\mathsf{bag}^X(t_{\mathfrak{C}}) = \mathsf{Interface}(\mathfrak{C})$.

(RT3) The running time of Step (2) is $2^{\mathcal{O}(k^3)} \cdot |T_{\mathrm{pref}}|$.

Proof. We prove the consecutive points of the lemma.

Claim 3.4.6. $\mathcal{T}^X$ *is a binary tree decomposition of* $\mathsf{torso}_G(X)$ *of width at most* $6k+5$ *and height* $\mathcal{O}(k+\log N)$. *Additionally,* $V(T^X) \leqslant 2^{\mathcal{O}(k)} \cdot |T_{\mathrm{pref}}|$.

Proof of the claim. Recall that since X is a k-closure, we have $\mathrm{tw}(\mathsf{torso}_G(X)) \leqslant 2k+1$. Hence, we can compute the decomposition $\mathcal{T}^X_{\mathrm{opt}}$ of width $2k+1$. Consequently, the width of the decomposition $\mathcal{T}^X_{\mathrm{shallow}}$ is bounded by $3 \cdot (2k+1) + 2 = 6k+5$. This is also the bound on the width of $\mathcal{T}^X$, as applying Lemma 3.4.4 does not increase the width of a decomposition.

Next, observe that $|X| \leqslant \mathcal{O}(k^4 \cdot |T_{\mathrm{pref}}|)$ because X is $\mathcal{O}(k^4)$-small, implying that $|V(T^X_{\mathrm{opt}})| \leqslant \mathcal{O}(k^4 \cdot |T_{\mathrm{pref}}|)$. It follows that the height of $\mathcal{T}^X_{\mathrm{shallow}}$ is bounded by $\mathcal{O}(\log |X|) = \mathcal{O}(k+\log N)$. Therefore, by Lemma 3.4.4, the height of $\mathcal{T}^X$ is indeed bounded by $\mathcal{O}(k+\log N)$.

Finally, for the size of $\mathcal{T}^X$, we have that $|V(T^X)| \leqslant 2^{\mathcal{O}(k)} \cdot |V(T^X_{\mathrm{shallow}})| \leqslant 2^{\mathcal{O}(k)} \cdot |T^X_{\mathrm{opt}}| \leqslant 2^{\mathcal{O}(k)} \cdot |\mathsf{bags}(T_{\mathrm{pref}})|$. ◁

Claim 3.4.7. *For every collected component* $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$, *there is a leaf* $t_{\mathfrak{C}} \in T^X$ *such that* $\mathsf{bag}^X(t_{\mathfrak{C}}) = \mathsf{Interface}(\mathfrak{C})$.

Proof of the claim. Consider a collected component $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$. First, observe that $\mathsf{Interface}(\mathfrak{C})$ is a clique in $\mathsf{torso}_G(X)$. Indeed, if $\mathfrak{C}$ is unblocked, then by Lemma 3.3.20, $\mathsf{Interface}(\mathfrak{C}) = N(\mathfrak{C})$ is a clique in $\mathsf{torso}_G(X)$. If $\mathfrak{C}$ is blocked, then $\mathsf{Interface}(\mathfrak{C}) = \mathsf{bag}(t) \cap X$, where t is the home bag of $\mathfrak{C}$. However, $\mathsf{bag}(t) \cap X$ is a clique in $\mathsf{torso}_G(X)$ as well, since t is a blockage.

Hence, by Observation 2.3.1, there is a node $t \in V(T^X_{\mathrm{shallow}})$ such that $\mathsf{Interface}(\mathfrak{C}) \subseteq \mathsf{bag}^X(t)$. Then, by Lemma 3.4.4, we know that there is a leaf $t_{\mathfrak{C}} \in V(T^X)$ such that $\mathsf{bag}^X(t_{\mathfrak{C}}) = \mathsf{Interface}(\mathfrak{C})$. ◁

Claim 3.4.8. *The running time of Step (2) is* $2^{\mathcal{O}(k^3)} \cdot |T_{\mathrm{pref}}|$.

Proof of the claim. According to previous observations, computing $\mathcal{T}^X_{\mathrm{opt}}$ takes $2^{\mathcal{O}(k^3)} \cdot |X|$ time, and since $|X| \leqslant k^{\mathcal{O}(1)}|T_{\mathrm{pref}}|$, we can write this bound as $2^{\mathcal{O}(k^3)} \cdot |T_{\mathrm{pref}}|$. The running time of the other two steps can be upper-bounded by $2^{\mathcal{O}(k)} \cdot |X|$, and thus the claim follows. ◁

The claims above verify all the required properties, so the proof is complete. □

Step (3) (Split the old appendices)
So far, we have constructed a tree decomposition $\mathcal{T}^X$ of $\mathsf{torso}_G(X)$. Now, we are going to construct for each collected component $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$ a tree decomposition $\mathcal{T}^{\mathfrak{C}}$ that will be attached to the decomposition $\mathcal{T}^X$. In particular, $\mathcal{T}^{\mathfrak{C}}$ will be a tree decomposition of the graph $G[\mathfrak{C} \cup \mathsf{Interface}(\mathfrak{C})]$ whose root bag contains $\mathsf{Interface}(\mathfrak{C})$. We will first describe the construction, then prove its required properties, and then argue that the relevant objects for constructing $\mathcal{T}^{\mathfrak{C}}$ for all $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$ can be computed in time $k^{\mathcal{O}(1)} \cdot |F|$.

We now describe the construction of $\mathcal{T}^{\mathfrak{C}}$. Let $\mathfrak{C}$ be a collected component, and recall the definition of home bag from Section 3.3.3. We say that a blockage $t \in \mathsf{Blockages}(T_{\mathrm{pref}}, X)$ is *associated* to $\mathfrak{C}$ if either (1) $\mathfrak{C}$ is a blocked component whose home bag t is, or (2) $\mathfrak{C}$ is an unblocked component such that $\mathfrak{C} \cap \mathsf{bag}(t)$ is nonempty. Note that the definitions of blockage and collected component imply that each blockage is associated with at most one collected component.

Now, if $\mathfrak{C}$ is a blocked component, we define $\mathcal{T}^{\mathfrak{C}} = (T^{\mathfrak{C}}, \mathsf{bag}^{\mathfrak{C}})$ so that $T^{\mathfrak{C}}$ is a copy of the subtree of T rooted at the home bag of $\mathfrak{C}$, and $\mathsf{bag}^{\mathfrak{C}}$ is similarly a copy of the bag function in this subtree. If $\mathfrak{C}$ is an unblocked component, $\mathcal{T}^{\mathfrak{C}} = (T^{\mathfrak{C}}, \mathsf{bag}^{\mathfrak{C}})$ is constructed as follows. For a blockage $b \in \mathsf{Blockages}(T_{\mathrm{pref}}, X)$ we denote by T_b the subtree of T rooted at b. We denote by $\mathsf{AB}(\mathfrak{C})$ the blockages associated with $\mathfrak{C}$. Then, the tree $T^{\mathfrak{C}}$ is defined as

$$T^{\mathfrak{C}} = T[\{t \in F \mid \mathfrak{C} \cap \mathsf{bag}(t) \neq \emptyset\} \cup \bigcup_{b \in \mathsf{AB}(\mathfrak{C})} V(T_b)].$$

That is, $T^{\mathfrak{C}}$ is the subtree of T consisting of (1) the explored nodes whose bags contain vertices in $\mathfrak{C}$ and (2) the subtrees rooted at blockages that are associated with $\mathfrak{C}$. We remark that $T^{\mathfrak{C}}$ is connected by the definition of a collected component, and also that the unique highest node in $T^{\mathfrak{C}}$ (in particular, the root of $T^{\mathfrak{C}}$ since $T^{\mathfrak{C}}$ is connected) corresponds to the home bag of $\mathfrak{C}$.

We observe that the trees $T^{\mathfrak{C}}$ across all (blocked and unblocked) collected components $\mathfrak{C}$ satisfy the following properties:

- each node in $V(T) \setminus F$ is in at most one tree $T^{\mathfrak{C}}$,
- each node in $F \setminus T_{\mathrm{pref}}$ is in at most $\ell + 1$ trees $T^{\mathfrak{C}}$ (because all nodes in $F \setminus T_{\mathrm{pref}}$ have bags of size at most $\ell + 1$), and
- no node in T_{pref} is in any tree $T^{\mathfrak{C}}$.

In the rest of this section, for a node $t \in V(T^{\mathfrak{C}})$, we denote by $\mathsf{origin}(t)$ the corresponding node in T. Note that origin is a mapping from the union $\bigcup_{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)} V(T^{\mathfrak{C}})$ to $V(T)$.

The bags of $\mathcal{T}^{\mathfrak{C}} = (T^{\mathfrak{C}}, \mathsf{bag}^{\mathfrak{C}})$ for unblocked components $\mathfrak{C}$ are defined as follows. For a node $t \in V(T^{\mathfrak{C}})$ that has $\mathsf{origin}(t) \notin F$ (i.e., $\mathsf{origin}(t) \in V(T_b)$ for some blockage b), we set $\mathsf{bag}^{\mathfrak{C}}(t) = \mathsf{bag}(\mathsf{origin}(t))$. In other words, the subtrees rooted at blockages associated with $\mathfrak{C}$ are just copied from $\mathcal{T}$ to $\mathcal{T}^{\mathfrak{C}}$ without any change. Then, for a node $t \in V(T^{\mathfrak{C}})$ with $\mathsf{origin}(t) \in F$, we define

$$\mathsf{pull}(t, \mathfrak{C}) = N(\mathfrak{C}) \cap (\mathsf{cmp}(\mathsf{origin}(t)) \setminus \mathsf{bag}(\mathsf{origin}(t))).$$

That is, $\mathsf{pull}(t, \mathfrak{C})$ consists of the vertices in $N(\mathfrak{C})$ that occur in the bags of the subtree of (T, bag) below $\mathsf{origin}(t)$ but not in $\mathsf{bag}(\mathsf{origin}(t))$. Finally, for nodes $t \in T^{\mathfrak{C}}$ with $\mathsf{origin}(t) \in F$ we define

$$\mathsf{bag}^{\mathfrak{C}}(t) = (\mathsf{bag}(\mathsf{origin}(t)) \cap N[\mathfrak{C}]) \cup \mathsf{pull}(t, \mathfrak{C}).$$

The purpose of having $\mathsf{pull}(t, \mathfrak{C})$ in $\mathsf{bag}^{\mathfrak{C}}(t)$ is to ensure that $N(\mathfrak{C})$ is in the root of $\mathcal{T}^{\mathfrak{C}}$: For every $v \in N(\mathfrak{C})$ not in the root bag of $\mathcal{T}^{\mathfrak{C}}$, we add v to every bag on the path between the root and the shallowest node of $\mathcal{T}^{\mathfrak{C}}$ containing v.

This concludes the definition of $\mathcal{T}^{\mathfrak{C}}$. Next, we prove some elementary properties of this construction.

Lemma 3.4.9. *Let $\mathfrak{C}$ be a collected component.*

(P4) $\mathcal{T}^{\mathfrak{C}}$ is a rooted binary tree decomposition of $G[\mathfrak{C} \cup \mathsf{Interface}(\mathfrak{C})]$.

(P5) The root bag of $\mathcal{T}^{\mathfrak{C}}$ contains $\mathsf{Interface}(\mathfrak{C})$ as a subset.

(P6) The height of $T^{\mathfrak{C}}$ is at most the height of the home bag of $\mathfrak{C}$.

Proof. First, when $\mathfrak{C}$ is a blocked component, let $t \in V(T)$ be the home bag of $\mathfrak{C}$. All of the properties hold directly by the facts that $\mathfrak{C} \subseteq \mathsf{cmp}(t)$, $\mathsf{Interface}(\mathfrak{C}) \subseteq \mathsf{bag}(t)$, and $\mathsf{bag}(t) \subseteq \mathfrak{C} \cup \mathsf{Interface}(\mathfrak{C})$.

Then, suppose that $\mathfrak{C}$ is unblocked. Recall that in this case, $\mathsf{Interface}(\mathfrak{C}) = N(\mathfrak{C})$. First, we have that $T^{\mathfrak{C}}$ with bags restricted to $\mathfrak{C}$ is a tree decomposition of $G[\mathfrak{C}]$ because all occurrences of the vertices in $\mathfrak{C}$ in

(T, bag) are in $T^{\mathfrak{C}}$, and we never remove occurrences of the vertices in $\mathfrak{C}$ when constructing the bags. By the same argument, $\mathcal{T}^{\mathfrak{C}}$ also covers all edges between $\mathfrak{C}$ and $N(\mathfrak{C})$. Then, recall that by Lemma 3.3.13, $\mathsf{cmp}(b) \cap X = \emptyset$ for each blockage b, implying that each $v \in N(\mathfrak{C})$ must be in some bag $\mathsf{bag}(t)$ of a node $t \in V(T^{\mathfrak{C}})$ with $\mathsf{origin}(t) \in F$. This implies that $T^{\mathfrak{C}}$ satisfies the vertex condition also for vertices $v \in N(\mathfrak{C})$, even after the insertions of the sets $\mathsf{pull}(t, \mathfrak{C})$. Note that these insertions ensure that $N(\mathfrak{C}) = \mathsf{Interface}(\mathfrak{C})$ is contained in the root of $\mathcal{T}^{\mathfrak{C}}$, which shows that $\mathcal{T}^{\mathfrak{C}}$ with bags restricted to $\mathfrak{C} \cup \mathsf{Interface}(\mathfrak{C})$ is a tree decomposition of $G[\mathfrak{C} \cup \mathsf{Interface}(\mathfrak{C})]$.

To show that the bags of $\mathcal{T}^{\mathfrak{C}}$ do not contain vertices outside of $\mathfrak{C} \cup \mathsf{Interface}(\mathfrak{C})$, observe that by the definition of an unblocked collected component, each blockage b associated with $\mathfrak{C}$ satisfies $\mathsf{bag}(b) \subseteq N[\mathfrak{C}]$ and $\mathsf{cmp}(b) \subseteq \mathfrak{C}$. Hence, the bags below the blockages, which were copied verbatim, do not contain any elements of $V(G) \setminus N[\mathfrak{C}]$; and the explored bags of $\mathcal{T}^{\mathfrak{C}}$ have been explicitly truncated to $N[\mathfrak{C}]$. This concludes the proof that $\mathcal{T}^{\mathfrak{C}}$ is a tree decomposition of $G[\mathfrak{C} \cup \mathsf{Interface}(\mathfrak{C})]$. The facts that $\mathcal{T}^{\mathfrak{C}}$ is a binary tree and that the height of $T^{\mathfrak{C}}$ is at most the height of the home bag of $\mathfrak{C}$ follow directly from the construction. □

Then, we prove that $\mathcal{T}^{\mathfrak{C}}$ has width at most ℓ. Moreover, an even stronger bound on the sizes of the bags holds, which will be crucial for bounding the potential function. This is finally a proof where we use the assumption that X is $d_{\mathcal{T}}$-minimal.

Lemma 3.4.10. *Let $t \in V(T^{\mathfrak{C}})$.*

(P7) If $\mathsf{origin}(t)$ is explored, then $|\mathsf{bag}^{\mathfrak{C}}(t)| < |\mathsf{bag}(\mathsf{origin}(t))|$ and $|\mathsf{origin}^{-1}(\mathsf{origin}(t))| \leqslant \ell + 1$.

(P8) If $\mathsf{origin}(t)$ is unexplored, then $|\mathsf{bag}^{\mathfrak{C}}(t)| = |\mathsf{bag}(\mathsf{origin}(t))|$ and $|\mathsf{origin}^{-1}(\mathsf{origin}(t))| = 1$.

(P9) It holds that $\mathsf{height}_{T^{\mathfrak{C}}}(t) \leqslant \mathsf{height}_T(\mathsf{origin}(t))$.

Proof. The height property (P9) is obvious from the construction. The property (P8) for unexplored nodes follows from the construction and the fact that each blockage is associated with at most one collected component. The property that $|\mathsf{origin}^{-1}(t)| \leqslant \ell + 1$ for explored nodes $t \in F \setminus T_{\mathrm{pref}}$ follows from the fact that each bag in $F \setminus T_{\mathrm{pref}}$ has size at most $\ell + 1$, and therefore can intersect at most $\ell + 1$ different components $\mathfrak{C}$.

It remains to prove that $|\mathsf{bag}^{\mathfrak{C}}(t)| < |\mathsf{bag}(\mathsf{origin}(t))|$ when $\mathsf{origin}(t) \in F$, for which we need to use the $d_{\mathcal{T}}$-minimality of X. Note that since $\mathsf{origin}(t) \in F$, we necessarily have that $\mathfrak{C}$ is unblocked. By the definition of $\mathsf{bag}^{\mathfrak{C}}(t)$, it suffices to prove that $|\mathsf{pull}(t, \mathfrak{C})| < |\mathsf{bag}(\mathsf{origin}(t)) \setminus N[\mathfrak{C}]|$.

First, consider the case when $\mathsf{pull}(t, \mathfrak{C}) = \emptyset$, in which case we need to prove that $\mathsf{bag}(\mathsf{origin}(t))$ is not a subset of $N[\mathfrak{C}]$. In this case, we observe that if $\mathsf{bag}(\mathsf{origin}(t))$ were be a subset of $N[\mathfrak{C}]$, then by Lemma 3.3.16, it would be a subset of $N[C]$ for $C \in \mathrm{cc}(G - X)$, in which case it would either be a component blockage if C intersects $\mathsf{bag}(\mathsf{origin}(t))$, or a subset of $N(C)$ and therefore a clique blockage if C does not intersect $\mathsf{bag}(\mathsf{origin}(t))$.

It remains to prove that if $\mathsf{pull}(t, \mathfrak{C})$ is nonempty, then $|\mathsf{pull}(t, \mathfrak{C})| < |\mathsf{bag}(\mathsf{origin}(t)) \setminus N[\mathfrak{C}]|$. For this proof, recall the definition of the function $d_{\mathcal{T}}$ in Section 2.3 as the depth function of $\mathcal{T}$. For the sake of contradiction, assume that $\mathsf{pull}(t, \mathfrak{C})$ is nonempty and $|\mathsf{pull}(t, \mathfrak{C})| \geqslant |\mathsf{bag}(\mathsf{origin}(t)) \setminus N[\mathfrak{C}]|$. Let C be a component of $G - X$ such that $N(\mathfrak{C}) = N(C)$ (its existence follows from Lemma 3.3.16). We claim that now,

$$S \coloneqq (N(C) \setminus \mathsf{pull}(t, \mathfrak{C})) \cup (\mathsf{bag}(\mathsf{origin}(t)) \setminus N[\mathfrak{C}])$$

is an $(N(C), \mathsf{bags}(T_{\mathrm{pref}}))$-separator that contradicts the fact that $N(C)$ is $d_{\mathcal{T}}$-linked into $\mathsf{bags}(T_{\mathrm{pref}})$, which by Lemma 3.3.7 contradicts that X is $d_{\mathcal{T}}$-minimal. First, note that because $\mathsf{pull}(t, \mathfrak{C}) \subseteq N(C)$, we indeed have that $|S| \leqslant |N(C)|$. Moreover, because for each $v \in \mathsf{pull}(t, \mathfrak{C})$ the highest bag containing v is a strict descendant of $\mathsf{origin}(t)$, we have for all $v \in \mathsf{pull}(t, \mathfrak{C})$ and $u \in \mathsf{bag}(\mathsf{origin}(t))$ that $d_{\mathcal{T}}(v) > d_{\mathcal{T}}(u)$, implying that $d_{\mathcal{T}}(S) < d_{\mathcal{T}}(N(C))$. It remains to prove that S indeed separates $N(C)$ from $\mathsf{bags}(T_{\mathrm{pref}})$. For this, it suffices to prove that it separates $\mathsf{pull}(t, \mathfrak{C})$ from $\mathsf{bags}(T_{\mathrm{pref}})$, because $N(C) \setminus S = \mathsf{pull}(t, \mathfrak{C})$. Suppose P is a shortest path from $\mathsf{pull}(t, \mathfrak{C})$ to $\mathsf{bags}(T_{\mathrm{pref}})$ in $G - S$. Because $\mathfrak{C}$ is disjoint from $\mathsf{bags}(T_{\mathrm{pref}})$ and $N(\mathfrak{C}) \setminus S = \mathsf{pull}(t, \mathfrak{C})$, we have that P intersects $N[\mathfrak{C}]$ only on its first vertex. However, observe that because the nodes of $\mathcal{T}$ whose bags contain vertices from $\mathsf{pull}(t, \mathfrak{C})$ are strict descendants of $\mathsf{origin}(t)$, and $\mathsf{origin}(t)$ is a descendant of an appendix of T_{pref}, it holds that $\mathsf{bag}(\mathsf{origin}(t))$ separates $\mathsf{pull}(t, \mathfrak{C})$ from $\mathsf{bags}(T_{\mathrm{pref}})$, and therefore P must intersect $\mathsf{bag}(\mathsf{origin}(t))$. However, as $\mathsf{bag}(\mathsf{origin}(t))$ is disjoint from $\mathsf{pull}(t, \mathfrak{C})$, the intersection of P and $\mathsf{bag}(\mathsf{origin}(t))$ must be in $\mathsf{bag}(\mathsf{origin}(t)) \setminus N[\mathfrak{C}]$, but $\mathsf{bag}(\mathsf{origin}(t)) \setminus N[\mathfrak{C}] \subseteq S$, so no such path P can exist, implying that S is an $(N(C), \mathsf{bags}(T_{\mathrm{pref}}))$-separator. □

As the next step, we show that constructing the decompositions $\mathcal{T}^{\mathfrak{C}}$ can be done efficiently. Note that this construction via a prefix-rebuilding update amounts to giving explicit constructions of $(T^{\mathfrak{C}}, \mathsf{bag}^{\mathfrak{C}})$ for the subtrees consisting of nodes in $\mathsf{origin}^{-1}(F)$, and then pointers how the subtrees rooted at blockages should be attached to such nodes. In particular, slightly abusing notation, let us denote by $\mathcal{T}^{\mathfrak{C}}|_F$ the restriction of $\mathcal{T}^{\mathfrak{C}}$ to nodes t with $\mathsf{origin}(t) \in F$.

Lemma 3.4.11. *There is an algorithm that given T_{pref}, X, F, H, and $\mathsf{Blockages}(T_{\mathrm{pref}}, X)$, computes a list of length $|\mathsf{Coll}(T_{\mathrm{pref}}, X)|$ of tuples: For every collected component $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$, the list contains a tuple consisting of $\mathsf{Interface}(\mathfrak{C})$, $\mathsf{height}(T^{\mathfrak{C}})$, and in addition,*

- *if $\mathfrak{C}$ is unblocked, the restriction $\mathcal{T}^{\mathfrak{C}}|_F$ of $\mathcal{T}^{\mathfrak{C}}$ to the nodes in $\mathsf{origin}^{-1}(F)$ and the mapping from blockages associated with $\mathfrak{C}$ to the leaves of $\mathcal{T}^{\mathfrak{C}}|_F$ to which they should be attached, and*
- *if $\mathfrak{C}$ is blocked, a pointer to the home bag of $\mathfrak{C}$.*

(RT4) The running time is $k^{\mathcal{O}(1)} \cdot |F|$.

Proof. First, the blocked components $\mathfrak{C}$ correspond to isolated blockage vertices t of H that have $\mathsf{cmp}(t) \neq \emptyset$. Such blockage vertices can be recognized by using the $\mathsf{Cmpsize}$ method of $\mathbb{D}^{\mathrm{aux}}$. We can use the $\mathsf{Cmpsize}$ method of $\mathbb{D}^{\mathrm{aux}}$ to test if $\mathsf{cmp}(t) \neq \emptyset$, and therefore recognize the blockages associated to blocked components and output them. Their height $\mathsf{height}(T^{\mathfrak{C}})$ can be obtained from $\mathbb{D}^{\mathrm{aux}}$, and for them $\mathsf{Interface}(\mathfrak{C}) = \mathsf{bag}(t) \cap X$.

Then, for constructing the objects for unblocked components, we first do precomputation step that computes for every node $t \in F \setminus T_{\mathrm{pref}}$ the set $X \cap \mathsf{cmp}(t)$. These sets can be computed in total time $k^{\mathcal{O}(1)} \cdot |F|$ in a bottom-up manner, because of the facts that (1) $|X \cap \mathsf{cmp}(t)| \leqslant \mathcal{O}(k^4)$ by c-smallness of X, and (2) by Lemma 3.3.13, for all $t \in \mathsf{Blockages}(T_{\mathrm{pref}}, X)$, it holds that $X \cap \mathsf{cmp}(t) = \emptyset$. We also precompute for every vertex $v \in \mathsf{bags}(F \setminus T_{\mathrm{pref}})$ a pointer to some bag in $F \setminus T_{\mathrm{pref}}$ that contains v.

We then iterate over the connected components $C \in \mathrm{cc}(H - X)$ of the exploration graph with $C \cap \mathsf{bags}(F) \neq \emptyset$, that is, corresponding to unblocked collected components. Let us now fix a component $C \in \mathrm{cc}(H - X)$. Note that C uniquely identifies an unblocked collected component $\xi^{-1}(C) = \mathfrak{C}$ and note that $N_H(C) = N(\mathfrak{C})$. Now, the nodes in $\mathcal{T}^{\mathfrak{C}}|_F$ can be identified as the set $\{t \in F \mid \mathsf{bag}(t) \cap C \neq \emptyset\}$, and they can be computed in time $k^{\mathcal{O}(1)} \cdot |\mathcal{T}^{\mathfrak{C}}|_F|$ by using the previously computed pointers, because they correspond to a connected subtree of T. The set $N(\mathfrak{C})$ can be computed in time $k^{\mathcal{O}(1)} \cdot |C|$ because $|N(\mathfrak{C})| \leqslant \mathcal{O}(k)$. Then, the sets $\mathsf{pull}(t, \mathfrak{C})$ for all $t \in \mathcal{T}^{\mathfrak{C}}|_F$ can be computed in time $k^{\mathcal{O}(1)} \cdot |\mathcal{T}^{\mathfrak{C}}|_F|$ by using the precomputed sets $X \cap \mathsf{cmp}(t)$ as we have $\mathsf{pull}(t, \mathfrak{C}) = N(\mathfrak{C}) \cap (X \cap \mathsf{cmp}(\mathsf{origin}(t))) \setminus \mathsf{bag}(\mathsf{origin}(t))$. The sets $\mathsf{bag}(\mathsf{origin}(t)) \cap N[\mathfrak{C}]$ can be trivially computed in time $k^{\mathcal{O}(1)} \cdot |\mathcal{T}^{\mathfrak{C}}|_F|$. Hence, all bags $\mathsf{bag}^{\mathfrak{C}}(t)$ can also be computed in time $k^{\mathcal{O}(1)} \cdot |\mathcal{T}^{\mathfrak{C}}|_F|$. This concludes the construction of $\mathcal{T}^{\mathfrak{C}}|_F$. Then, the pointers from the blockages associated with $\mathfrak{C}$ to the leaves of $\mathcal{T}^{\mathfrak{C}}|_F$ can be computed in time $k^{\mathcal{O}(1)} \cdot |F|$, because these blockages are exactly $C \cap \mathsf{Blockages}(T_{\mathrm{pref}}, X)$, and their parents are in $\mathcal{T}^{\mathfrak{C}}|_F$. The height $\mathsf{height}(T^{\mathfrak{C}})$ can be computed in time $\mathcal{O}(|\mathcal{T}^{\mathfrak{C}}|_F|)$ by using $\mathbb{D}^{\mathrm{aux}}$ for computing the heights of the blockages associated with $\mathfrak{C}$ and induction.

Summing up, for each collected component $\mathfrak{C}$ we spend time $k^{\mathcal{O}(1)} \cdot (|C| + |V(T^{\mathfrak{C}}|_F)|)$ constructing $\mathcal{T}^{\mathfrak{C}}|_F$ and the pointers. The sizes of C sum up to at most $|V(H)| \leqslant \mathcal{O}(k \cdot |F|)$, and the sizes of $V(T^{\mathfrak{C}}|_F)$ sum up to at most $\mathcal{O}(k \cdot |F|)$, so the total running time is $k^{\mathcal{O}(1)} \cdot |F|$. □

Step (4) (Append auxiliary subtrees)

At this point, we have constructed a tree decomposition $\mathcal{T}^X$ of $\mathsf{torso}_G(X)$ and tree decompositions $\mathcal{T}^{\mathfrak{C}}$ of $G[\mathfrak{C} \cup \mathsf{Interface}(\mathfrak{C})]$ for every collected component $\mathfrak{C}$. We know that the root of $T^{\mathfrak{C}}$ contains $\mathsf{Interface}(\mathfrak{C})$ (property (P5)) and that there is a leaf node $t_{\mathfrak{C}} \in V(T^X)$ such that $\mathsf{bag}^X(t_{\mathfrak{C}}) = \mathsf{Interface}(\mathfrak{C})$ (property (P3)). Hence, a natural idea is to create the final decomposition $\mathcal{T}'$ by connecting the root of each tree $T^{\mathfrak{C}}$ with the corresponding vertex $t_{\mathfrak{C}} \in V(T^X)$ by an edge.

Unfortunately, many collected components can have the same interface $B \subseteq X$, and connecting their roots with the same vertex of T^X would break the invariant that the data structure maintains a binary tree decomposition. It is tempting to work around this problem by building a sufficiently large complete binary tree T^{bin} rooted at $t_{\mathfrak{C}}$ and containing $\mathsf{Interface}(\mathfrak{C})$ in each of its bags. Then, we could append each tree $T^{\mathfrak{C}}$ to a different leaf of T^{bin}.

However, this approach fails in a subtle way: The construction of T^{bin} increases the potential value of the resulting tree decomposition; perhaps quite significantly if a lot of collected components share the same interface. Hence, we must ensure that the amortized cost of the construction of T^{bin} is upper-bounded by the decrease in the potential value resulting from the creation of the collected components. Roughly

speaking, this can only be achieved when the sum of the heights of the vertices of T^{bin} in the final tree decomposition is at most proportional to the sum of heights of the trees $T^{\mathfrak{C}}$. Complete binary trees unfortunately do not always satisfy this condition.

However, we will fix this using an idea resembling Huffman codes as follows. Intuitively, the tree T^{bin} should be skewed so as to satisfy the following property: Given two tree decompositions $T^{\mathfrak{C}_1}, T^{\mathfrak{C}_2}$ of collected components $\mathfrak{C}_1$ and $\mathfrak{C}_2$ with the same interface, if $T^{\mathfrak{C}_1}$ has much larger height than $T^{\mathfrak{C}_2}$, then $T^{\mathfrak{C}_1}$ should be attached to a leaf of T^{bin} that is closer to the root of T^{bin} than the leaf to which $T^{\mathfrak{C}_2}$ is attached. The following lemma formalizes this intuition.

Lemma 3.4.12. *Let $h_1, \ldots, h_m$ be positive integers and let $Q \coloneqq h_1 + \ldots + h_m$. Then, there exists a rooted binary tree T of height $\mathcal{O}(\log Q)$ and leaves labeled $h_1, \ldots, h_m$ in some order such that $\sum_{v \in V(T)} \mathsf{lheight}(v) \leqslant 26Q$, where $\mathsf{lheight}$ is defined as follows:*

$$\mathsf{lheight}(v) \coloneqq \begin{cases} \mathsf{label}(v) & \text{if } v \text{ is a leaf,} \\ 1 + \max\{\mathsf{lheight}(c) \mid c \text{ is a child of } v\} & \text{otherwise.} \end{cases}$$

Moreover, such a tree can be computed in time $\mathcal{O}(m + \log Q)$.

Proof. Let $C \colon \mathbb{Z}_{\geqslant 1} \to \mathbb{Z}_{\geqslant 0}$ be the function given by the formula $C(a) \coloneqq \lceil \log_2 a \rceil$. In other words, $C(a)$ is the smallest nonnegative integer such that $2^{C(a)} \geqslant a$. In particular, we have $2^{C(a)} < 2a$.

Partition $h_1, \ldots, h_m$ into groups $G_0, \ldots, G_{C(Q)}$ as follows: We put h_i into G_j if $h_i \in (\frac{Q}{2^{j+1}}, \frac{Q}{2^j}]$. This is a well-defined partition as these intervals for $j = 0, \ldots, C(Q)$ are disjoint and their union covers the interval $[1, Q]$.

Now, for each nonempty group G_i let us create a rooted binary tree T_i as follows. Let the elements of G_i be $g_1, \ldots, g_a$. Then, let T_i be an arbitrary rooted binary tree with a leaves labeled $g_1, \ldots, g_a$, where all leaves are at distance at most $C(a)$ from the root. As $2^{C(a)} \geqslant a$, such a tree exists. If G_i is empty, we assume that T_i is empty as well.

As the next step create a path $P = v_0, \ldots, v_{C(Q)}$ rooted at v_0. For each $i = 0, \ldots, C(Q)$, if G_i is nonempty, assign the root of T_i as a child of v_i. Remove a suffix of P that has no subtrees attached to it. This completes the description of desired T. One can readily see that the construction can be computed in time $\mathcal{O}(m + \log Q)$.

What remains is to prove the required properties of T. As each T_i has height $\mathcal{O}(\log Q)$ and the path P has length $\mathcal{O}(\log Q)$, it is clear that the height of T is $\mathcal{O}(\log Q)$ as well.

Next, we will prove a bound on the sum of $\mathsf{lheight}(v)$ for v in a particular T_i. Let the elements of G_i be $g_1, \ldots, g_a \in (\frac{Q}{2^{i+1}}, \frac{Q}{2^i}]$. Let us group the vertices $v \in V(T_i)$ by their distance j to the farthest leaf in the subtree of T_i rooted at v. For $j = 0$, the group comprises the leaves of T_i. The leaves are labeled by G_i, so their values of $\mathsf{lheight}$ do not exceed $\frac{Q}{2^i}$. Next, the vertices at distance $j \geqslant 1$ from the farthest leaf have the value of $\mathsf{lheight}$ at most $\frac{Q}{2^i} + j$ (which follows from the inspection of the definition of $\mathsf{lheight}$) and there are at most $2^{C(a)-j}$ of them (which follows from the fact that each such vertex is at depth at most $C(a) - j$ in T_i). Hence, we have the bound

$$\sum_{v \in V(T_i)} \mathsf{lheight}(v) \leqslant \sum_{j=0}^{C(a)} 2^{C(a)-j} \left(\frac{Q}{2^i} + j \right) = 2^{C(a)} \left(\frac{Q}{2^i} \sum_{j=0}^{C(a)} 2^{-j} + \sum_{j=0}^{C(a)} j \cdot 2^{-j} \right) \leqslant$$
$$\leqslant 2^{C(a)} \left(\frac{Q}{2^i} \sum_{j=0}^{\infty} 2^{-j} + \sum_{j=0}^{\infty} j \cdot 2^{-j} \right) = 2^{C(a)} \left(2 \cdot \frac{Q}{2^i} + 2 \right).$$

As additionally $2^{C(a)} \cdot 2 \leqslant 4a$ and $2^{C(a)} \cdot 2 \cdot \frac{Q}{2^i} \leqslant 8a \cdot \frac{Q}{2^{i+1}} < 8(g_1 + \ldots + g_a)$, we find that

$$\sum_{v \in V(T_i)} \mathsf{lheight}(v) \leqslant 8(g_1 + \ldots + g_a) + 4a \leqslant 12(g_1 + \ldots + g_a). \tag{3.2}$$

Summing Eq. (3.2) over all $i = 0, 1, \ldots, C(Q)$, we get that

$$\sum_{i=0}^{C(Q)} \sum_{v \in V(T_i)} \mathsf{lheight}(v) \leqslant 12Q.$$

What remains is to bound the sum of $\mathsf{lheight}$ for vertices of P. Define

$$l_i := \frac{Q}{2^i} + 2C(Q) - i + 4.$$

We now prove that $\mathsf{lheight}(v_i) \leqslant l_i$ by induction on $i = C(Q), C(Q)-1, \ldots, 0$. Each vertex v_i has at most two children:

- the root r_i of T_i if G_i is nonempty: since $|G_i| \leqslant 2^{i+1}$, we get that $\mathsf{lheight}(r_i) \leqslant \frac{Q}{2^i} + (i+1) \leqslant l_i - 1$;
- v_{i+1} if v_{i+1} exists: we have that $\mathsf{lheight}(v_{i+1}) \leqslant l_{i+1} \leqslant l_i - 1$.

Thus indeed $\mathsf{lheight}(v_i) \leqslant l_i$. Therefore,

$$\sum_{v \in P} \mathsf{lheight}(v) \leqslant \sum_{i=0}^{C(Q)} l_i \leqslant \sum_{i=0}^{C(Q)} \frac{Q}{2^i} + (C(Q)+1)(2C(Q)+4) \leqslant 2Q + (C(Q)+1)(2C(Q)+4).$$

Now, since $C(Q) < \log_2 Q + 1$, we find that $(C(Q)+1)(2C(Q)+4) < (\log_2 Q + 2)(2\log_2 Q + 6)$. Using the standard tools of the real analysis, it can be shown that for all $Q \in \mathbb{Z}_{\geqslant 1}$ we have that

$$(\log_2 Q + 2)(2\log_2 Q + 6) \leqslant 12Q.$$

Hence,

$$\sum_{v \in P} \mathsf{lheight}(v) \leqslant 14Q.$$

We conclude that

$$\sum_{v \in V(T)} \mathsf{lheight}(v) = \sum_{i=0}^{C(Q)} \sum_{v \in V(T_i)} \mathsf{lheight}(v) + \sum_{v \in P} \mathsf{lheight}(v) \leqslant 12Q + 14Q = 26Q. \qquad \square$$

Having proved Lemma 3.4.12, we can define the tree decompositions that will be attached to the already constructed decomposition $\mathcal{T}^X$. Let $\mathcal{I}(T_{\mathrm{pref}}, X) := \{\mathsf{Interface}(\mathfrak{C}) \mid \mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)\}$ denote the set of all interfaces of collected components. After obtaining the tuples representing the collected components from Lemma 3.4.11, we group these tuples by their interfaces, that is, we iterate over all sets $B \in \mathcal{I}(T_{\mathrm{pref}}, X)$ and list all components $\mathfrak{C}_1, \mathfrak{C}_2, \ldots, \mathfrak{C}_m \in \mathsf{Interface}^{-1}(B)$ with interface B. This can be implemented in $|\mathsf{Coll}(T_{\mathrm{pref}}, X)| \cdot k^{\mathcal{O}(1)}$ time by standard arguments using bucket sorting.

Now, let us fix $B \in \mathcal{I}(T_{\mathrm{pref}}, X)$, consider components $\mathfrak{C}_1, \mathfrak{C}_2, \ldots, \mathfrak{C}_m \in \mathsf{Interface}^{-1}(B)$, and let $h_i = \mathsf{height}(T^{\mathfrak{C}_i}) + 1$, for $i \in \{1, \ldots, m\}$. We run the algorithm from Lemma 3.4.12 for integers $h_1, \ldots, h_m$ to obtain a tree T^B_{pref} with leaves labeled with integers h_i. Then, our construction is to attach the trees $T^{\mathfrak{C}_i}$ to T^B_{pref}, for $i = 1, \ldots, m$, by appending the root of $T^{\mathfrak{C}_i}$ as a child of the leaf of T^B_{pref} labeled with h_i, so that each leaf of T^B_{pref} has exactly one child. Let T^B be the obtained binary tree, and observe that for each node t_i of T^B that is a leaf of T^B_{pref} to which $T^{\mathfrak{C}_i}$ was attached, $\mathsf{height}_{T^B}(t_i) = h_i$.

To define the tree decomposition $\mathcal{T}^B = (T^B, \mathsf{bag}^B)$ it remains to define the function bag^B. We set $\mathsf{bag}^B|_{V(T^{\mathfrak{C}_i})} = \mathsf{bag}^{\mathfrak{C}_i}$, and $\mathsf{bag}^B(t) = B$ for every $t \in V(T^B_{\mathrm{pref}})$.

Now, let us analyze this procedure. First, for each appendix t of T_{pref}, we show that the number of all possible interfaces of all collected components contained in $\mathsf{cmp}(t)$ is bounded by a number depending only on k. Note that this bound is precisely the reason why we require the closure X to be c-small.

Lemma 3.4.13. *For every appendix $t \in \mathsf{App}(T_{\mathrm{pref}})$, we have that*

$$|\{\mathsf{Interface}(\mathfrak{C}) \mid \mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X),\ \mathfrak{C} \subseteq \mathsf{cmp}(t)\}| \leqslant k^{\mathcal{O}(k)}.$$

Proof. Let $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$ be such that $\mathfrak{C} \subseteq \mathsf{cmp}(t)$. Recall that $\mathsf{Interface}(\mathfrak{C}) \subseteq X$, implying that $\mathsf{Interface}(\mathfrak{C}) \subseteq X \cap (\mathsf{cmp}(t) \cup \mathsf{adh}(t))$. Since X is c-small, we have $|X \cap \mathsf{cmp}(t)| \leqslant c \in \mathcal{O}(k^4)$. Moreover, we have that $|X \cap \mathsf{adh}(t)| \leqslant |\mathsf{bag}(t)| \leqslant \ell + 1$. As $|\mathsf{Interface}(\mathfrak{C})| \leqslant 2k+2$ ($\mathsf{Interface}(\mathfrak{C})$ is a clique in $\mathsf{torso}_G(X)$ and $\mathrm{tw}(\mathsf{torso}_G(X)) \leqslant 2k+1$), there are at most $\sum_{i=0}^{2k+2} \binom{c+\ell+1}{i} = (k^4)^{\mathcal{O}(k)} = k^{\mathcal{O}(k)}$ possible values for $\mathsf{Interface}(\mathfrak{C})$ such that $\mathfrak{C} \subseteq \mathsf{cmp}(t)$. $\square$

We immediately derive a bound on the number of all interfaces of all collected components in the graph:

Corollary 3.4.14. *The following inequality holds:*

$$|\mathcal{I}(T_{\mathrm{pref}}, X)| \leqslant k^{\mathcal{O}(k)} \cdot |T_{\mathrm{pref}}|.$$

Proof. For each collected component $\mathfrak{C}$ there is a unique appendix t such that $\mathfrak{C} \subseteq \mathsf{cmp}(t)$. Since (T, bag) is a binary tree decomposition, we know that $|\mathsf{App}(T_{\mathrm{pref}})| \leqslant |T_{\mathrm{pref}}| + 1$. Consequently, $|\mathcal{I}(T_{\mathrm{pref}}, X)| \leqslant k^{\mathcal{O}(k)} \cdot |T_{\mathrm{pref}}|$, as desired. □

Let us summarize the properties of the construction from this step of the refinement operation.

Lemma 3.4.15. *Let $B \in \mathcal{I}(T_{\mathrm{pref}}, X)$.*

(P10) *$\mathcal{T}^B = (T^B, \mathsf{bag}^B)$ is a binary tree decomposition of $G[B \cup \bigcup_{\mathfrak{C} \in \mathsf{Interface}^{-1}(B)} \mathfrak{C}]$ and has width at most ℓ.*

(P11) *The height of T^B is at most*

$$\mathcal{O}(\log N) + \max_{\mathfrak{C} \in \mathsf{Interface}^{-1}(B)} \mathsf{height}(\mathcal{T}^{\mathfrak{C}}).$$

(P12) *The root bag of $\mathcal{T}^B$ contains exactly the set B.*

(P13) *The sum of the heights of the nodes in $V(T^B_{\mathrm{pref}})$ is bounded as follows:*

$$\sum_{t \in V(T^B_{\mathrm{pref}})} \mathsf{height}_{T^B}(t) \leqslant 52 \cdot \sum_{\mathfrak{C} \in \mathsf{Interface}^{-1}(B)} \mathsf{height}(T^{\mathfrak{C}}).$$

Proof. First, we prove that $\mathcal{T}^B$ is indeed a valid tree decomposition. $\mathcal{T}^B$ is a concatenation of valid tree decompositions $\mathcal{T}^{\mathfrak{C}}$ with a prefix T^B_{pref} (which is, in fact, a valid tree decomposition of $G[B]$). Recall that all collected components are vertex-disjoint. Also, the edge condition is satisfied: For an edge uv with $u, v \in B$ the edge condition is trivial, and if some endpoint of an edge uv, say v, belongs to a collected component $\mathfrak{C}$, then $u \in N[\mathfrak{C}] \subseteq \mathfrak{C} \cup \mathsf{Interface}(\mathfrak{C})$. Since $\mathcal{T}^{\mathfrak{C}}$ is a tree decomposition of $G[\mathfrak{C} \cup \mathsf{Interface}(\mathfrak{C})]$ (property (P4)) we infer that u and v are together in some bag of $\mathcal{T}^{\mathfrak{C}}$. Hence, it remains to argue that every vertex $v \in B$ appears in a connected subset of nodes of $\mathcal{T}^B$. This follows immediately from the fact that T^B_{pref} contains B in all of the bags, and for every collected component $\mathfrak{C}$, the root bag of $\mathcal{T}^{\mathfrak{C}}$ contains $\mathsf{Interface}(\mathfrak{C}) = B$ as a subset (property (P5)).

By Lemma 3.4.12, the height of T^B_{pref} is at most $\mathcal{O}(\log \sum_{\mathfrak{C} \in \mathsf{Interface}^{-1}(B)} (\mathsf{height}(T^{\mathfrak{C}}) + 1))$, which is at most $\mathcal{O}(\log(N^2)) = \mathcal{O}(\log N)$ since $\mathsf{height}(T^{\mathfrak{C}}) \leqslant N$ and $|\mathsf{Interface}^{-1}(B)| \leqslant N$. It follows from the construction that the height of T^B is at most $\mathcal{O}(\log N) + \max_{\mathfrak{C} \in \mathsf{Interface}^{-1}(B)} \mathsf{height}(T^{\mathfrak{C}})$.

Since T^B_{pref} is a nonempty tree, we have that its root bag (which is the root bag of $\mathcal{T}^B$ as well) contains exactly the set B.

For bounding the sum of the heights of the nodes in $V(T^B_{\mathrm{pref}})$, we observe that the height $\mathsf{height}_{T^B}(t_i)$ of a node $t_i \in V(T^B_{\mathrm{pref}})$ that is a leaf in T^B_{pref} to which $\mathcal{T}^{\mathfrak{C}_i}$ was attached is exactly $\mathsf{height}(T^{\mathfrak{C}_i}) + 1 = h_i$, and therefore the heights in $V(T^B_{\mathrm{pref}})$ correspond exactly to the $\mathsf{lheight}$ in the statement of Lemma 3.4.12. Therefore, by Lemma 3.4.12 we get that

$$\sum_{t \in V(T^B_{\mathrm{pref}})} \mathsf{height}_{T^B}(t) \leqslant 26 \cdot \sum_{\mathfrak{C} \in \mathsf{Interface}^{-1}(B)} (\mathsf{height}(T^{\mathfrak{C}}) + 1) \leqslant 52 \cdot \sum_{\mathfrak{C} \in \mathsf{Interface}^{-1}(B)} \mathsf{height}(T^{\mathfrak{C}}).$$

□

Finally, we bound the running time of this step.

Lemma 3.4.16. *There is an algorithm that given the output of Lemma 3.4.11, computes the set $\mathcal{I}(T_{\mathrm{pref}}, X)$ and for each $B \in \mathcal{I}(T_{\mathrm{pref}}, X)$ the tree T^B_{pref} and pointers from the representations of collected components $\mathfrak{C} \in \mathsf{Interface}^{-1}(B)$ to the leaves of T^B_{pref} to which the tree decompositions $\mathcal{T}^{\mathfrak{C}}$ are attached in the construction.*

(RT5) *The running time is $|F| \cdot k^{\mathcal{O}(1)} + |T_{\mathrm{pref}}| \cdot k^{\mathcal{O}(k)} \cdot \log N$.*

Proof. Note that the output of Lemma 3.4.11 has size at most $|F| \cdot k^{\mathcal{O}(1)}$. We first group the interfaces $B = \mathsf{Interface}(\mathfrak{C})$ by using bucket sorting, which takes $\mathsf{Coll}(T_{\mathrm{pref}}, X) \cdot k^{\mathcal{O}(1)} = k^{\mathcal{O}(1)} \cdot |F|$ time. Then, the remaining running time is clearly dominated by the total running time of the calls to Lemma 3.4.12. A single application of this lemma for a set $B \in \mathcal{I}(T_{\mathrm{pref}}, X)$ takes time

$$\mathcal{O}(|\mathsf{Interface}^{-1}(B)| + \log N)$$

as the sum of heights of all collected components can be bounded $\mathcal{O}(N^2)$.

Hence, as the sum of $|\mathsf{Interface}^{-1}(B)|$ over all B can be bounded by $|F|k^{\mathcal{O}(1)}$ simply by the size of the output of Lemma 3.4.11, and by $|T_{\mathrm{pref}}| \cdot k^{\mathcal{O}(k)}$ by Corollary 3.4.14, the total running time can be bounded by $|F| \cdot k^{\mathcal{O}(1)} + |T_{\mathrm{pref}}| \cdot k^{\mathcal{O}(k)} \cdot \log N$. □

Step (5) (Join the pieces of the decomposition together)
It is time to define the final decomposition $\mathcal{T}' = (T', \mathsf{bag}')$. In the previous step, we have defined, for every $B \in \mathcal{I}(T_{\mathrm{pref}}, X)$, a binary tree decomposition $\mathcal{T}^B$ of all collected components $\mathfrak{C}$ such that $\mathsf{Interface}(\mathfrak{C}) = B$. Moreover, the root bag of $\mathcal{T}^B$ contains exactly the set B (property (P12)) and in the tree decomposition $\mathcal{T}^X$ constructed in Step (2) there exists a leaf bag $t_B \in V(T^X)$ such that $\mathsf{bag}^X(t_B) = B$ (property (P3)).

Hence, to construct T' it is enough to connect via an edge, for every $B \in \mathcal{I}(T_{\mathrm{pref}}, X)$, the root of T^B with the corresponding vertex $t_B \in V(T^X)$. The function bag' is just the union of the functions bag^X and bag^B, for $B \in \mathcal{I}(T_{\mathrm{pref}}, X)$. Clearly, given $\mathcal{T}^X$ and the outputs of Lemmas 3.4.11 and 3.4.16, we can in time $|F| \cdot 2^{\mathcal{O}(k)}$ construct a description of a prefix-rebuilding update of size bounded by $|F| \cdot 2^{\mathcal{O}(k)}$ that turns $\mathcal{T}$ into $\mathcal{T}'$.

(RT6) The running time of Step (5) is $|F| \cdot 2^{\mathcal{O}(k)}$.

At this point, we should prove the correctness of the given procedure. The analysis of the amortized running time together with the exact formula for the potential function Φ will be given in the next section.

Lemma 3.4.17. *(T', bag') is a valid tree decomposition of G of width at most ℓ.*

Proof. First, recall that (T', bag') was obtained by gluing the tree decompositions $\mathcal{T}^X$ and $\mathcal{T}^B$, for every $B \in \mathcal{I}(T_{\mathrm{pref}}, X)$. All of these decompositions are of width at most ℓ (properties (P1) and (P10)), and thus (T', bag') is of width at most ℓ as well.

It remains to prove that (T', bag') is indeed a valid tree decomposition of G. According to the definition of tree decomposition, we need to verify two facts.

Claim 3.4.18. *For each vertex $v \in V(G)$, the subset of nodes $\{t \in V(T') \mid v \in \mathsf{bag}'(t)\}$ induces a nonempty connected subtree of T'.*

Proof of the claim. Consider a vertex $v \in V(G)$.

- If $v \notin X$, then there is a unique collected component $\mathfrak{C}$ such that $v \in \mathfrak{C}$. Since $\mathcal{T}^{\mathfrak{C}}$ is a valid tree decomposition of $G[\mathfrak{C} \cup \mathsf{Interface}(\mathfrak{C})]$, the subset of nodes of $T^{\mathfrak{C}}$ containing v must induce a connected subtree of $T^{\mathfrak{C}}$. Moreover, from the construction of $\mathcal{T}'$, we conclude that v cannot appear in any bag of $\mathcal{T}'$ outside of $T^{\mathfrak{C}}$.
- Now, assume that $v \in X$. The vertex v appears in a connected subset of nodes of $\mathcal{T}^X$ and in connected subtrees of nodes in trees $\mathcal{T}^B$ for each $B \in \mathcal{I}(T_{\mathrm{pref}}, X)$ satisfying $x \in B$. Moreover, by the construction of $\mathcal{T}'$, the trees T^B are attached to the tree T^X by connecting two bags containing precisely the set B. Hence, v must appear in a connected subset of nodes of $\mathcal{T}'$. ◁

Claim 3.4.19. *For each edge $uv \in E(G)$, there exists a node $t \in V(T')$ such that $\{u, v\} \subseteq \mathsf{bag}'(t)$.*

Proof of the claim. Consider an edge $uv \in E(G)$.

- If $\{u, v\} \subseteq X$, then $uv \subseteq E(\mathsf{torso}_G(X))$, and thus there must be a bag t in the prefix T^X containing both u and v, as $\mathcal{T}^X$ is a tree decomposition of $\mathsf{torso}_G(X)$ (property (P1)).
- If $\{u, v\} \cap X = \emptyset$, there is a unique component $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$ containing both u and v (since $uv \in E(G)$ and there are no edges between different collected components). Let $B = \mathsf{Interface}(\mathfrak{C})$. Then $\mathcal{T}$ contains the tree decomposition $\mathcal{T}^B$ as a subtree. Since $\mathcal{T}^B$ is a tree decomposition of $G[B \cup \mathsf{Interface}^{-1}(B)]$ (property (P10)), u and v are together in some bag of $\mathcal{T}^B$.

- Finally, if exactly one of the vertices u and v belongs to X, say $u \in X$, then there is a unique component $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$ containing v. Again let $B = \mathsf{Interface}(\mathfrak{C})$. By the definition of an interface, we have that $u \in B$. Again, since $\mathcal{T}^B$ is a tree decomposition of $G[B \cup \mathsf{Interface}^{-1}(B)]$ (property (P10)), u and v are together in some bag of $\mathcal{T}^B$. ◁

Claims 3.4.18 and 3.4.19 conclude the proof of the lemma. □

3.4.3 Analysis of the amortized running time

In this section, we provide the proofs of properties (RT1) and (POT) of the defined refinement operation. We preserve all notation from Section 3.4.2. Furthermore, denote by $\widetilde{F} := F \setminus T_{\mathrm{pref}}$ the set of all explored nodes excluding the nodes of T_{pref}.

By summing the running times (RT2) – (RT6) of consecutive steps of the refinement operation, we immediately obtain the following statement.

Fact 3.4.20. *The worst-case time complexity of* $\mathsf{Refine}(T_{\mathrm{pref}})$ *is upper-bounded by*

$$2^{\mathcal{O}(k^9)}(|F| + |T_{\mathrm{pref}}| \cdot \log N) \leqslant 2^{\mathcal{O}(k^9)}(|\widetilde{F}| + |T_{\mathrm{pref}}| \cdot \log N).$$

Observe that both $|T_{\mathrm{pref}}|$ and $\log N$ are expressions we can easily control during the run of the data structure. Unfortunately, the size $|\widetilde{F}|$ of the explored region can be arbitrarily large; in the worst-case the exploration F may contain all the nodes of T. Hence, to prove the desired bounds on the running time of our algorithm, we need to bind the drop of the potential function to the size of exploration $\widetilde{F}$.

Before we proceed with the analysis of the amortized running time, we need to prove one more auxiliary lemma.

Lemma 3.4.21. *There exists a mapping*

$$\mathsf{collmap} : \mathsf{Coll}(T_{\mathrm{pref}}, X) \to \widetilde{F} \cup \mathsf{App}(T_{\mathrm{pref}})$$

satisfying the following properties:

(i) for each $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$*, we have that* $\mathsf{height}(\mathfrak{C}) \leqslant \mathsf{height}_T(\mathsf{collmap}(\mathfrak{C}))$*;*

(ii) for each $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$*, if* $\mathsf{collmap}(\mathfrak{C}) \notin \mathsf{App}(T_{\mathrm{pref}})$*, then* $\mathsf{weight}(\mathfrak{C}) < |\mathsf{bag}(\mathsf{collmap}(\mathfrak{C}))|$*;*

(iii) for each $t \in \widetilde{F} \cup \mathsf{App}(T_{\mathrm{pref}})$*, we have that* $|\mathsf{collmap}^{-1}(t)| \leqslant \ell + 4$*.*

Proof. We define $\mathsf{collmap}$ as follows. Take $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$. Let t be the home bag of $\mathfrak{C}$. Then:

- If $\mathfrak{C}$ is unblocked, then set $\mathsf{collmap}(t) := t$. Recall here that t is the shallowest node of T such that $\mathsf{bag}(t)$ contains an explored vertex of $\mathfrak{C}$.
- If $\mathfrak{C}$ is blocked, then set $\mathsf{collmap}(t) := t$ if $t \in \mathsf{App}(T_{\mathrm{pref}})$; otherwise set $\mathsf{collmap}(t) := \mathsf{parent}(t)$. Recall here that t is the blockage that is the only element of $\xi(\mathfrak{C})$.

Recall that $\mathsf{height}(\mathfrak{C}) = \mathsf{height}_T(t)$ and $\mathsf{weight}(\mathfrak{C}) = |\mathsf{Interface}(\mathfrak{C})|$.

We first prove that for each $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$ we indeed have that $\mathsf{collmap}(\mathfrak{C}) \in \widetilde{F} \cup \mathsf{App}(T_{\mathrm{pref}})$. First, the case where $\mathfrak{C}$ is unblocked is immediately resolved by Lemma 3.3.21: We always have that $\mathsf{collmap}(\mathfrak{C}) \in \widetilde{F}$. On the other hand, if $\mathfrak{C}$ is blocked, then t is a blockage and not the root of T as $T_{\mathrm{pref}} \neq \emptyset$. Therefore, $\mathsf{parent}(t)$ exists and belongs to F. If $\mathsf{parent}(t) \in T_{\mathrm{pref}}$, then $\mathsf{collmap}(\mathfrak{C}) = t \in \mathsf{App}(T_{\mathrm{pref}})$. Otherwise, $\mathsf{collmap}(\mathfrak{C}) = \mathsf{parent}(t) \in \widetilde{F}$.

That property (i) is satisfied is immediate. Next we show property (ii). Consider $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$ with $\mathsf{collmap}(\mathfrak{C}) \notin \mathsf{App}(T_{\mathrm{pref}})$. Again, the case of unblocked collected components is resolved by Lemma 3.3.21. Now assume $\mathfrak{C}$ is blocked. As $\mathsf{collmap}(\mathfrak{C}) \notin \mathsf{App}(T_{\mathrm{pref}})$, we find by examining the definition of $\mathsf{collmap}$ that $\mathsf{parent}(t) \notin T_{\mathrm{pref}}$ and $\mathsf{collmap}(\mathfrak{C}) = \mathsf{parent}(t)$. But then Lemma 3.3.22 applies, yielding that $\mathsf{weight}(\mathfrak{C}) = |\mathsf{bag}(t) \cap X| < |\mathsf{bag}(\mathsf{parent}(t))| = |\mathsf{bag}(\mathsf{collmap}(\mathfrak{C}))|$.

It remains to argue property (iii). Consider a node $t \in \widetilde{F} \cup \mathsf{App}(T_{\mathrm{pref}})$. Recall that since $t \notin T_{\mathrm{pref}}$, we have $|\mathsf{bag}(t)| \leqslant \ell + 1$. Note that each unblocked component $\mathfrak{C}$ is assigned by $\mathsf{collmap}$ to some explored node t' such that $\mathsf{bag}(t')$ intersects $\mathfrak{C}$. Since collected components form a partition of $V(G) \setminus X$, at most $\ell + 1$ different collected components may intersect $\mathsf{bag}(t)$ and thus at most $\ell + 1$ different unblocked collected components may be assigned t by $\mathsf{collmap}$. Next, each blocked collected component $\mathfrak{C}$ is mapped by ξ to a different blockage $b \in \mathsf{Blockages}(T_{\mathrm{pref}}, X)$. Observe that, by the definition of $\mathsf{collmap}$, $\mathfrak{C}$ may be mapped by $\mathsf{collmap}$ to t only if b is equal to either t or a child of t. Since T is binary, we conclude that $\mathsf{collmap}$ may map at most three different blocked components to t. □

Now, we prove a slight strengthening of property (POT). Note that since ℓ is fixed, for the rest of the section we omit it in the lower index of the potential function; that is, we use $\Phi := \Phi_\ell$.

Lemma 3.4.22. *The following inequality holds:*

$$\Phi(\mathcal{T}) - \Phi(\mathcal{T}') \geqslant |\widetilde{F}| + \sum_{t \in T_{\mathrm{pref}}} \mathsf{height}_T(t) - k^{\mathcal{O}(k)} \cdot \left(|T_{\mathrm{pref}}| + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(t) \right) \cdot \log N.$$

Proof. Within the proof, for clarity we use $\Phi_{\mathcal{T}}(\cdot)$ to denote the potential function on nodes of $\mathcal{T}$ computed in $\mathcal{T}$, and $\Phi_{\mathcal{T}'}(\cdot)$ to denote the potential function on nodes of $\mathcal{T}'$ computed in $\mathcal{T}'$. Also, for a subtree S of T, denote $\Phi_{\mathcal{T}}(S) := \Phi_{\mathcal{T}}(V(S)) = \sum_{t \in V(S)} \Phi_{\mathcal{T}}(t)$, and similarly for $\Phi_{\mathcal{T}'}(\cdot)$.

For the tree decomposition $\mathcal{T}$, we consider the partition of its nodes given by: the prefix T_{pref}, the set $\widetilde{F}$ of all explored vertices excluding T_{pref}, and the set of unexplored vertices. Hence, we can write $\Phi(\mathcal{T})$ in the form:

$$\Phi(\mathcal{T}) = \Phi_{\mathcal{T}}(T_{\mathrm{pref}}) + \Phi_{\mathcal{T}}(\widetilde{F}) + \sum_{t \notin F} \Phi_{\mathcal{T}}(t).$$

Applying the inequality $\Phi_{\mathcal{T}}(t) \geqslant \mathsf{height}_T(t)$ for every $t \in T_{\mathrm{pref}}$, we get that

$$\Phi(\mathcal{T}) \geqslant \sum_{t \in T_{\mathrm{pref}}} \mathsf{height}_T(t) + \Phi_{\mathcal{T}}(\widetilde{F}) + \sum_{t \notin F} \Phi_{\mathcal{T}}(t). \tag{3.3}$$

Now, let us focus on the structure of the tree decomposition $\mathcal{T}'$. From Step (5) we know that $V(T')$ is the disjoint union of $V(T^X)$ and $V(T^B)$ for every $B \in \mathcal{I}(T_{\mathrm{pref}}, X)$. Hence,

$$\Phi(\mathcal{T}') = \Phi_{\mathcal{T}'}(T^X) + \sum_{B \in \mathcal{I}(T_{\mathrm{pref}}, X)} \Phi_{\mathcal{T}'}(T^B).$$

From Step (4), we know that every tree T^B has a prefix T^B_{pref} with attached decompositions $\mathcal{T}^{\mathfrak{C}}$ of collected components in such a way that for every component $\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)$ there is a unique tree T^B_{pref} to which $T^{\mathfrak{C}}$ is attached. Therefore,

$$\Phi(\mathcal{T}') = \Phi_{\mathcal{T}'}(T^X) + \sum_{B \in \mathcal{I}(T_{\mathrm{pref}}, X)} \Phi_{\mathcal{T}'}(T^B_{\mathrm{pref}}) + \sum_{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)} \Phi_{\mathcal{T}'}(T^{\mathfrak{C}}). \tag{3.4}$$

Now, we are going to bound the three terms on the right-hand side of Eq. (3.4) separately.

Claim 3.4.23. *The following inequality holds:*

$$\Phi_{\mathcal{T}'}(T^X) \leqslant k^{\mathcal{O}(k)} \cdot \left(|T_{\mathrm{pref}}| + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(t) \right) \cdot \log N.$$

Proof of the claim. For brevity, we introduce the following notation: For two nodes $t, t' \in V(T')$, we write $t \preccurlyeq t'$ if t is an ancestor of t' in T'.

Recall that $\mathcal{T}'$ is created by attaching the subtree T^B for each $B \in \mathcal{I}(T_{\mathrm{pref}}, X)$ to a different leaf t_B of T^X. Therefore, for each $t \in V(T^X)$, we have that

$$\mathsf{height}_{T'}(t) \leqslant \mathsf{height}(T^X) + \max\{\mathsf{height}(T^B) \mid B \in \mathcal{I}(T_{\mathrm{pref}}, X) \text{ and } t \preccurlyeq t_B\}. \tag{3.5}$$

(We assume that the maximum value of the empty set is 0.) Recall that $\mathsf{height}(T^X) \leqslant \mathcal{O}(\log N + k)$ (property (P1)) and that for each $B \in \mathcal{I}(T_{\mathrm{pref}}, X)$, we have

$$\mathsf{height}(T^B) \leqslant \mathcal{O}(\log N) + \max_{\mathfrak{C} \in \mathsf{Interface}^{-1}(B)} \mathsf{height}(T^{\mathfrak{C}})$$

(property (P11)). Now, for each $B \in \mathcal{I}(T_{\mathrm{pref}}, X)$, let us fix $\mathfrak{C}_B$ to be an arbitrary collected component $\mathfrak{C} \in \mathsf{Interface}^{-1}(B)$ maximizing $\mathsf{height}(T^{\mathfrak{C}})$.

Plugging the inequalities above into Eq. (3.5), we get

$$\mathsf{height}_{T'}(t) \leqslant \mathcal{O}(\log N + k) + \max\{\mathsf{height}(T^{\mathfrak{C}_B}) \mid B \in \mathcal{I}(T_{\mathrm{pref}}, X) \text{ and } t \preccurlyeq t_B\}.$$

Define a function $\delta : V(T^X) \to \mathbb{N}$ and a partial function $\Lambda : V(T^X) \rightharpoonup \mathsf{Coll}(T_{\mathrm{pref}}, X)$ as follows: For $t \in V(T^X)$, let $\Lambda(t)$ be a collected component $\mathfrak{C}_B$ of maximum height such that $t \preccurlyeq t_B$ (if any such component exists). Following that, set $\delta(t) := \mathsf{height}(\Lambda(t))$; or set $\delta(t) := 0$ if $\Lambda(t)$ is undefined. Note that we have that $\mathsf{height}_{T'}(t) \leqslant \mathcal{O}(\log N + k) + \delta(t)$; therefore,

$$\begin{aligned}\Phi_{\mathcal{T}'}(T^X) &= \sum_{t \in V(T^X)} g(|\mathsf{bag}'(t)|) \cdot \mathsf{height}_{T'}(t) \leqslant \sum_{t \in V(T^X)} g(\ell+1) \cdot (\mathcal{O}(\log N + k) + \delta(t)) \\ &\leqslant \sum_{t \in V(T^X)} k^{\mathcal{O}(k)} \cdot (\log N + \delta(t)) \\ &\overset{(\star)}{\leqslant} k^{\mathcal{O}(k)} \cdot |T_{\mathrm{pref}}| \cdot \log N \; + \; k^{\mathcal{O}(k)} \cdot \sum_{t \in V(T^X)} \delta(t).\end{aligned} \tag{3.6}$$

(In $(\star)$, we used the fact that $|V(T^X)| \leqslant 2^{\mathcal{O}(k)} \cdot |T_{\mathrm{pref}}|$ by property (P2).) Thus, it remains to bound $\sum_{t \in V(T^X)} \delta(t)$.

Let $B \in \mathcal{I}(T_{\mathrm{pref}}, X)$ be an interface. Since T^X is a tree of height $\mathcal{O}(\log N + k)$, there are at most $\mathcal{O}(\log N + k)$ ancestors of t_B in T^X. Hence, there are at most $\mathcal{O}(\log N + k)$ nodes $t \in V(T^X)$ for which $\Lambda(t) = \mathfrak{C}_B$. Naturally, for each such t, we have $\delta(t) = \mathsf{height}(T^{\mathfrak{C}_B})$. As such,

$$\sum_{t \in V(T^X)} \delta(t) \leqslant \sum_{B \in \mathcal{I}(T_{\mathrm{pref}}, X)} \mathcal{O}(\log N + k) \cdot \mathsf{height}(T^{\mathfrak{C}_B}). \tag{3.7}$$

For each interface B, let $a_B \in \mathsf{App}(T_{\mathrm{pref}})$ be the unique appendix of T_{pref} for which $\mathfrak{C}_B \subseteq \mathsf{cmp}(a_B)$. Now, by Lemma 3.4.13, for each appendix $t \in \mathsf{App}(T_{\mathrm{pref}})$, there are at most $k^{\mathcal{O}(k)}$ interfaces B such that $a_B = t$. For each such interface B, we have that $\mathsf{height}(T^{\mathfrak{C}_B})$ is at most the height of the home bag of $\mathfrak{C}_B$ (property (P6)) in T. Since $\mathfrak{C}_B \subseteq \mathsf{cmp}(t)$, the home bag of $\mathfrak{C}_B$ is a descendant of t. Therefore, $\mathsf{height}(T^{\mathfrak{C}_B}) \leqslant \mathsf{height}_T(t)$. Combined with Eq. (3.7), this gives us that

$$\sum_{t \in V(T^X)} \delta(t) \leqslant \mathcal{O}(\log N + k) \cdot \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} k^{\mathcal{O}(k)} \cdot \mathsf{height}_T(t) \leqslant k^{\mathcal{O}(k)} \cdot \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(t) \cdot \log N.$$

Together with Eq. (3.6), this concludes the proof. ◁

Claim 3.4.24. *The following inequality holds:*

$$\sum_{B \in \mathcal{I}(T_{\mathrm{pref}}, X)} \Phi_{\mathcal{T}'}(T^B_{\mathrm{pref}}) \leqslant 52(\ell+4) \cdot \left(\sum_{t \in \widetilde{F}} g(|\mathsf{bag}(t)| - 1) \cdot \mathsf{height}_T(t) \right) + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} k^{\mathcal{O}(k)} \cdot \mathsf{height}_T(t).$$

Proof of the claim. Fix a set $B \in \mathcal{I}(T_{\mathrm{pref}}, X)$. Recall that

$$\sum_{t \in V(T^B_{\mathrm{pref}})} \mathsf{height}_{T^B}(t) \leqslant 52 \cdot \sum_{\mathfrak{C} \in \mathsf{Interface}^{-1}(B)} \mathsf{height}(T^{\mathfrak{C}})$$

(property (P13)). Thus,

$$\begin{aligned}\Phi_{\mathcal{T}'}(T^B_{\mathrm{pref}}) &= \sum_{t \in T^B_{\mathrm{pref}}} g(|B|) \cdot \mathsf{height}_{T'}(t) = \sum_{t \in T^B_{\mathrm{pref}}} g(|B|) \cdot \mathsf{height}_{T^B}(t) \\ &\leqslant g(|B|) \cdot 52 \cdot \sum_{\mathfrak{C} \in \mathsf{Interface}^{-1}(B)} \mathsf{height}(T^{\mathfrak{C}}) \\ &\overset{(\star)}{\leqslant} 52 \cdot \sum_{\mathfrak{C} \in \mathsf{Interface}^{-1}(B)} g(\mathsf{weight}(\mathfrak{C})) \cdot \mathsf{height}(\mathfrak{C}).\end{aligned}$$

Note that $(\star)$ follows from the facts that $\mathsf{weight}(\mathfrak{C}) = |\mathsf{Interface}(\mathfrak{C})| = |B|$ for $\mathfrak{C} \in \mathsf{Interface}^{-1}(B)$ (Definition 8); and $\mathsf{height}_T(\mathfrak{C})$, or by definition the height of the home bag of $\mathfrak{C}$, is at least $\mathsf{height}(T^{\mathfrak{C}})$ (property (P6)). Hence, after summing the above over all sets $B \in \mathcal{I}(T_{\mathrm{pref}}, X)$, we obtain that

$$\sum_{B \in \mathcal{I}(T_{\mathrm{pref}}, X)} \Phi_{\mathcal{T}'}(T^B_{\mathrm{pref}}) \leqslant 52 \cdot \sum_{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)} g(\mathsf{weight}(\mathfrak{C})) \cdot \mathsf{height}(\mathfrak{C}).$$

Now we use Lemma 3.4.21. Let $\mathsf{collmap} : \mathsf{Coll}(T_{\mathrm{pref}}, X) \to \widetilde{F} \cup \mathsf{App}(T_{\mathrm{pref}})$ be the mapping from this lemma. Let us split the sum above into two terms:

$$\sum_{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)} g(\mathsf{weight}(\mathfrak{C})) \cdot \mathsf{height}(\mathfrak{C}) \leqslant \sum_{\substack{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X) \\ \mathsf{collmap}(\mathfrak{C}) \in \widetilde{F}}} g(\mathsf{weight}(\mathfrak{C})) \cdot \mathsf{height}(\mathfrak{C}) + \sum_{\substack{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X) \\ \mathsf{collmap}(\mathfrak{C}) \in \mathsf{App}(T_{\mathrm{pref}})}} g(\mathsf{weight}(\mathfrak{C})) \cdot \mathsf{height}(\mathfrak{C}). \tag{3.8}$$

(The inequality comes from the fact that it is possible that $\mathsf{App}(T_{\mathrm{pref}}) \cap \widetilde{F} \neq \emptyset$.) To bound the first sum on the right-hand side we use the fact that if $\mathsf{collmap}(\mathfrak{C}) \in \widetilde{F}$, then

- $\mathsf{weight}(\mathfrak{C}) < |\mathsf{bag}(\mathsf{collmap}(\mathfrak{C}))|$,
- $\mathsf{height}(\mathfrak{C}) \leqslant \mathsf{height}_T(\mathsf{collmap}(\mathfrak{C}))$, and
- $|\mathsf{collmap}^{-1}(t)| \leqslant \ell + 4$, for each $t \in \widetilde{F}$.

Hence,

$$\begin{aligned} \sum_{\substack{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X) \\ \mathsf{collmap}(\mathfrak{C}) \in \widetilde{F}}} g(\mathsf{weight}(\mathfrak{C})) \cdot \mathsf{height}(\mathfrak{C}) &\leqslant \sum_{\substack{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X) \\ \mathsf{collmap}(\mathfrak{C}) \in \widetilde{F}}} g(|\mathsf{bag}(\mathsf{collmap}(\mathfrak{C}))| - 1) \cdot \mathsf{height}_T(\mathsf{collmap}(\mathfrak{C})) \\ &\leqslant \sum_{t \in \widetilde{F}} (\ell + 4) \cdot g(|\mathsf{bag}(t)| - 1) \cdot \mathsf{height}_T(t). \end{aligned} \tag{3.9}$$

To bound the second sum on the right-hand side of Eq. (3.8), we use the fact that if $\mathsf{collmap}(\mathfrak{C}) \in \mathsf{App}(T_{\mathrm{pref}})$, then

- $\mathsf{weight}(\mathfrak{C}) = |\mathsf{Interface}(\mathfrak{C})| \leqslant \ell + 1 = 6k + 6$ (this follows from the fact that $\mathsf{Interface}(\mathfrak{C})$ is a subset of the home bag of $\mathsf{Interface}(\mathfrak{C})$ by property (P5)),
- $\mathsf{height}(\mathfrak{C}) \leqslant \mathsf{height}_T(\mathsf{collmap}(\mathfrak{C}))$, and
- $|\mathsf{collmap}^{-1}(t)| \leqslant \ell + 4$, for each $t \in \mathsf{App}(T_{\mathrm{pref}})$.

Therefore,

$$\begin{aligned} \sum_{\substack{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X) \\ \mathsf{collmap}(\mathfrak{C}) \in \mathsf{App}(T_{\mathrm{pref}})}} g(\mathsf{weight}(\mathfrak{C})) \cdot \mathsf{height}(\mathfrak{C}) &\leqslant \sum_{\substack{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X) \\ \mathsf{collmap}(\mathfrak{C}) \in \mathsf{App}(T_{\mathrm{pref}})}} g(6k + 6) \cdot \mathsf{height}_T(\mathsf{collmap}(\mathfrak{C})) \\ &\leqslant \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} (\ell + 4) \cdot g(6k + 6) \cdot \mathsf{height}_T(t) \\ &\leqslant \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} k^{\mathcal{O}(k)} \cdot \mathsf{height}_T(t). \end{aligned} \tag{3.10}$$

By plugging Eq. (3.9) and Eq. (3.10) into the inequality Eq. (3.8), we obtain the desired bound. ◁

Claim 3.4.25. *The following inequality holds:*

$$\sum_{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)} \Phi_{\mathcal{T}'}(T^{\mathfrak{C}}) \leqslant (\ell + 1) \cdot \left(\sum_{t \in \widetilde{F}} g(|\mathsf{bag}(t)| - 1) \cdot \mathsf{height}_T(t) \right) + \sum_{t \notin F} \Phi_{\mathcal{T}}(t).$$

Proof of the claim. We have that

$$\sum_{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)} \Phi_{\mathcal{T}'}(T^{\mathfrak{C}}) = \sum_{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)} \sum_{t \in V(T^{\mathfrak{C}})} \Phi_{\mathcal{T}'}(t)$$

For each node $t \in V(T^{\mathfrak{C}})$, consider the original copy $\mathsf{origin}(t)$ of the node in T. Note that $\mathsf{origin}(t)$ is either an explored node and then $\mathsf{origin}(t) \in \widetilde{F}$, or a descendant of a blockage and then $\mathsf{origin}(t) \notin F$. Thus:

$$\sum_{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)} \Phi_{\mathcal{T}'}(T^{\mathfrak{C}}) = \sum_{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)} \sum_{\substack{t \in V(T^{\mathfrak{C}}) \\ \mathsf{origin}(t) \in \widetilde{F}}} \Phi_{\mathcal{T}'}(t) + \sum_{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)} \sum_{\substack{t \in V(T^{\mathfrak{C}}) \\ \mathsf{origin}(t) \notin F}} \Phi_{\mathcal{T}'}(t). \tag{3.11}$$

Recall that for every $t \in V(T^{\mathfrak{C}})$, if $\mathsf{origin}(t)$ is unexplored, then $|\mathsf{bag}^{\mathfrak{C}}(t)| = |\mathsf{bag}(\mathsf{origin}(t))|$ and $|\mathsf{origin}^{-1}(\mathsf{origin}(t))| = 1$ (property (P8)), and moreover $\mathsf{height}_{T^{\mathfrak{C}}}(t) \leqslant \mathsf{height}_T(\mathsf{origin}(t))$ (property (P9)). Since $\mathsf{height}_{T'}(t) = \mathsf{height}_{T^{\mathfrak{C}}}(t)$ for every $t \in V(T^{\mathfrak{C}})$, we get that

$$\begin{aligned}
\sum_{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)} \sum_{\substack{t \in V(T^{\mathfrak{C}}) \\ \mathsf{origin}(t) \notin F}} \Phi_{\mathcal{T}'}(t) &= \sum_{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)} \sum_{\substack{t \in V(T^{\mathfrak{C}}) \\ \mathsf{origin}(t) \notin F}} g(|\mathsf{bag}^{\mathfrak{C}}(t)|) \cdot \mathsf{height}_{T'}(t) \\
&\leqslant \sum_{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)} \sum_{\substack{t \in V(T^{\mathfrak{C}}) \\ \mathsf{origin}(t) \notin F}} g(|\mathsf{bag}(\mathsf{origin}(t))|) \cdot \mathsf{height}_T(\mathsf{origin}(t)) \\
&\leqslant \sum_{t \notin F} g(|\mathsf{bag}(t)|) \cdot \mathsf{height}_T(t) = \sum_{t \notin F} \Phi_{\mathcal{T}}(t).
\end{aligned} \tag{3.12}$$

Note that above there might exist unexplored nodes $t \in V(T) \setminus F$ for which $\mathsf{origin}^{-1}(t) = \emptyset$.

Now, consider the explored nodes. By the fact that for every $t \in V(T^{\mathfrak{C}})$ with explored $\mathsf{origin}(t)$, we have $|\mathsf{bag}^{\mathfrak{C}}(t)| < |\mathsf{bag}(\mathsf{origin}(t))|$ and $|\mathsf{origin}^{-1}(\mathsf{origin}(t))| \leqslant \ell + 1$ (property (P7)), and moreover $\mathsf{height}_{T^{\mathfrak{C}}}(t) \leqslant \mathsf{height}_T(\mathsf{origin}(t))$ (again property (P9)), we find that

$$\begin{aligned}
\sum_{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)} \sum_{\substack{t \in V(T^{\mathfrak{C}}) \\ \mathsf{origin}(t) \in \widetilde{F}}} \Phi_{\mathcal{T}'}(t) &= \sum_{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)} \sum_{\substack{t \in V(T^{\mathfrak{C}}) \\ \mathsf{origin}(t) \in \widetilde{F}}} g(|\mathsf{bag}^{\mathfrak{C}}(t)|) \cdot \mathsf{height}_{T'}(t) \\
&\leqslant \sum_{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)} \sum_{\substack{t \in V(T^{\mathfrak{C}}) \\ \mathsf{origin}(t) \in \widetilde{F}}} g(|\mathsf{bag}(\mathsf{origin}(t))| - 1) \cdot \mathsf{height}_T(\mathsf{origin}(t)) \\
&\leqslant \sum_{t \in \widetilde{F}} (\ell + 1) \cdot g(|\mathsf{bag}(t) - 1|) \cdot \mathsf{height}_T(t).
\end{aligned}$$

By plugging the bound above together with Eq. (3.12) into Eq. (3.11), we obtain the desired inequality. ◁

We then combine the above to obtain a bound for the potential of $\mathcal{T}'$.

Claim 3.4.26. *The following inequality holds:*

$$\Phi(\mathcal{T}') \leqslant \Phi_{\mathcal{T}}(\widetilde{F}) - |\widetilde{F}| + \sum_{t \notin F} \Phi_{\mathcal{T}}(t) + k^{\mathcal{O}(k)} \cdot \left(|T_{\mathrm{pref}}| + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(t) \right) \cdot \log N$$

Proof of the claim. Recall that (Eq. (3.4)):

$$\Phi(\mathcal{T}') = \Phi_{\mathcal{T}'}(T^X) + \sum_{B \in \mathcal{I}(T_{\mathrm{pref}}, X)} \Phi_{\mathcal{T}'}(T^B_{\mathrm{pref}}) + \sum_{\mathfrak{C} \in \mathsf{Coll}(T_{\mathrm{pref}}, X)} \Phi_{\mathcal{T}'}(T^{\mathfrak{C}}).$$

By applying inequalities from Claim 3.4.23, Claim 3.4.24 and Claim 3.4.25, we obtain that:

$$\begin{aligned}
\Phi(\mathcal{T}') \leqslant\ & k^{\mathcal{O}(k)} \cdot \left(|T_{\mathrm{pref}}| + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(t) \right) \cdot \log N \\
& + 52(\ell + 4) \cdot \left(\sum_{t \in \widetilde{F}} g(|\mathsf{bag}(t)| - 1) \cdot \mathsf{height}_T(t) \right) + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} k^{\mathcal{O}(k)} \cdot \mathsf{height}_T(t) \\
& + (\ell + 1) \cdot \left(\sum_{t \in \widetilde{F}} g(|\mathsf{bag}(t)| - 1) \cdot \mathsf{height}_T(t) \right) + \sum_{t \notin F} \Phi_{\mathcal{T}}(t).
\end{aligned}$$

From the inspection of the definition of $g(\cdot)$ in Section 3.4.1 it follows that

$$g(x) \geqslant (53\ell + 209) \cdot g(x-1) + 1 \qquad \text{for every } x \in \mathbb{Z}_{\geqslant 1}.$$

Using that we can bound two of the terms from the inequality above:

$$\begin{aligned} & 52(\ell+4) \cdot \left(\sum_{t \in \widetilde{F}} g(|\mathsf{bag}(t)| - 1) \cdot \mathsf{height}_T(t) \right) + (\ell+1) \cdot \left(\sum_{t \in \widetilde{F}} g(|\mathsf{bag}(t)| - 1) \cdot \mathsf{height}_T(t) \right) \\ = & \sum_{t \in \widetilde{F}} (53\ell + 209) \cdot g(|\mathsf{bag}(t)| - 1) \cdot \mathsf{height}_T(t) \\ \leqslant & \sum_{t \in \widetilde{F}} (g(|\mathsf{bag}(t)|) \cdot \mathsf{height}_T(t) - 1) = \Phi_{\mathcal{T}}(\widetilde{F}) - |\widetilde{F}|. \end{aligned}$$

Therefore,

$$\Phi(\mathcal{T}') \leqslant \Phi_{\mathcal{T}}(\widetilde{F}) - |\widetilde{F}| + \sum_{t \notin F} \Phi_{\mathcal{T}}(t) + k^{\mathcal{O}(k)} \cdot \left(|T_{\mathrm{pref}}| + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(t) \right) \cdot \log N,$$

as desired. ◁

We are ready to prove Lemma 3.4.22. Recall that from Eq. (3.3):

$$\Phi(\mathcal{T}) \geqslant \sum_{t \in T_{\mathrm{pref}}} \mathsf{height}_T(t) + \Phi_{\mathcal{T}}(\widetilde{F}) + \sum_{t \notin F} \Phi_{\mathcal{T}}(t).$$

This inequality, combined with the bound from Claim 3.4.26, gives us:

$$\Phi(\mathcal{T}) - \Phi(\mathcal{T}') \geqslant \sum_{t \in T_{\mathrm{pref}}} \mathsf{height}_T(t) + |\widetilde{F}| - k^{\mathcal{O}(k)} \cdot \left(|T_{\mathrm{pref}}| + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(t) \right) \cdot \log N;$$

this ends the proof of the lemma. □

Clearly, Lemma 3.4.22 implies property (POT) of the refinement operation. Furthermore, we can derive from it a bound on the size of $\widetilde{F}$:

$$|\widetilde{F}| \leqslant \Phi(\mathcal{T}) - \Phi(\mathcal{T}') - \sum_{t \in T_{\mathrm{pref}}} \mathsf{height}_T(t) + k^{\mathcal{O}(k)} \cdot \left(|T_{\mathrm{pref}}| + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(t) \right) \cdot \log N.$$

Recall that, by Fact 3.4.20, the running time of the refinement call is upper-bounded by:

$$2^{\mathcal{O}(k^9)}(|\widetilde{F}| + |T_{\mathrm{pref}}| \cdot \log N)$$

By using the obtained bound on $|\widetilde{F}|$, we obtain the desired bound on the running time as given by property (RT1).

3.5 Height improvement

In this section we leverage the refinement operation defined in Section 3.4 to produce a data structure that allows us to maintain a tree decomposition of small height. As in Section 3.4, we assume $\ell = 6k + 5$ and take $\Phi := \Phi_\ell$ and $g := g_\ell$ as defined in Section 3.4.1. We prove that:

Lemma 3.5.1 (Height improvement data structure)**.** *Fix $k \in \mathbb{N}$ and let $\ell = 6k + 5$. The $(\ell+1)$-prefix-rebuilding data structure from Lemma 3.4.1 maintaining a tree decomposition $\mathcal{T} = (T, \mathsf{bag})$ can be extended to additionally support the following operation:*

- ImproveHeight(): *Updates $\mathcal{T}$ through a sequence of prefix-rebuilding updates, producing a tree decomposition $\mathcal{T}' = (T', \mathsf{bag}')$ such that*

$$\mathsf{height}(T') \leqslant 2^{\mathcal{O}(k \log k \sqrt{\log n \log\log n})} \qquad \text{and} \qquad |V(T')| \leqslant k^{\mathcal{O}(k)} \cdot n^3.$$

Also, $\Phi(\mathcal{T}') \leqslant \Phi(\mathcal{T})$ and if the width of $\mathcal{T}$ is at most ℓ, then the width of $\mathcal{T}'$ is also at most ℓ.

The worst-case running time of ImproveHeight *is bounded by*

$$2^{\mathcal{O}(k^9)} \cdot (\Phi(\mathcal{T}) - \Phi(\mathcal{T}')) + \mathcal{O}(1).$$

Lemma 3.5.1 will be crucial in ensuring the efficiency of the data structure maintaining tree decompositions of graphs dynamically: After each update to the tree decomposition, we will call ImproveHeight so as to ensure that the height of the decomposition stays sufficiently small. Note here that all prefix-rebuilding updates performed by ImproveHeight in order to decrease the height of the decomposition are essentially "free" in terms of amortized running time: The running time of the improvement is fully amortized by the decrease in the potential value, i.e. $\Phi(\mathcal{T}) - \Phi(\mathcal{T}')$. In particular, this decrease in potential is always nonnegative.

The rest of this section is dedicated to the proof of Lemma 3.5.1.

3.5.1 Algorithm description

The centerpiece of the proof of Lemma 3.5.1 is a prefix-rebuilding data structure that, given a sufficiently high tree T (the shape of our current tree decomposition), determines an *unbalanced* prefix T_{pref}. This prefix will then be passed to the refinement operation from Lemma 3.4.1, aiming to improve the height of the decomposition. The precise statement of this claim follows.

Lemma 3.5.2. *Let $c \geqslant 2$ and T be a binary tree with at most N nodes. If the height of T is at least $2^{\Omega(\sqrt{\log N \log c})}$, then there exists a nonempty prefix T_{pref} of T so that*

$$c \cdot \left(|T_{\mathrm{pref}}| + \sum_{a \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(a) \right) \leqslant \sum_{x \in T_{\mathrm{pref}}} \mathsf{height}_T(x). \tag{3.13}$$

Moreover, if we can access the height of each node of T in constant time, then such T_{pref} can be computed in time $\mathcal{O}(|T_{\mathrm{pref}}|)$.

Note that Lemma 3.5.2 immediately implies that, for every $c \geqslant 2$ and $\ell \geqslant 1$, there exists an ℓ-prefix rebuilding data structure with overhead $\mathcal{O}(1)$ maintaining a tree decomposition $\mathcal{T} = (T, \mathsf{bag})$ that additionally supports the following operation:

- GetUnbalanced(): Assuming that T has size at most N and height at least $2^{\Omega(\sqrt{\log N \log c})}$, returns a nonempty prefix T_{pref} of T satisfying Eq. (3.13). The worst-case running time is bounded by $|T_{\mathrm{pref}}|$.

We now show how this prefix-rebuilding data structure is used in the proof of Lemma 3.5.1.

Proof of Lemma 3.5.1. We assume that the data structure has access to the refinement prefix-rebuilding data structure from Lemma 3.4.1. Also we set $B := g_\ell(\ell + 1) \leqslant k^{\mathcal{O}(k)}$ and initialize the prefix-rebuilding data structure promised by Lemma 3.5.2 with $N := B \cdot n^3$ and $c := 2^{\mathcal{O}(k \log k)} \log n$. We will fix the precise value of c later in the course of the proof. Let $H = 2^{\Theta(\sqrt{\log N \log c})}$ be the threshold on the height promised by the statement of Lemma 3.4.1. We shall call the maintained tree decomposition T *deep* if its height is at least H and *shallow* otherwise.

We now describe the implementation of ImproveHeight. First, we ensure that T is of reasonable size: If $|V(T)|$ is at least $B \cdot n^3$, then we recompute the entire tree decomposition from scratch using the algorithm from Theorem 3.4.3 on $\mathcal{T}$ and make it binary, while increasing its size only by a constant factor, thus getting a new tree decomposition $\mathcal{T}^\star = (T^\star, \mathsf{bag}^\star)$. We call this process *shrinking*. If shrinking occurs, we discard the old refinement data structure and initialize a fresh one with $\mathcal{T}^\star$ (note that this can be represented as a prefix rebuilding update of size $|V(T)| + |V(T^\star)|$). Note that $\mathcal{T}^\star$ is a tree decomposition of width at most k and size at most $\mathcal{O}(n)$, hence $\Phi(\mathcal{T}^\star) = \mathcal{O}(B \cdot n^2)$. Observe that since $\Phi(\mathcal{T}) \geqslant B \cdot n^3 \gg \Phi(\mathcal{T}^\star)$, the decrease in the potential value will cover the computational cost incurred from the shrinking.

After ensuring that T has reasonable size, we check whether T is shallow. If this is the case, we are done. Otherwise, we extract an unbalanced prefix T_{pref} of T through $\mathsf{GetUnbalanced}()$, call $\mathsf{Refine}(T_{\mathrm{pref}})$ and apply to (T, bag) the prefix-rebuilding update it returned.

Now, after a single pass of this procedure, T might still be deep. Therefore we repeat this sequence of operations above in a loop until the tree becomes shallow. We stress that at each iteration of the loop we also check whether T is too big and potentially shrink it as described above.

Let us now prove that each pass of this procedure decreases $\Phi(\mathcal{T})$ significantly. Suppose $\mathcal{T}$ is our current deep tree decomposition of size at most N, T_{pref} is the prefix located by $\mathsf{GetUnbalanced}$, and $\mathcal{T}'$ is the tree decomposition resulting from applying $\mathsf{Refine}(T_{\mathrm{pref}})$ to T. The property (POT) of the tree decomposition refinement, adjusted to our current notation, states that

$$\Phi(\mathcal{T}) - \Phi(\mathcal{T}') \geqslant \sum_{t \in T_{\mathrm{pref}}} \mathsf{height}_T(t) - 2^{\mathcal{O}(k \log k)} \cdot \left(|T_{\mathrm{pref}}| + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(t) \right) \cdot \log |V(T)|.$$

We have that $|V(T)| \leqslant N = B \cdot n^3$, where $B = k^{\mathcal{O}(k)}$, hence $\log |V(T)| \leqslant \mathcal{O}(\log n + k \log k)$. Therefore, we can simplify the bound above as follows:

$$\Phi(\mathcal{T}) - \Phi(\mathcal{T}') \geqslant \sum_{t \in T_{\mathrm{pref}}} \mathsf{height}_T(t) - 2^{\mathcal{O}(k \log k)} \cdot \left(|T_{\mathrm{pref}}| + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(t) \right) \cdot \log n. \tag{3.14}$$

Let us now expand the $\mathcal{O}(\cdot)$ notation in Eq. (3.14) and fix a constant $1 < C \in 2^{\mathcal{O}(k \log k)}$, depending only on k, such that the $2^{\mathcal{O}(k \log k)}$ term in Eq. (3.14) is upper-bounded by C. Let also $c \coloneqq 2C \log n = 2^{\mathcal{O}(k \log k)} \log n$. With that, we can rewrite the inequality again as:

$$\Phi(\mathcal{T}) - \Phi(\mathcal{T}') \geqslant \sum_{t \in T_{\mathrm{pref}}} \mathsf{height}_T(t) - \frac{1}{2} c \cdot \left(|T_{\mathrm{pref}}| + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(t) \right). \tag{3.15}$$

So by Eq. (3.13), we have

$$\Phi(\mathcal{T}) - \Phi(\mathcal{T}') \geqslant \frac{1}{2} \sum_{t \in T_{\mathrm{pref}}} \mathsf{height}_T(t). \tag{3.16}$$

Then the running time of a single improvement pass is dominated by the call $\mathsf{Refine}(T_{\mathrm{pref}})$, whose time complexity by (RT1) is bounded by

$$\begin{aligned}
&2^{\mathcal{O}(k^9)} \cdot \left(\left(|T_{\mathrm{pref}}| + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(t) \right) \cdot \log n + \max(\Phi(\mathcal{T}) - \Phi(\mathcal{T}'), 0) \right) \\
&\overset{(3.13)}{\leqslant} 2^{\mathcal{O}(k^9)} \cdot \left(\sum_{t \in T_{\mathrm{pref}}} \mathsf{height}_T(t) + \max(\Phi(\mathcal{T}) - \Phi(\mathcal{T}'), 0) \right) \\
&\overset{(3.16)}{\leqslant} 2^{\mathcal{O}(k^9)} \left(\Phi(\mathcal{T}) - \Phi(\mathcal{T}') \right).
\end{aligned}$$

So whenever a tree $\mathcal{T}$ is improved to $\mathcal{T}'$ through $\mathsf{GetUnbalanced}$, this decreases the potential function of the decomposition and the improvement takes time at most $2^{\mathcal{O}(k^9)} (\Phi(\mathcal{T}) - \Phi(\mathcal{T}'))$. Similarly, when $\mathcal{T}$ is changed to $\mathcal{T}^\star$ through the shrinking step, this also decreases the potential function significantly and the improvement takes time $\mathcal{O}(\Phi(\mathcal{T}) - \Phi(\mathcal{T}^\star))$. Hence, the decomposition $\mathcal{T}'$ formed after all improvement and shrinking steps is determined in total time $2^{\mathcal{O}(k^9)} (\Phi(\mathcal{T}) - \Phi(\mathcal{T}')) + \mathcal{O}(1)$; here, we add $\mathcal{O}(1)$ to the complexity to account for the fact that no improvements may be necessary for the initial decomposition. This concludes the proof. □

Hence it only remains to prove Lemma 3.5.2. We give the proof of this lemma in the following section.

3.5.2 Finding an unbalanced prefix

In this section we will describe how an unbalanced prefix of a deep tree T can be found efficiently; this will constitute the proof of Lemma 3.5.2. So assume that the parameters c, N are fixed as in the statement of the lemma, and let T be a binary tree with at most N nodes and height $2^{\Omega(\sqrt{\log N \log c})}$. Note that we can assume that $c < \frac{1}{6}N$; otherwise there is nothing to prove if the constant hidden in the Ω notation is large enough (then $2^{\Omega(\sqrt{\log N \log c})} > N$ and so the tree T as above cannot exist). We look for a prefix T_{pref} of T satisfying Eq. (3.13).

Assume that we are given two sequences of positive integers: $h_1 > h_2 > \ldots > h_a$ and $n_1 > n_2 > \ldots > n_a$, where $h_j \leqslant n_j$ for $j < a$ and $h_a > n_a$. For a node t of T and $j \in [a]$, we say that the subtree T_t of T rooted at t is *j-shallow* if $\mathsf{height}(t) < h_j$, and *j-deep* otherwise. Similarly, we say that the subtree is *j-big* if $\mathsf{size}(t) > n_j$; and *j-small* otherwise (not to be confused with the notion of c-small closures). We define a recursive function $\mathsf{GetUnbalanced}(T_t, j)$ that takes an integer $j \in [a]$ and a subtree T_t of T rooted at $t \in V(T)$ that is both j-small and j-deep: Let P be any longest path from t to a leaf of T_t. Define

$$\mathsf{UBApp}(P, j) \coloneqq \{t' \in \mathsf{App}(P) \mid T_{t'} \text{ is } (j+1)\text{-deep and } (j+1)\text{-small}\}$$

as the set of unbalanced trees rooted at the appendices of P. On the other hand, we will say that each tree rooted at a node of $\mathsf{App}(P) \setminus \mathsf{UBApp}(P, j)$, i.e., each balanced tree rooted at an appendix of P, is *P-found*. Then, $\mathsf{GetUnbalanced}(T_t, j)$ returns

$$P \cup \bigcup_{t' \in \mathsf{UBApp}(P,j)} \mathsf{GetUnbalanced}(T_{t'}, j+1).$$

Clearly, GetUnbalanced is well-defined: In each recursive call $\mathsf{GetUnbalanced}(T_{t'}, j+1)$ it is guaranteed that $T_{t'}$ is both $(j+1)$-deep and $(j+1)$-small; and thus, since $h_a > n_a$ and the height of each tree is upper-bounded by its size, $\mathsf{GetUnbalanced}(\cdot, a)$ is never called. Moreover, $\mathsf{GetUnbalanced}(T, 1)$ returns a prefix of T.

Before arguing why GetUnbalanced satisfies the bounds in the statement of Lemma 3.5.1 and why it even terminates at all, we first specify the sequences (h_i) and (n_i). To this end, will use the following lemma:

Lemma 3.5.3. *Let N and d be real numbers with $1 < d < N$. There exist sequences of real numbers $h_1 > h_2 > \ldots > h_a$ and $n_1 > n_2 > \ldots > n_a$ such that*

- *$h_1 \leqslant 2^{\mathcal{O}(\sqrt{\log N \log d})}$ and $n_1 \geqslant N$,*
- *$h_i \leqslant n_i$ for all $1 \leqslant i < a$,*
- *$h_a > n_a > 1$, and*
- *$n_i = n_{i+1} \cdot h_{i+1} \geqslant d \cdot h_{i+1} = h_i$ for all $1 \leqslant i < a$.*

Proof. Let a be the smallest integer such that $d^{\frac{a(a+1)}{2}} \geqslant N$. We set

$$h_i \coloneqq d^{a+2-i} \qquad \text{and} \qquad n_i \coloneqq d^{\frac{(a-i+1)\cdot(a-i+2)}{2}}.$$

It is easy to verify the three last required properties. Therefore, it remains to prove that $h_1 \leqslant 2^{\mathcal{O}(\sqrt{\log N \log d})}$.

As $d < N$, we have that $a \geqslant 2$. By the definition of a, we also have $d^{\frac{a(a-1)}{2}} < N \leqslant d^{\frac{a(a+1)}{2}}$, which implies that $\frac{a(a-1)}{2} \cdot \log d < \log N \leqslant \frac{a(a+1)}{2} \cdot \log d$. As $a^2 \geqslant \frac{a(a+1)}{2}$, we also have that $a^2 \log d \geqslant \log N$ and thus $a \geqslant \sqrt{\frac{\log N}{\log d}}$. On the other hand, as $a \geqslant 2$, we have $a^2 \leqslant 4 \cdot \frac{a(a-1)}{2}$, so

$$a \log d = \frac{a^2 \log d}{a} \leqslant \frac{4\frac{a(a-1)}{2} \log d}{a} \leqslant \frac{4 \log N}{a} \leqslant \frac{4 \log N}{\sqrt{\frac{\log N}{\log d}}} = 4\sqrt{\log N \log d}.$$

Therefore, $h_1 = d^{a+1} \leqslant d^{2a} = 2^{2a \log d} = 2^{\mathcal{O}(\sqrt{\log N \log d})}$. □

So let $(h_i), (n_i)$ are the sequences from Lemma 3.5.3 with $d = 6c$. Note that $1 < d < N$ by the discussion above, so the lemma can indeed be applied.

It remains to show that the set $T_{\mathrm{pref}} := \mathsf{GetUnbalanced}(T, 1)$ indeed satisfies Eq. (3.13). Let $\mathcal{W}$ be the set of subtrees of T rooted at the vertices of $\mathsf{App}(T_{\mathrm{pref}})$. As T_{pref} is the disjoint union of the vertical paths found by $\mathsf{GetUnbalanced}$, each subtree in $\mathcal{W}$ can be assigned the unique vertical path P found by $\mathsf{GetUnbalanced}$ such that the subtree is P-found.

Let us first focus on a recursive call $\mathsf{GetUnbalanced}(T_r, j)$, where T_r is the subtree of T rooted at r and $j \in [a-1]$. Let P be the longest path rooted at r found in the algorithm. Let $T_1, T_2, \ldots, T_b$ be the set of P-found subtrees. As T_r is binary, we have that $b \leqslant |P|$. From the invariant of $\mathsf{GetUnbalanced}$ we know that T_r is j-deep, that is $|P| = \mathsf{height}(T_r) \geqslant h_j$. As each P-found subtree is either $(j+1)$-shallow or $(j+1)$-big, we have that

$$\sum_{i=1}^{b} \mathsf{height}(T_i) \leqslant \sum_{i \mid T_i \text{ is } (j+1)\text{-shallow}} \mathsf{height}(T_i) + \sum_{i \mid T_i \text{ is } (j+1)\text{-big}} \mathsf{height}(T_i).$$

(Note that a subtree can be both $(j+1)$-shallow and $(j+1)$-big.) We will now bound each sum on the right hand side separately.

First, each $(j+1)$-shallow subtree has height smaller than h_{j+1}. As there are at most $b \leqslant |P|$ of them among the P-found subtrees, we get that

$$\sum_{i \mid T_i \text{ is } (j+1)\text{-shallow}} \mathsf{height}(T_i) \leqslant |P| \cdot h_{j+1}.$$

Second, P is a longest root-to-leaf path in T_r, so each $T_1, T_2, \ldots, T_b$ has height at most $|P|$. Moreover, as T_r is j-small, we have $|V(T_r)| \leqslant n_j$; and for each $(j+1)$-big tree T_i we have that $|V(T_i)| > n_{j+1}$. As $T_1, \ldots, T_b$ are pairwise disjoint and are subtrees of T_r, we conclude that at most $\frac{n_j}{n_{j+1}}$ of P-found trees are $(j+1)$-big. Hence,

$$\sum_{i \mid T_i \text{ is } (j+1)\text{-big}} \mathsf{height}(T_i) \leqslant |P| \cdot \frac{n_j}{n_{j+1}}.$$

We conclude that

$$\sum_{i=1}^{b} \mathsf{height}(T_i) \leqslant |P| \cdot \left(h_{j+1} + \frac{n_j}{n_{j+1}} \right). \tag{3.17}$$

Also we observe that

$$\sum_{t \in P} \mathsf{height}(t) = 1 + 2 + \cdots + |P| \geqslant \frac{|P|^2}{2} \geqslant \frac{|P| \cdot h_j}{2}. \tag{3.18}$$

We are now ready to prove the counterpart of Eq. (3.13) restricted to a single path P and the set of P-found subtrees.

Claim 3.5.4. *The following inequality holds:*

$$c \cdot \left(|P| + \sum_{i=1}^{b} \mathsf{height}(T_i) \right) \cdot \log n \leqslant \sum_{t \in P} \mathsf{height}(t).$$

Proof of the claim. Let us recall that $h_j = d \cdot h_{j+1}$, $h_j \geqslant 1$, and $\frac{n_j}{n_{j+1}} = h_{j+1}$. Because of that, we derive that

$$\begin{aligned} c \cdot \left(|P| + \sum_{i=1}^{b} \mathsf{height}(T_i) \right) &\overset{(3.17)}{\leqslant} c \left(|P| + |P| \left(h_{j+1} + \frac{n_j}{n_{j+1}} \right) \right) \\ &\leqslant c(|P| \cdot h_{j+1} + |P|(h_{j+1} + h_{j+1})) \\ &= 3c \cdot |P| \cdot h_{j+1}. \end{aligned}$$

On the other hand, we have that

$$\sum_{t \in P} \mathsf{height}(t) \overset{(3.18)}{\geqslant} \frac{|P| \cdot h_j}{2} = \frac{|P| \cdot d \cdot h_{j+1}}{2} = 3c \cdot |P| \cdot h_{j+1}$$

and the claim follows. ◁

Summing the inequality from Claim 3.5.4 over all paths P corresponding to the calls of $\mathsf{GetUnbalanced}$, we immediately get Eq. (3.13), which settles the combinatorial part of the proof of Lemma 3.5.2. For the algorithmic part, it is easy to see that $\mathsf{GetUnbalanced}(T, 1)$ indeed returns T_{pref} in time $|T_{\mathrm{pref}}|$. Hence Lemma 3.5.2 holds, and this implies that Lemma 3.5.1 is also true.

3.6 Proof of Lemma 3.2.5

After having done all the necessary preparations, we are ready to prove Lemma 3.2.5. For convenience, we recall its statement.

Lemma 3.2.5. *There is a data structure that for an integer $k \in \mathbb{N}$, fixed upon initialization, and a dynamic graph G, updated by edge insertions and deletions, maintains an annotated tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of G of width at most $6k+5$ using prefix-rebuilding updates under the promise that $\mathrm{tw}(G) \leqslant k$ at all times. More precisely, at every point in time the graph is guaranteed to have treewidth at most k and the data structure contains an annotated tree decomposition of G of width at most $6k+5$. The data structure can be initialized on k and an edgeless n-vertex graph G in time $2^{\mathcal{O}(k^8)} \cdot n$, and then every update:*

- *returns the sequence of prefix-rebuilding updates used to modify the tree decomposition; and*
- *takes amortized time $2^{\mathcal{O}(k^9 + k \log k \cdot \sqrt{\log n \log\log n})}$.*

Again, as in Section 3.4, we fix $\ell = 6k+5$ and take $\Phi := \Phi_\ell$ and $g := g_\ell$ as defined in Section 3.4.1.

First, we are going to describe how the data structure is implemented; then, we will bound the running time of a series of edge updates.

3.6.1 Data structure

In order to initialize a dynamically changing annotated tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$, we instantiate the following $(6k+6)$-prefix-rebuilding data structures:

- $\mathbb{D}$: the refinement data structure with overhead $2^{\mathcal{O}(k^8)}$ from Lemma 3.4.1, additionally supporting $\mathsf{ImproveHeight}$ (Lemma 3.5.1);
- $\mathbb{H}$: the data structure with overhead $\mathcal{O}(1)$ from Lemma 3.2.1, allowing us to query, for each vertex $v \in V(G)$, the top bag $\mathsf{Top}(v) \in V(T)$ containing v; and for each $t \in V(T)$, the height of t in $V(T)$.

Both data structures store the same decomposition $(T, \mathsf{bag}, \mathsf{edges})$: all prefix-rebuilding updates performed by $\mathbb{D}$ are forwarded to $\mathbb{H}$. After the initialization and after each query, we maintain the following invariant: $(T, \mathsf{bag}, \mathsf{edges})$ contains an annotated tree decomposition of G of width at most $\ell = 6k+5$, height at most $2^{\mathcal{O}(k \log k \sqrt{\log n \log\log n})}$ and size at most $k^{\mathcal{O}(k)} \cdot n^3$. After each query, we return the sequence of prefix-rebuilding updates performed by $\mathbb{D}$.

As for the initialization of our structure for an empty graph, we can initialize $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ with T being a complete binary tree (of height $\mathcal{O}(\log n)$) with n nodes and each bag containing a different vertex of $V(G)$. Obviously, for each $t \in V(T)$, we have $\mathsf{edges}(t) = \emptyset$. The initialization of $\mathbb{D}$ and $\mathbb{H}$ with it takes $2^{\mathcal{O}(k^8)} \cdot n$ time. We remark that such a decomposition $\mathcal{T}$ satisfies $\Phi(\mathcal{T}) \leqslant \mathcal{O}(kn)$ since the average height of a node in T is $\mathcal{O}(1)$, and $g(1) \in \mathcal{O}(k)$.

First assume that we are to add an edge uv to G. Let G' be equal to the graph G with the edge uv added. Towards that goal, we must first ensure that the edge condition is satisfied for the new edge, i.e., both u and v belong to the same bag of $\mathcal{T}$. Let $t_u = \mathsf{Top}(u)$ and $t_v = \mathsf{Top}(v)$ be the top bags of u and v, respectively, in T. If it is the case that $v \in \mathsf{bag}(t_u)$ or $u \in \mathsf{bag}(t_v)$, then the edge condition is already satisfied; however, we still need to update the function edges with the newly added edge uv. Let us recall from the definition of edges that we should include the edge uv in exactly one set: the set $\mathsf{edges}(t)$ for the topmost bag t containing both u and v. It is easy to verify that this bag is actually one of t_u or t_v, whichever is at the smaller height in T. Without loss of generality, assume it is t_v. We include uv in $\mathsf{edges}(t_v)$ by performing a simple prefix-rebuilding update on the path from the root of T to t_v. The prefix rebuilt has size at most $\mathsf{height}(T)$; hence, the update takes time $2^{\mathcal{O}(k^8)} \cdot \mathsf{height}(T)$ and produces a tree decomposition $\mathcal{T}'$ of G'. The structure of the tree decomposition remains unchanged here, so the invariant is preserved by the update.

Let us assume now that $v \notin \mathsf{bag}(t_u)$ and $u \notin \mathsf{bag}(t_v)$; in this case, we must expand some bags in the decomposition so as to satisfy the edge condition. Let P_u and P_v be the paths from the root of T to t_u and t_v, respectively. Then, the update of the decomposition proceeds in two steps: First, we add v to all the bags $\mathsf{bag}(t)$ for $t \in P_u \cup P_v$, and only then we add the edge uv to an appropriate set edges. The former is done by obtaining a weak description $\widehat{u}$ of a prefix-rebuilding update adding v to the required bags, invoking $\bar{u} := \mathsf{Strengthen}(\widehat{u})$ (Lemma 3.2.3) and then applying the resulting prefix-rebuilding operation $\bar{u}$. Then, the insertion of the edge uv to the appropriate set $\mathsf{edges}(t)$ is conducted as in the previous case. Note that all of the above can be performed in time $2^{\mathcal{O}(k^8)} \cdot \mathsf{height}(T)$.

Now, $\mathcal{T}' = (T, \mathsf{bag}', \mathsf{edges}')$ is a correct annotated tree decomposition of G', but adding u to the bags in $P_u \cup P_v$ might have increased the width of the decomposition to $\ell+1 = 6k+6$, thus invalidating the width invariant. In order to counteract this, we call $\mathsf{Refine}(P_u \cup P_v)$. Note that $P_u \cup P_v$ covers all bags of size $\ell+2 = 6k+7$, so the call satisfies the precondition of Lemma 3.4.1 and the annotated tree decomposition $\mathcal{T}''$ of G' produced by Refine has width at most ℓ (condition (WIDTH)). However, $\mathcal{T}''$ might now have too large height or size; we resolve this issue by invoking the height improvement ($\mathsf{ImproveHeight}$), resulting in a tree decomposition $\mathcal{T}'''$ of G' of width at most ℓ, height at most $2^{\mathcal{O}(k \log k \sqrt{\log n \log\log n})}$ and size at most $k^{\mathcal{O}(k)} \cdot n^3$. The final tree decomposition $\mathcal{T}'''$ satisfies all the prescribed invariants.

Deleting an edge uv from G is much simpler: We do not change the tree structure T or the contents of the bags of the tree decomposition. However, we still need to remove uv from the appropriate set $\mathsf{edges}(t)$ for $t \in V(T)$ using a prefix-rebuilding update. Locating the appropriate $t \in V(T)$ is done as before: Let $t_u = \mathsf{Top}(u)$, $t_v = \mathsf{Top}(v)$ and uv belongs to either $\mathsf{edges}(t_u)$ or $\mathsf{edges}(t_v)$, depending on whether t_u or t_v has smaller height. Assuming without loss of generality that $uv \in \mathsf{edges}(t_v)$, we construct a prefix-rebuilding update removing uv from $\mathsf{edges}(t_v)$ by rebuilding the path from the root of T to t_v.

3.6.2 Complexity analysis

Now, we are going to bound the amortized running time of the data structure. As already mentioned, we start with an edgeless graph and a decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of it with potential $\Phi(\mathcal{T}) = \mathcal{O}(kn)$.

We are only going to focus on the analysis of edge insertions; edge removals are more straightforward. Computing P_u, P_v and performing the introductory prefix-rebuilding updates takes time

$$2^{\mathcal{O}(k^8)} \cdot \mathsf{height}(T) \leqslant 2^{\mathcal{O}(k^8)} \cdot 2^{\mathcal{O}(k \log k \sqrt{\log n \log\log n})}. \tag{3.19}$$

Let G' be the graph G with an edge uv added and $\mathcal{T}' = (T, \mathsf{bag}', \mathsf{edges}')$ be the tree decomposition of G' after the initial prefix-rebuilding updates. (We stress that the decompositions $\mathcal{T}$ and $\mathcal{T}'$ have the same shape T.) If u was not added to any bags, the potential of $\mathcal{T}'$ has not increased and the entire query took time $2^{\mathcal{O}(k \log k \sqrt{\log n \log\log n})}$. Assume now that u has been added to some new bags. Recall then that u was added to at most $\mathcal{O}(\mathsf{height}(T))$ new bags, increasing the size of each of them to at most $\ell+2 = 6k+7$; the sizes of other bags remained unchanged. Therefore,

$$\begin{aligned} \Phi(\mathcal{T}') - \Phi(\mathcal{T}) &\leqslant g(6k+7) \cdot \mathcal{O}(\mathsf{height}(T)^2) \leqslant 2^{\mathcal{O}(k \log k)} \cdot \mathsf{height}(T)^2 \\ &\leqslant 2^{\mathcal{O}(k \log k \sqrt{\log n \log\log n})}. \end{aligned} \tag{3.20}$$

Let $\mathcal{T}''$ be the decomposition produced by applying $\mathsf{Refine}(T_{\mathrm{pref}})$ on $\mathcal{T}'$ with a prefix T_{pref} with $|T_{\mathrm{pref}}| \leqslant \mathcal{O}(\mathsf{height}(T))$. Recall the property (POT) bounding the potential change:

$$\Phi(\mathcal{T}') - \Phi(\mathcal{T}'') \geqslant \sum_{t \in T_{\mathrm{pref}}} \mathsf{height}_T(t) - 2^{\mathcal{O}(k \log k)} \cdot \left(|T_{\mathrm{pref}}| + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(t) \right) \cdot \log |V(T)|.$$

Therefore,

$$\begin{aligned} \Phi(\mathcal{T}'') - \Phi(\mathcal{T}') &\leqslant - \sum_{t \in T_{\mathrm{pref}}} \mathsf{height}_T(t) + 2^{\mathcal{O}(k \log k)} \cdot \left(|T_{\mathrm{pref}}| + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(t) \right) \cdot \log |V(T)| \\ &\leqslant 2^{\mathcal{O}(k \log k)} \cdot |T_{\mathrm{pref}}| \cdot \mathsf{height}(T) \cdot \log |V(T)| \\ &\leqslant 2^{\mathcal{O}(k \log k)} \cdot \mathsf{height}(T)^2 \cdot \log n \\ &\leqslant 2^{\mathcal{O}(k \log k \sqrt{\log n \log\log n})}. \end{aligned} \tag{3.21}$$

Here, we used the facts that $|T_{\mathrm{pref}}| \leqslant \mathcal{O}(\mathsf{height}(T))$, $|\mathsf{App}(T_{\mathrm{pref}})| \leqslant |T_{\mathrm{pref}}| + 1$ and $\log |V(T)| \leqslant \mathcal{O}(k \log k + \log n)$ (following from the invariants and the fact that T is binary).

Finally, let $\mathcal{T}'''$ be the final tree decomposition produced by running the height improvement operation ($\mathsf{ImproveHeight}$) on $\mathcal{T}''$. Since $\mathsf{ImproveHeight}$ never increases the potential value, we have $\Phi(\mathcal{T}''') \leqslant \Phi(\mathcal{T}'')$. We conclude that

$$\Phi(\mathcal{T}''') - \Phi(\mathcal{T}) \leqslant 2^{\mathcal{O}(k \log k)} \cdot \mathsf{height}(T)^2 \cdot \log n \leqslant 2^{\mathcal{O}(k \log k \sqrt{\log n \log\log n})}.$$

For the running time of the edge insertion, recall from property (RT1) that the application of the refinement operation on $\mathcal{T}'$ takes worst-case time

$$\begin{aligned} & 2^{\mathcal{O}(k^9)} \left(|T_{\mathrm{pref}}| + \sum_{t \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}(t) + \max(\Phi(\mathcal{T}') - \Phi(\mathcal{T}''), 0) \right) \\ \leqslant \; & 2^{\mathcal{O}(k^9)} \left(\mathsf{height}(T)^2 + \max(\Phi(\mathcal{T}') - \Phi(\mathcal{T}''), 0) \right) \\ \overset{(3.21)}{\leqslant} \; & 2^{\mathcal{O}(k^9)} \left(\mathsf{height}(T)^2 + (\Phi(\mathcal{T}') - \Phi(\mathcal{T}'')) + 2^{\mathcal{O}(k \log k)} \cdot \mathsf{height}(T)^2 \cdot \log n \right) \\ \leqslant \; & 2^{\mathcal{O}(k^9)} \left(2^{\mathcal{O}(k \log k \sqrt{\log n \log \log n})} + \Phi(\mathcal{T}') - \Phi(\mathcal{T}'') \right); \end{aligned} \tag{3.22}$$

immediately followed by the height improvement, which by Lemma 3.5.1 takes worst-case time

$$2^{\mathcal{O}(k^9)} \cdot (\Phi(\mathcal{T}'') - \Phi(\mathcal{T}''')) + \mathcal{O}(1). \tag{3.23}$$

Thus, the total running time is bounded by the sum of Eqs. (3.19), (3.22) and (3.23):

$$\begin{aligned} & 2^{\mathcal{O}(k^9 + k \log k \sqrt{\log n \log \log n})} + 2^{\mathcal{O}(k^9)} (\Phi(\mathcal{T}') - \Phi(\mathcal{T}''')) \\ \overset{(3.20)}{\leqslant} \; & 2^{\mathcal{O}(k^9 + k \log k \sqrt{\log n \log \log n})} + 2^{\mathcal{O}(k^9)} (\Phi(\mathcal{T}) - \Phi(\mathcal{T}''')). \end{aligned}$$

Since each update increases the potential value by at most $2^{\mathcal{O}(k \log k \sqrt{\log n \log \log n})}$, it follows that the amortized time complexity of each update is $2^{\mathcal{O}(k^9 + k \log k \sqrt{\log n \log \log n})}$, as claimed.

3.7 Dynamic automata

In this section we introduce a framework for dynamic maintenance of dynamic programming tables on tree decompositions under prefix-rebuilding updates. Concrete outcomes of this are proofs of Lemma 3.2.1, Lemma 3.2.4, and Lemma 3.3.23, but the introduced framework is general enough to also capture maintenance of any reasonable dynamic programming scheme.

We remark that maintenance of runs of automata on dynamic forests has already been investigated in the literature, and even for the much more general problem of dynamic enumeration; see for instance the works of Niewerth [Nie18] and of Amarilli et al. [ABMN19] and the bibliographic discussion within. In particular, many (though not all) results contained in this section could be in principle derived from [ABMN18, Lemma 7.3], but not in a black-box manner and without concrete bounds on update time. Therefore, for the sake of completeness, in this section we provide a self-contained presentation.

3.7.1 Tree decomposition automata

Our framework is based on a notion of automata processing tree decompositions. While this notion is tailored here to our specific purposes, the idea of processing tree decompositions using various kinds of automata or dynamic programming procedures dates back to the work of Courcelle [Cou90] and is a thoroughly researched topic; see appropriate chapters of textbooks [DF13, CFK+15, FG06] and bibliographic notes within. Hence, the entirety of this section can be considered a formalization of folklore.

Throughout this section we assume that all vertices of considered graphs come from a fixed, totally ordered, countable set of vertices Ω. Further, we assume that elements of Ω can be manipulated upon in constant time in the RAM model. The reader may assume that $\Omega = \mathbb{N}$.

Boundaried graphs. We will work with graphs with specified boundaries, as formalized next.

Definition 9. *A* boundaried graph *is an undirected graph G together with a vertex subset $\partial G \subseteq V(G)$, called the* boundary*, such that G has no edge with both endpoints in ∂G. A* boundaried tree decomposition *of a boundaried graph G is a triple $(T, \mathsf{bag}, \mathsf{edges})$ that is an annotated tree decomposition of G (treated as a normal graph) where in addition we require that ∂G is contained in the root bag.*

When speaking about a boundaried tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of a boundaried graph G, we redefine the adhesion of the root of T to be ∂G, rather than the empty set.

Suppose $(T, \mathsf{bag}, \mathsf{edges})$ is a boundaried tree decomposition of a boundaried graph G, and x is a node of T. Then we say that x *induces* a boundaried graph G_x and its boundaried tree decomposition $(T_x, \mathsf{bag}_x, \mathsf{edges}_x)$, defined as follows: If X is the set of descendants of x in T, then

$$\begin{aligned} G_x &= \left(\bigcup_{y \in X} \mathsf{bag}(y), \bigcup_{y \in X} \mathsf{edges}(y) \right); \\ \partial G_x &= \mathsf{adh}(x); \\ (T_x, \mathsf{bag}_x, \mathsf{edges}_x) &= (T, \mathsf{bag}, \mathsf{edges})|_X. \end{aligned}$$

It is clear that $(T_x, \mathsf{bag}_x, \mathsf{edges}_x)$ defined as above is a boundaried tree decomposition of G_x. We use the above notation only when the boundaried tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ is clear from the context.

Automata. We now introduce our automaton model.

Definition 10. *A* (deterministic) tree decomposition automaton *of* width ℓ *consists of*

- *a* state set Q*;*
- *a set of* accepting states $F \subseteq Q$*;*
- *an* initial mapping ι *that maps every boundaried graph G on at most $\ell + 1$ vertices to a state $\iota(G) \in Q$; and*
- *a* transition mapping δ *that maps every* 7*-tuple of form (B, X, Y, Z, J, q', q''), where $B \subseteq \Omega$ is a set of size at most $\ell + 1$, $X, Y, Z \subseteq B$, $J \in \binom{B}{2} \setminus \binom{X}{2}$, $q' \in Q$, and $q'' \in Q \cup \{\bot\}$ to a state $\delta(B, X, Y, Z, J, q', q'') \in Q$.*

The run *of a tree decomposition automaton $\mathcal{A}$ on a binary boundaried tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of a boundaried graph G is the unique labeling $\rho_{\mathcal{A}} \colon V(T) \to Q$ satisfying the following properties:*

- *For every leaf l of T, we have*
$$\rho_{\mathcal{A}}(l) = \iota(G_l).$$
- *For every nonleaf node x of T with one child y, we have*
$$\rho_{\mathcal{A}}(x) = \delta(\mathsf{bag}(x), \mathsf{adh}(x), \mathsf{adh}(y), \emptyset, \mathsf{edges}(x), \rho_{\mathcal{A}}(y), \bot).$$
- *For every nonleaf node x of T with two children y and z, we have*
$$\rho_{\mathcal{A}}(x) = \delta(\mathsf{bag}(x), \mathsf{adh}(x), \mathsf{adh}(y), \mathsf{adh}(z), \mathsf{edges}(x), \rho_{\mathcal{A}}(y), \rho_{\mathcal{A}}(z)).$$

A tree decomposition automaton $\mathcal{A}$ accepts *$(T, \mathsf{bag}, \mathsf{edges})$ if $\rho_{\mathcal{A}}(r) \in F$, where r is the root of T.*

Note that in the transitions described above, nodes with one child are treated by passing a "dummy state" $\bot \notin Q$ to the transition function instead of a state. Note that this allows δ also to recognize when there is only one child. Also, the automata model presented above could in principle distinguish the left child y from the right child z and treat states passed from them differently. However, this will never be the case in our applications: In all constructed automata, the transition mapping will be symmetric with respect to swapping the role of the children y and z.

We say that a tree decomposition automaton $\mathcal{A}$ has *evaluation time* τ if functions ι and δ can be evaluated on any tuple of their arguments in time τ, and moreover for a given $q \in Q$ it can be decided whether $q \in F$ in time τ. Note that we do *not* require the state space Q to be finite. In fact, in most of our applications it will be infinite, but we will be able to efficiently represent and manipulate the states.

We will often run a tree decomposition automaton on a non-boundaried annotated tree decomposition of a non-boundaried graph G. In such cases, we simply apply all the above definitions while treating G as a boundaried graph with an empty boundary.

We will also use *nondeterministic tree decomposition automata*, which are defined just like in Definition 10, except that ι and δ are the *initial relation* and the *transition relation*, instead of mappings. That is, ι is a relation consisting of pairs of the form (G, q), where G is a boundaried graph on at most $\ell + 1$ vertices, and $q \in Q$. Similarly, δ is a relation consisting of pairs of the form

$((B, X, Y, Z, J, q', q''), q)$, where (B, X, Y, Z, J, q', q'') is a 7-tuple like in the domain of the transition mapping, and $q \in Q$. Then a run of a nondeterministic automaton $\mathcal{A}$ on a boundaried binary tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ is a labeling ρ of the nodes of T with states such that $(G_l, \rho(l)) \in \iota$ for every leaf l, $((\mathsf{bag}(x), \mathsf{adh}(x), \mathsf{adh}(y), \emptyset, \mathsf{edges}(x), \rho(y), \bot), \rho(x)) \in \delta$ for every node x with one child y, and $((\mathsf{bag}(x), \mathsf{adh}(x), \mathsf{adh}(y), \mathsf{adh}(z), \mathsf{edges}(x), \rho(y), \rho(z)), \rho(x)) \in \delta$ for every node x with two children y and z. Note that a nondeterministic tree decomposition automaton may have multiple runs on a single tree decomposition. We say that $\mathcal{A}$ accepts $(T, \mathsf{bag}, \mathsf{edges})$ if there is a run of $\mathcal{A}$ on $(T, \mathsf{bag}, \mathsf{edges})$ that is *accepting*: the state associated with the root node is accepting.

In the context of nondeterministic automata, by evaluation time we mean the time needed to decide whether a given pair belongs to any of the relations ι or δ, or to decide whether a given state is accepting. Note that if $\mathcal{A}$ is a nondeterministic tree decomposition automaton with a finite state space Q, then we can determinize it – find a deterministic automaton $\mathcal{A}'$ that accepts the same tree decompositions – using the standard powerset construction. Then the state space of $\mathcal{A}'$ is 2^Q. In the following, all automata are deterministic unless explicitly stated.

3.7.2 Automata constructions

In subsequent sections we will use several automata. We now present four automata constructions that we will use.

Tree decomposition properties automata. We first construct three very simple automata that are used in Lemma 3.2.1 for maintaining properties of the tree decomposition itself.

Lemma 3.7.1. *For every $\ell \in \mathbb{N}$ there exists tree decomposition automata $\mathcal{H}_\ell$, $\mathcal{S}_\ell$, $\mathcal{C}_\ell$, each of width ℓ, with the following properties: For any graph G, annotated binary tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of G of width at most ℓ, and any node x of T:*

- *$\rho_{\mathcal{H}_\ell}(x)$ is equal to $\mathsf{height}(T_x)$,*
- *$\rho_{\mathcal{S}_\ell}(x)$ is equal to $|V(T_x)|$, and*
- *$\rho_{\mathcal{C}_\ell}(x)$ is equal to $|\mathsf{cmp}(x)|$.*

The evaluation times of $\mathcal{H}_\ell$ and $\mathcal{S}_\ell$ are $\mathcal{O}(1)$, and the evaluation time of $\mathcal{C}_\ell$ is $\mathcal{O}(\ell)$.

Proof. The state sets of each of the automata are $\mathbb{N}$. Let us define for all $n \in \mathbb{N}$ that $\max(n, \bot) = n$ and $n + \bot = n$. For the height automaton $\mathcal{H}_\ell$, the initial mapping and the transition mapping are defined as follows (here _ denotes any input value):

$$\begin{aligned} \iota(\_) &= 1 \\ \delta(\_, \_, \_, \_, \_, q', q'') &= 1 + \max(q', q''). \end{aligned}$$

For the size automaton $\mathcal{S}_\ell$, the initial mapping and the transition mapping are defined as follows:

$$\begin{aligned} \iota(\_) &= 1 \\ \delta(\_, \_, \_, \_, \_, q', q'') &= 1 + q' + q''. \end{aligned}$$

For the $|\mathsf{cmp}(x)|$ automaton $\mathcal{C}_\ell$, the initial mapping and the transition mapping are defined as follows:

$$\begin{aligned} \iota(G) &= |V(G) \setminus \partial G| \\ \delta(B, X, \_, \_, \_, q', q'') &= q' + q'' + |B \setminus X|. \end{aligned}$$

It is straightforward to see that these automata satisfy the required properties. □

CMSO_2-types automaton. The classic Courcelle's theorem [Cou90] states that there is an algorithm that given a CMSO_2 sentence φ and an n-vertex graph G together with a tree decomposition of width at most ℓ, decides whether $G \models \varphi$ in time $f(\ell, \varphi) \cdot n$, where f is a computable function. One way of proving Courcelle's theorem is to construct a dynamic programming procedure that processes the provided tree decomposition in a bottom-up fashion. This dynamic programming procedure can be understood as a tree decomposition automaton in the sense of Definition 10, yielding the following result. The proof is a completely standard application of the concept of CMSO_2-types, hence we only sketch it.

Lemma 3.7.2. *For every integer ℓ and CMSO_2 sentence φ there exists a tree decomposition automaton $\mathcal{A}_{\ell,\varphi}$ of width ℓ with the following property: For any graph G and its annotated binary tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of width at most ℓ, $\mathcal{A}_{\ell,\varphi}$ accepts $(T, \mathsf{bag}, \mathsf{edges})$ if and only if $G \models \varphi$. The evaluation time is bounded by $f(\ell, \varphi)$ for some computable function f.*

Proof sketch. Let p be the *rank* of φ: the maximum among the quantifier rank of φ and the moduli of modular predicates appearing in φ. Consider any finite $X \subseteq \Omega$ and let $\mathsf{CMSO}_2(X)$ consists of all CMSO_2 sentences that can additionally use elements of X as constants (formally, these are CMSO_2 sentences over the signature of graphs enriched by adding every $x \in X$ as a constant). It is well-known (see e.g. [Imm99, Exercise 6.11]) that for a given X as above, one can compute a set $\mathrm{Sentences}^p(X)$ consisting of at most $g(\ell, |X|)$ sentences of $\mathsf{CMSO}_2(X)$, for some computable g, such that for every $\psi \in \mathsf{CMSO}_2(X)$ of rank at most p, there is $\psi' \in \mathrm{Sentences}^p(X)$ that is equivalent to ψ in the sense of being satisfied in exactly the same graphs containing X. Also, the mapping $\psi \mapsto \psi'$ is computable.

For a given boundaried graph G with $X = \partial G$, we define the *p-type* of G as follows:

$$\mathrm{tp}^p(G) = \{\psi \in \mathrm{Sentences}^p(X) \mid G \models \psi\}.$$

Now, we construct automaton $\mathcal{A} = \mathcal{A}_{\ell,\varphi}$ so that for every boundaried binary tree decomposition $(T, \mathsf{bags}, \mathsf{edges})$ of a graph G, the run of $\mathcal{A}$ on $(T, \mathsf{bags}, \mathsf{edges})$ is as follows:

$$\rho_{\mathcal{A}}(x) = \mathrm{tp}^p(G_x) \qquad \text{for all } x \in V(T).$$

When constructing $\mathcal{A}$, the only nontrivial check is that one can define a suitable transition mapping δ. For this, it suffices to show that for every node x of T with children y and z, given types $\mathrm{tp}^p(G_y)$ and $\mathrm{tp}^p(G_z)$ together with the information about the bag of x (consisting of $\mathsf{bag}(x)$, $\mathsf{adh}(x)$, $\mathsf{adh}(y)$, $\mathsf{adh}(z)$, and $\mathsf{edges}(x)$), one can compute the type $\mathrm{tp}^p(G_x)$; and same for nodes with one child. This follows from a standard argument involving Ehrenfeucht-Fraïsse games, cf. [GK09, Mak04]. □

Lemma 3.7.2 can also be adapted to handle optimization problems expressible by $\mathsf{LinCMSO}_2$ sentences.

Lemma 3.7.3. *For every integer ℓ and $\mathsf{LinCMSO}_2$ sentence (φ, f) there exists a tree decomposition automaton $\mathcal{A} = \mathcal{A}_{\ell,(\varphi,f)}$ of width ℓ with the following property: For any graph G and its annotated binary tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of width at most ℓ with root r, the value of (φ, f) in G can be uniquely determined from $\rho_{\mathcal{A}}(r)$. The evaluation time is bounded by $g(\ell, |(\varphi, f)|)$ for some computable function g.*

Proof sketch. Fix a CMSO_2 formula $\varphi(X_1, \ldots, X_q)$ and an affine function $f(x_1, \ldots, x_q) = c_0 + c_1 x_1 + \ldots + c_q x_q$ that form a $\mathsf{LinCMSO}_2$ sentence (φ, f). Assume $X_1, \ldots, X_{q'}$ are vertex set variables and $X_{q'+1}, \ldots, X_q$ are edge set variables. We adapt the proof of Lemma 3.7.2 as follows: Let $\mathsf{CMSO}_2(X; X_1, \ldots, X_q)$ consist of all CMSO_2 sentences that can additionally use: elements of X as constants, and unary predicates $X_1(\cdot), \ldots, X_q(\cdot)$, where $X_1, \ldots, X_{q'}$ accept a single vertex as an argument, and $X_{q'+1}, \ldots, X_q$ accept a single edge as an argument. In the same way, we can compute a set $\mathrm{Sentences}^p(X; X_1, \ldots, X_q)$ consisting of at most $g(\ell, |X|, q)$ sentences of $\mathsf{CMSO}_2(X; X_1, \ldots, X_q)$ for some computable g so that for every $\psi \in \mathsf{CMSO}_2(X; X_1, \ldots, X_q)$ of rank at most p, there is $\psi' \in \mathrm{Sentences}^p(X; X_1, \ldots, X_q)$ equivalent to ψ; and the mapping $\psi \to \psi'$ is computable. Next, given a boundaried graph G with $X = \partial G$, vertex sets $A_1, \ldots, A_{q'} \subseteq V(G)$ and edge sets $A_{q'+1}, \ldots, A_q \subseteq E(G)$, define the p-type of $(G, A_1, \ldots, A_q)$ as follows:

$$\mathrm{tp}^p(G; A_1, \ldots, A_q) = \{\psi \in \mathrm{Sentences}^p(X; X_1, \ldots, X_q) \mid (G, A_1, \ldots, A_q) \models \psi\}.$$

Then we construct automaton $\mathcal{A} = \mathcal{A}_{\ell,(\varphi,f)}$ so that for every boundaried binary tree decomposition $(T, \mathsf{bags}, \mathsf{edges})$ of a graph G, the run of $\mathcal{A}$ on $(T, \mathsf{bags}, \mathsf{edges})$ is as follows. For every node x of T, $\rho_{\mathcal{A}}(x)$ is a function that maps every possible p-type $\tau \subseteq \mathrm{Sentences}^p(G; X_1, \ldots, X_q)$ to the value

$$\rho_{\mathcal{A}}(x)(\tau) = \max\left\{ c_0 + \sum_{i=1}^{q} c_i |A_i| \;\middle|\; A_1, \ldots, A_{q'} \subseteq V(G_x),\ A_{q'+1}, \ldots, A_q \subseteq E(G_x),\ \mathrm{tp}^p(G; A_1, \ldots, A_q) = \tau \right\},$$

where we place $\rho_{\mathcal{A}}(x)(\psi) = \bot$ if the set on the right-hand side of the equation above is empty.

As before, when constructing $\mathcal{A}$, we must check that one can define a transition mapping δ. Again it suffices to show that for every node x of T with children y and z, given $\rho_{\mathcal{A}}(y)$ and $\rho_{\mathcal{A}}(z)$ together with the information about the bag of x (consisting of $\mathsf{bag}(x)$, $\mathsf{adh}(x)$, $\mathsf{adh}(y)$, $\mathsf{adh}(z)$, and $\mathsf{edges}(x)$), one can compute $\rho_{\mathcal{A}}(x)$; and same for nodes with one child. Here, we use the fact that the objective function is affine in the following way. Suppose that the sets $A_1, \ldots, A_q$ are *optimal* for the graph G_x and sentence $\psi \in \mathrm{tp}^p(G; A_1, \ldots, A_q)$, that is, $A_1, \ldots, A_{q'} \subseteq V(G_x)$, $A_{q'+1}, \ldots, A_q \subseteq E(G_x)$, $(G, A_1, \ldots, A_q) \models \psi$ and $\rho_{\mathcal{A}}(x)(\psi) = c_0 + \sum_{i=1}^{q} c_i |A_i|$. Then, for each child $c \in \{y, z\}$ of x, the sets $A_1^c, \ldots, A_q^c$, defined as follows: $A_i^c = A_i \cap V(G_c)$ for $i \in [q']$ and $A_i^c = A_i \cap E(G_c)$ for $i \in [q'+1, q]$ are optimal for the graph G_c. For more details we refer to [ALS91]. □

Bodlaender-Kloks automaton. In [BK96], Bodlaender and Kloks gave an algorithm that given a graph G, a binary tree decomposition of G of width at most ℓ, and a number $k \leqslant \ell$, decides whether the treewidth of G is at most k in time $2^{\mathcal{O}(k\ell^2)} \cdot n$, where n is the vertex count of G. This algorithm proceeds by bottom-up dynamic programming on the provided tree decomposition of G, computing, for every node x, a table consisting of $2^{\mathcal{O}(k\ell^2)}$ boolean entries. Intuitively, each entry encodes the possibility of constructing a partial tree decomposition of the subgraph induced by the subtree at x with a certain "signature" on the adhesion of x; the number of possible signatures is $2^{\mathcal{O}(k\ell^2)}$.

Inspecting the proof provided in [BK96] it is not hard to see that this dynamic programming can be understood as a nondeterministic tree decomposition automaton. Thus, from the work of Bodlaender and Kloks we can immediately deduce the following statement.

Lemma 3.7.4. *For every pair of integers $k \leqslant \ell$ there is a nondeterministic tree decomposition automaton $\mathcal{BK}_{k,\ell}$ of width ℓ with the following property: For any graph G and its binary annotated tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of width at most ℓ, $\mathcal{BK}_{k,\ell}$ accepts $(T, \mathsf{bag}, \mathsf{edges})$ if and only if the treewidth of G is at most k. The state space of $\mathcal{BK}_{k,\ell}$ is of size $2^{\mathcal{O}(k\ell^2)}$ and can be computed in time $2^{\mathcal{O}(k\ell^2)}$. The evaluation time of $\mathcal{BK}_{k,\ell}$ is $2^{\mathcal{O}(k\ell^2)}$ as well.*

We remark that since the property of having treewidth at most k can be expressed in CMSO_2[12], Lemma 3.7.4 with an unspecified bound on the evaluation time also follows from Lemma 3.7.2. The reason behind formulating Lemma 3.7.4 explicitly is to keep track of the evaluation time more precisely in further arguments.

Closure automaton. Finally, we introduce automata for computing small closures within subtrees of a tree decomposition. These will be used in the proof of Lemma 3.3.23. We first need a few definitions.

Let G be a boundaried graph. We say that two sets of non-boundary vertices $Y, Z \subseteq V(G) \setminus \partial G$ are *torso-equivalent* if there is an isomorphism between $\mathsf{torso}_G(Y \cup \partial G)$ and $\mathsf{torso}_G(Z \cup \partial G)$ that fixes every vertex of ∂G. Note that being torso-equivalent is an equivalence relation and for every integer c, let $\sim_{c,G}$ be the restriction of this equivalence relation to subsets of $V(G) \setminus \partial G$ of cardinality at most c. Note $\sim_{c,G}$ has at most $2^{\mathcal{O}((c+|\partial G|)^2)}$ equivalence classes.

Suppose further that $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ is a boundaried tree decomposition of G. We generalize the depth function $d_{\mathcal{T}}$ defined in Section 3.2 to boundaried tree decompositions as follows: Let $d_{\mathcal{T}} \colon V(G) \setminus \partial G \to \mathbb{Z}_{\geqslant 0}$ be such that $d_{\mathcal{T}}(u)$ is the depth of the top-most node of T whose bag contains u. In particular, $d_{\mathcal{T}}(u)$ depends on the vertex u and the tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$. Further, for a set of vertices $Y \subseteq V(G) \setminus \partial G$, we define $d_{\mathcal{T}}(Y) = \sum_{u \in Y} d_{\mathcal{T}}(u)$. Recalling that there is a total order $\preccurlyeq$ on the vertices of G inherited from Ω, subsets of $V(G) \setminus \partial G$ can be compared as follows: For $Y, Y' \subseteq V(G) \setminus \partial G$, we set $Y \preccurlyeq_{\mathcal{T}} Y'$ if

- $d_{\mathcal{T}}(Y) < d_{\mathcal{T}}(Y')$, or
- $d_{\mathcal{T}}(Y) = d_{\mathcal{T}}(Y')$ and Y is lexicographically not larger than Y' with respect to $\preccurlyeq$.

For a nonempty set of subsets $\mathcal{S}$ of $V(G) \setminus \partial G$, we let $\min_{\mathcal{T}} \mathcal{S}$ be the $\preccurlyeq_{\mathcal{T}}$-smallest element of $\mathcal{S}$. This allows us to define the *c-small torso representatives* as follows:

$$\mathsf{reps}^c(G, (T, \mathsf{bag}, \mathsf{edges})) :=$$
$$\{(d_{\mathcal{T}}(\min_{\mathcal{T}} \mathcal{K}), \mathsf{torso}_G(\min_{\mathcal{T}} \mathcal{K} \cup \partial G)) \mid \mathcal{K} \text{ is an equivalence class of } \sim_{c,G}\}.$$

Note that each member of an equivalence class of $\sim_{c,G}$ has the same size, so at this point we do not optimize for the size even though it will later be needed for the proof of Lemma 3.3.23. The set $\mathsf{reps}^c(G, (T, \mathsf{bag}, \mathsf{edges}))$ depends on both the boundaried graph G and its boundaried tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$. Also, the cardinality of $\mathsf{reps}^c(G, (T, \mathsf{bag}, \mathsf{edges}))$ is equal to the number of equivalence classes of $\sim_{c,G}$, which, as noted, is at most $2^{\mathcal{O}((c+|\partial G|)^2)}$.

The closure automata we are going to use are provided by the following statement.

Lemma 3.7.5. *For every pair of integers c, ℓ there is a tree decomposition automaton $\mathcal{R} = \mathcal{R}_{c,\ell}$ with the following property: For any graph G and its annotated binary tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of width at most ℓ, the run of $\mathcal{R}$ on $(T, \mathsf{bag}, \mathsf{edges})$ satisfies*

$$\rho_{\mathcal{R}}(x) = \mathsf{reps}^c(G_x, (T_x, \mathsf{bag}_x, \mathsf{edges}_x)) \qquad \text{for all } x \in V(T).$$

[12] This can be done, for instance, by stating that the given graph does not contain any of the forbidden minor obstructions for having treewidth at most k. It is known that the sizes of such obstructions are bounded by a doubly-exponential function in k^5, hence their number is at most triply-exponential in k^5 [Lag98].

The evaluation time of $\mathcal{R}$ is $2^{\mathcal{O}((c+\ell)^2)}$.

Proof. For the state space Q of $\mathcal{R}$ we take the set of all sets of pairs of the form (p, H), where $p \in \mathbb{N}$ and H is a graph on at most $c+\ell+1$ vertices contained in Ω. The final states of $\mathcal{R}$ are immaterial for the lemma statement, hence we can set $F = \emptyset$. As for the initial mapping ι, for a boundaried graph H on at most $\ell+1$ vertices we can set $\iota(H) = \mathsf{reps}^c(H, (T_0, \mathsf{bags}_0, \mathsf{edges}_0))$, where $(T_0, \mathsf{bags}_0, \mathsf{edges}_0)$ is the trivial one-node tree decomposition of H in which all vertices and edges are put in the root bag. It is straightforward to see that $\iota(H)$ can be computed in time $2^{\mathcal{O}((c+\ell)^2)}$ directly from the definition.

It remains to define the transition mapping δ. For this, it suffices to prove the following. Suppose $(T, \mathsf{bags}, \mathsf{edges})$ is a boundaried tree decomposition of a boundaried graph G and x is a node of G with children y and z. Then knowing

$$R_y := \mathsf{reps}^c(G_y, (T_y, \mathsf{bags}_y, \mathsf{edges}_y)), \qquad R_z := \mathsf{reps}^c(G_z, (T_z, \mathsf{bags}_z, \mathsf{edges}_z)),$$

as well as $\mathsf{bag}(x)$, $\mathsf{edges}(x)$, and adhesions of x, y, z, one can compute

$$R_x := \mathsf{reps}^c(G_x, (T_x, \mathsf{bags}_x, \mathsf{edges}_x))$$

in time $2^{\mathcal{O}((c+\ell)^2)}$. Formally, we would also need such an argument for the case when x has only one child y, but this follows from the argument for the case of two children by considering a dummy second child z with an empty bag. So we focus only on the two-children case.

For any pair of sets $Y \subseteq \mathsf{cmp}(y)$ and $Z \subseteq \mathsf{cmp}(z)$, define $G_x(Y, Z)$ to be the graph with vertex set $Y \cup Z \cup \mathsf{bag}(x)$ and edge set consisting of the union of the edge sets of the following graphs:

$$\mathsf{torso}_{G_y}(Y \cup \mathsf{adh}(y)), \qquad \mathsf{torso}_{G_z}(Z \cup \mathsf{adh}(z)), \qquad \text{and} \qquad (\mathsf{bag}(x), \mathsf{edges}(x)).$$

The following claim is straightforward.

Claim 3.7.6. *For any triple of sets $X \subseteq \mathsf{bag}(x) \setminus \mathsf{adh}(x)$, $Y \subseteq \mathsf{cmp}(y)$ and $Z \subseteq \mathsf{cmp}(z)$, we have*

$$\mathsf{torso}_{G_x}(X \cup Y \cup Z \cup \mathsf{adh}(x)) = \mathsf{torso}_{G_x(Y,Z)}(X \cup Y \cup Z \cup \mathsf{adh}(x)).$$

To compute R_x, we first construct a family of candidates C as follows. Consider every pair of pairs $(p_y, H_y) \in R_y$ and $(p_z, H_z) \in R_z$, and every $X \subseteq \mathsf{bag}(x) \setminus \mathsf{adh}(x)$. Let $Y = V(H_y) \setminus \mathsf{adh}(y)$ and $Z = V(H_z) \setminus \mathsf{adh}(z)$, and note that the graph $G_x(Y, Z)$ is the union of graphs H_y, H_z, and $(\mathsf{bag}(x), \mathsf{edges}(x))$. If $|X \cup Y \cup Z| \leqslant c$, then we add to C the pair

$$(p_y + p_z + |Y| + |Z|, \mathsf{torso}_{G_x(Y,Z)}(X \cup Y \cup Z \cup \mathsf{adh}(x))).$$

Otherwise, if $|X \cup Y \cup Z| > c$, no pair is added to C. The first coordinate of the pair added to C is equal to $d_{T_x}(X \cup Y \cup Z)$ because $X \subseteq \mathsf{bag}(x)$ and $Y, Z \subseteq V(G_x) \setminus \mathsf{bag}(x)$. By Claim 3.7.6, the second coordinate of the pair added to C is equal to $\mathsf{torso}_{G_x}(X \cup Y \cup Z \cup \mathsf{adh}(x))$.

Further, we have

$$|C| \leqslant |R_y| \cdot |R_z| \cdot 2^{\ell+1} \leqslant 2^{\mathcal{O}((c+\ell)^2)},$$

and C can be computed in time $2^{\mathcal{O}((c+\ell)^2)}$.

Next, the candidates are filtered as follows. As long as in C there is are distinct pairs (p, H) and (p', H') such that H and H' are isomorphic by an isomorphism that fixes $\mathsf{adh}(x)$, we remove the pair that has the larger first coordinate; if both pairs have the same first coordinate, remove the one where the vertex set of the second coordinate is larger in $\preccurlyeq$. Clearly, this filtering procedure can be performed exhaustively in time $2^{\mathcal{O}((c+\ell)^2)}$.

Thus, after filtering, all second coordinates of the pairs in C are pairwise nonequivalent in $\sim_{c,G_x}$. It is now straightforward to see using a simple exchange argument that R_x is equal to C after the filtering. This constitutes the definition and the algorithm computing the transition function δ. □

3.7.3 Dynamic maintenance of automata runs

Having defined the automata we are going to use, we now show how to maintain their runs effectively under prefix-rebuilding updates.

Lemma 3.7.7. *Fix $\ell \in \mathbb{N}$ and a tree decomposition automaton $\mathcal{A} = (Q, F, \iota, \delta)$ of width ℓ and evaluation time τ. Then there exists an ℓ-prefix-rebuilding data structure with overhead τ that additionally implements the following operation:*

- $\mathsf{Query}(x)$: *Given a node x of T, returns $\rho_{\mathcal{A}}(x)$. Runs in worst-case time $\mathcal{O}(1)$.*

Proof. At every point in time, the data structure stores the decomposition $(T, \mathsf{bag}, \mathsf{edges})$, where every node x is supplied with a pointer to its parent, a pair of pointers to its children, and the state $\rho_{\mathcal{A}}(x)$ in the run of $\mathcal{A}$ on $(T, \mathsf{bag}, \mathsf{edges})$. This allows for answering queries in constant time, as requested.

For initialization, we just compute the run of $\mathcal{A}$ on $(T, \mathsf{bag}, \mathsf{edges})$ in a bottom-up manner: The states for leaves are computed according to the initialization mapping, while the states for internal nodes are computed according to the transition mapping bottom-up. This requires time τ per node, so $\mathcal{O}(\tau \cdot |V(T)|)$ in total.

For applying a prefix-rebuilding update $\overline{u} = (T_{\mathrm{pref}}, T'_{\mathrm{pref}}, T^\star, \mathsf{bag}^\star, \mathsf{edges}^\star, \pi)$, the pointer structure representing the decomposition can be easily rebuilt in time $\ell^{\mathcal{O}(1)} \cdot |\overline{u}|$ by building the tree T'_{pref} and reattaching all appendices of T_{pref} in T according to π, using a single pointer change per appendix. Observe here that the information about the run of $\mathcal{A}$ on the reattached subtrees does not need to be altered, except for the appendices of T'_{pref}, for which the run could have to be altered because their adhesions could change. Hence, it remains to compute the states associated with the nodes of $T'_{\mathrm{pref}} \cup \mathsf{App}(T'_{\mathrm{pref}})$ in the run of $\mathcal{A}$ on the new decomposition $(T', \mathsf{bag}', \mathsf{edges}')$. This can be done by processing $T'_{\mathrm{pref}} \cup \mathsf{App}(T'_{\mathrm{pref}})$ in a bottom-up manner, and computing each consecutive state using either the initialization mapping ι (for nodes in T'_{pref} that are leaves of T') or the transition mapping δ (for the other nodes in T'_{pref}), in total time $\mathcal{O}(\tau \cdot |T'_{\mathrm{pref}} \cup \mathsf{App}(T'_{\mathrm{pref}})|) \leqslant \mathcal{O}(\tau \cdot |\overline{u}|)$. □

Now, Lemma 3.7.7 combined with Lemma 3.7.2 immediately implies Lemma 3.2.4. Similarly, an ℓ-prefix-rebuilding data structure implementing the three first operations of Lemma 3.2.1 follows immediately by applying Lemma 3.7.7 to the automaton provided by Lemma 3.7.1. Let us here complete the proof of Lemma 3.2.1.

Proof of Lemma 3.2.1. By above discussion, it suffices to implement an ℓ-prefix-rebuilding data structure with overhead $\mathcal{O}(1)$ that implements the operation $\mathsf{Top}(v)$, that given a vertex $v \in V(G)$ returns the unique highest node t of T such that $v \in \mathsf{bag}(t)$. Consider a prefix-rebuilding update changing $(T, \mathsf{bag}, \mathsf{edges})$ to $(T', \mathsf{bag}', \mathsf{edges}')$. Observe that a prefix-rebuilding update can change the highest node where v occurs only if $v \in \mathsf{bags}_T(T_{\mathrm{pref}} \cup \mathsf{App}(T_{\mathrm{pref}}))$, and in particular, in that case the highest node of $(T', \mathsf{bag}', \mathsf{edges}')$ where v occurs will be in $T'_{\mathrm{pref}} \cup \mathsf{App}(T'_{\mathrm{pref}})$. Both $|T_{\mathrm{pref}} \cup \mathsf{App}(T_{\mathrm{pref}})|$ and $|T'_{\mathrm{pref}} \cup \mathsf{App}(T'_{\mathrm{pref}})|$ are linear in $|\overline{u}|$, so we simply maintain the mapping $\mathsf{Top}(v)$ explicitly by recomputing it for all vertices $v \in \mathsf{bags}_T(T_{\mathrm{pref}} \cup \mathsf{App}(T_{\mathrm{pref}}))$. □

3.7.4 Closures and blockages

We can now give a proof of Lemma 3.3.23. This will require more work, as apart from dynamically maintaining relevant information we also need to implement methods for extracting a closure and blockages.

Proof of Lemma 3.3.23. For the proof, we fix the following two automata:

- $\mathcal{R} = \mathcal{R}_{c,\ell}$ is the closure automaton for parameters c and ℓ, provided by Lemma 3.7.5.
- $\mathcal{BK} = \mathcal{BK}_{2k+1,c+\ell}$ is the Bodlaender-Kloks automaton for parameters $2k+1$ and $c+\ell$, provided by Lemma 3.7.4.

Let $\mathcal{BK} = (Q, F, \iota, \delta)$. Recall that $\mathcal{BK}$ is nondeterministic, $|Q| \leqslant 2^{\mathcal{O}(k(c+\ell)^2)}$, Q can be computed in time $2^{\mathcal{O}(k(c+\ell)^2)}$, and membership in F, ι, or δ for relevant objects can be decided in time $2^{\mathcal{O}(k(c+\ell)^2)}$.

Our data structure just consists of the data structure provided by Lemma 3.7.7 for the automaton $\mathcal{R}$. Thus, the initialization time is $2^{\mathcal{O}((c+\ell)^2)} \cdot |V(T)|$ and the update time is $2^{\mathcal{O}((c+\ell)^2)} \cdot |\overline{u}|$ as requested. It remains to implement method $\mathsf{Query}(T_{\mathrm{pref}})$. For this, we may assume that the stored annotated tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ is labeled with the run $\rho_{\mathcal{R}}$ of $\mathcal{R}$ on $(T, \mathsf{bag}, \mathsf{edges})$. This means that for every $x \in V(T)$, we have access to $\mathsf{reps}^c(T_x, \mathsf{bag}_x, \mathsf{edges}_x)$.

Consider a query $\mathsf{Query}(T_{\mathrm{pref}})$. We break answering this query into two steps. In the first step, we compute a $d_{\mathcal{T}}$-minimal c-small k-closure X of $\mathsf{bags}(T_{\mathrm{pref}})$, together with $\mathsf{torso}_G(X)$. This will take time $2^{\mathcal{O}(k(c+\ell)^2)} \cdot |T_{\mathrm{pref}}|$. In the second step, we find $\mathsf{Blockages}(T_{\mathrm{pref}}, X)$. This will take time $2^{\mathcal{O}((c+\ell)^2)} \cdot |\mathsf{Exploration}(T_{\mathrm{pref}}, X)|$.

Step 1: Finding the closure and its torso. Let A be the set of appendices of T_{pref}. For $a \in A$, let

$$R(a) := \mathsf{reps}^c(T_a, \mathsf{bag}_a, \mathsf{edges}_a);$$

recall that $R(a)$ is stored along with a in the data structure. Let Λ be the set of all mappings λ with domain A such that $\lambda(a) \in R(a)$ for all a. For $\lambda \in \Lambda$ and $a \in A$, let $d^\lambda(a)$ and $H^\lambda(a)$ be the first, respectively the second coordinate of $\lambda(a)$, and let $s^\lambda(a) = |V(H^\lambda(a)) \setminus \mathsf{adh}(a)|$. For $x \in V(T)$ we denote by $d_T(x)$ the depth of x in $(T, \mathsf{bag}, \mathsf{edges})$. For $\lambda \in \Lambda$ we define

$$\begin{aligned} H^\lambda &:= \left(\mathsf{bags}(T_{\mathrm{pref}}) \cup \bigcup_{a\in A} V(H^\lambda(a)), \mathsf{edges}(T_{\mathrm{pref}}) \cup \bigcup_{a\in A} E(H^\lambda(a))\right), \\ s^\lambda &:= \sum_{a\in A} s^\lambda(a), \\ d^\lambda &:= \sum_{a\in A} \left(d^\lambda(a) + d_T(a)\cdot s^\lambda(a)\right). \end{aligned}$$

Note that by the definition of reps^c, we have that

- $H^\lambda = \mathsf{torso}_G(V(H^\lambda))$,
- $s^\lambda = |V(H^\lambda)| - |\mathsf{bags}(T_{\mathrm{pref}})|$, and
- $d^\lambda = d_{\mathcal{T}}(V(H^\lambda)) - d_{\mathcal{T}}(\mathsf{bags}(T_{\mathrm{pref}}))$

for all $\lambda \in \Lambda$.

Further, for each $a \in A$, the set $R(a)$ comprises all possible nonisomorphic torsos that can be obtained by picking at most c vertices within $\mathsf{cmp}(a)$, and with each possible torso $R(a)$ stores a realization with the least possible total depth. This immediately implies the following statement.

Claim 3.7.8. *Let $\lambda \in \Lambda$ be such that the treewidth of H^λ is at most $2k+1$ and, among such mappings λ, s^λ is minimum, and among those d^λ is minimum. Then $V(H^\lambda)$ is a $d_{\mathcal{T}}$-minimal c-small k-closure of $\mathsf{bags}(T_{\mathrm{pref}})$. Further, if no λ as above exists, then there is no c-small k-closure of $\mathsf{bags}(T_{\mathrm{pref}})$.*

So, by Claim 3.7.8, it suffices to find $\lambda \in \Lambda$ that primarily minimizes s^λ and secondarily d^λ such that H^λ has treewidth at most k, or conclude that no such λ exists. Indeed, then we can output $X := V(H^\lambda)$ and $\mathsf{torso}_G(X) = H^\lambda$ as the output to the query. Note here that once a mapping λ as above is found, one can easily construct H^λ in time $(\ell+c)^{\mathcal{O}(1)} \cdot |T_{\mathrm{pref}}|$ right from the definition. Intuitively, to find a suitable λ we analyze possible runs of the Bodlaender-Kloks automaton $\mathcal{BK}$ on the natural tree decomposition of H^λ inherited from T_{pref}, for different choices of λ.

Let $S = T_{\mathrm{pref}} \cup A$. For $x \in S$ let S_x be the subset of S consisting of descendants of x and $A_x = S_x \cap A$, and Λ_x be defined just like Λ, but for domain A_x instead of A. For $x \in V(T)$, let $T_{\mathrm{pref},x} \subseteq T_{\mathrm{pref}}$ consist of the nodes in T_{pref} that are descendants of x. For $x, y \in V(T)$, let $d_T(x,y)$ denote their distance in T. Further, for $\lambda \in \Lambda_x$, define

$$\begin{aligned} H^\lambda_x &:= \left(\mathsf{bags}(T_{\mathrm{pref},x}) \cup \bigcup_{a\in A_x} V(H^\lambda(a)), \mathsf{edges}(T_{\mathrm{pref},x}) \cup \bigcup_{a\in A_x} E(H^\lambda(a)) \setminus \binom{\mathsf{adh}(x)}{2}\right), \\ s^\lambda_x &:= \sum_{a\in A_x} s^\lambda(a), \\ d^\lambda_x &:= \sum_{a\in A_x} \left(d^\lambda(a) + d_T(x,a)\cdot s^\lambda(a)\right). \end{aligned}$$

We treat H^λ_x as a boundaried graph with boundary $\mathsf{adh}(x)$. Then, H^λ_x has a boundaried tree decomposition $(T^\lambda_x, \mathsf{bag}^\lambda_x, \mathsf{edges}^\lambda_x)$ naturally inherited from $(T, \mathsf{bag}, \mathsf{edges})$ as follows:

- $T^\lambda_x = T[S_x]$;
- $\mathsf{bag}^\lambda_x(y) = \mathsf{bag}(y)$ for all $y \in T_{\mathrm{pref},x}$, and $\mathsf{bag}^\lambda_x(a) = V(H^\lambda(a))$ for all $a \in A_x$;
- $\mathsf{edges}^\lambda_x(y) = \mathsf{edges}(y) \cup \bigcup_{a\in A_y} E(H^\lambda(a)) \cap \left(\binom{\mathsf{bag}(y)}{2} \setminus \binom{\mathsf{adh}(y)}{2}\right)$ for all $y \in T_{\mathrm{pref},x}$, and $\mathsf{edges}^\lambda_x(a) = E(H^\lambda(a)) \setminus \binom{\mathsf{adh}(a)}{2}$ for all $a \in A_x$.

Note that the width of this tree decomposition is at most $c + \ell$.

Let $\overline{\mathbb{Z}}$ denote the set of nonnegative integers together with $+\infty$. Let us use a total order on pairs in $\overline{\mathbb{Z}} \times \overline{\mathbb{Z}}$ where we first compare the first elements and if they are equal the second elements. In a bottom-up fashion, for every $x \in S$ we compute the mapping $\zeta_x \colon Q \times 2^{\binom{\mathsf{adh}(x)}{2}} \to \overline{\mathbb{Z}} \times \overline{\mathbb{Z}}$ defined as follows: For $q \in Q$ and $W \subseteq \binom{\mathsf{adh}(x)}{2}$, $\zeta_x(q, W)$ is the minimum value of $(s_x^\lambda, d_x^\lambda)$ among $\lambda \in \Lambda_x$ such that

- $\mathcal{BK}$ has a run on $(T_x^\lambda, \mathsf{bag}_x^\lambda, \mathsf{edges}_x^\lambda)$ in which x is labeled with q; and
- $\bigcup_{a \in A_x} E(H^\lambda(a)) \cap \binom{\mathsf{adh}(x)}{2} = W$.

In case there is no λ as above, we set $\zeta_x(q, W) = (+\infty, +\infty)$.

We now argue that the mappings ζ_x can be computed in a bottom-up manner. This follows from the following rules, whose correctness is straightforward.

- For every $a \in A$, $\zeta_a(q, W)$ is the minimum pair (s, d) such that there is $(d, H) \in R(a)$ with the following properties: $\left(H - \binom{\mathsf{adh}(a)}{2}, q\right) \in \iota$, $E(H) \cap \binom{\mathsf{adh}(a)}{2} = W$, and $|V(H) \setminus \mathsf{adh}(a)| = s$.
- For every $x \in T_{\mathrm{pref}}$ with no children, $\zeta_x(q, W) = (0, 0)$ if $(G_x, q) \in \iota$ and $W = \emptyset$, and $\zeta_x(q, W) = (+\infty, +\infty)$ otherwise.
- For every $x \in T_{\mathrm{pref}}$ with one child y, $\zeta_x(q, W)$ is the minimum (s, d) such that the following holds: There exist $q' \in Q$ and $W' \subseteq \binom{\mathsf{adh}(y)}{2}$ with $(s', d') = \zeta_y(q', W')$ such that

$$\left(\left(\mathsf{bag}(x), \mathsf{adh}(x), \mathsf{adh}(y), \emptyset, \mathsf{edges}(x) \cup W' \setminus \binom{\mathsf{adh}(x)}{2}, q', \bot\right), q\right) \in \delta,$$
$$W = W' \cap \binom{\mathsf{adh}(x)}{2},$$
$$s = s', \text{ and}$$
$$d = d' + s'.$$

  If there are no q', q'', W' as above, then $\zeta_x(q, W) = +\infty$.
- For every $x \in T_{\mathrm{pref}}$ with two children y and z, $\zeta_x(q, W)$ is the minimum (s, d) such that the following holds: There exist $q', q'' \in Q$, $W' \subseteq \binom{\mathsf{adh}(y)}{2}$, and $W'' \subseteq \binom{\mathsf{adh}(z)}{2}$ with $(s', d') = \zeta_y(q', W')$ and $(s'', d'') = \zeta_z(q'', W'')$ such that

$$\left(\left(\mathsf{bag}(x), \mathsf{adh}(x), \mathsf{adh}(y), \mathsf{adh}(z), \mathsf{edges}(x) \cup W' \cup W'' \setminus \binom{\mathsf{adh}(x)}{2}, q', q''\right), q\right) \in \delta,$$
$$W = (W' \cup W'') \cap \binom{\mathsf{adh}(x)}{2},$$
$$s = s' + s'', \text{ and}$$
$$d = d' + d'' + s' + s''.$$

  If there are no q', q'', W', W'' as above, then $\zeta_x(q, W) = +\infty$.

Using the rules above, all mappings ζ_x for $x \in S$ can be computed in total time $2^{\mathcal{O}(k(\ell+c)^2)} \cdot |T_{\mathrm{pref}}|$, because $|Q| \leqslant 2^{\mathcal{O}(k(\ell+c)^2)}$ and the evaluation time of $\mathcal{BK}$ is $2^{\mathcal{O}(k(\ell+c)^2)}$.

By the properties of $\mathcal{BK}$ asserted in Lemma 3.7.4, the minimum (s^λ, d^λ) among those $\lambda \in \Lambda$ for which H^λ has treewidth at most k is equal to $\min_{q \in F} \zeta_r(q, \emptyset)$, where r is the root of T. The latter minimum can be computed in time $2^{\mathcal{O}(k(\ell+c)^2)}$ knowing ζ_r. Finally, to find $\lambda \in \Lambda$ witnessing the minimum, it suffices to retrace the dynamic programming in the standard way. That is, when computing mappings ζ_x, for every computed value of $\zeta_x(q, W)$ we memorize how this value was obtained. After finding $q \in F$ that minimizes $\zeta_r(q, \emptyset)$ we recursively retrace how the value of $\zeta_r(q, \emptyset)$ was obtained along S in a top-down manner, up to values computed in the nodes of A; the ways in which these values were obtained give us the mapping λ. This concludes the construction of the closure $X = V(H^\lambda)$ and its torso H^λ.

Step 2: Finding blockages. Having constructed $X = V(H^\lambda)$ together with $\mathsf{torso}_G(X) = H^\lambda$, we proceed to finding the set $\mathsf{Blockages}(T_{\mathrm{pref}}, X)$. We may assume that every vertex of X has been marked as belonging to X (which can be done after computing X in time $\mathcal{O}(|X|) \leqslant (c+\ell)^{\mathcal{O}(1)} \cdot |T_{\mathrm{pref}}|$), hence checking whether a given vertex belongs to X can be done in constant time.

First, we observe that for every node $x \in V(T) \setminus T_{\mathrm{pref}}$, we can efficiently find out the information about the behavior of X in the subtree rooted at x. Denote $X_x := X \cap \mathsf{cmp}(x)$.

Claim 3.7.9. *Given a node $x \in V(T) \setminus T_{\mathrm{pref}}$, one can compute X_x and $\mathsf{torso}_{G_x}(X_x \cup \mathsf{adh}(x))$ in time $2^{\mathcal{O}((c+\ell)^2)}$.*

Proof. For every $(d, H) \in \mathsf{reps}^c(T_x, \mathsf{bag}_x, \mathsf{edges}_x)$, call H a *candidate* if $V(H) \setminus \mathsf{adh}(x) \subseteq X$. Note that we can find all candidates in time $(c+\ell)^{\mathcal{O}(1)} \cdot |\mathsf{reps}^c(T_x, \mathsf{bag}_x, \mathsf{edges}_x)| \leqslant 2^{\mathcal{O}((c+\ell)^2)}$ by inspecting all elements of $\mathsf{reps}^c(T_x, \mathsf{bag}_x, \mathsf{edges}_x)$ one by one. Now, a simple exchange argument shows that $X_x \cup \mathsf{adh}(x)$ is equal to the largest (in terms of the number of vertices) candidate, and $\mathsf{torso}_{G_x}(X_x \cup \mathsf{adh}(x))$ is equal to this candidate. □

For technical purposes, we need to set up a simple data structure for checking adjacencies in H^λ. The idea is based on the notion of degeneracy and can be considered folklore.

Claim 3.7.10. *In time $(c+\ell)^{\mathcal{O}(1)} \cdot |T_{\mathrm{pref}}|$ one can set up a data structure that for given vertices $u, v \in X$, can decide whether u and v are adjacent in H^λ time $\mathcal{O}(\ell)$.*

Proof. The treewidth of H^λ is at most $2k+1 = \mathcal{O}(\ell)$, implying that we can in time $\mathcal{O}(k \cdot |X|)$ compute a total order $\preccurlyeq$ on X so that every vertex $u \in X$ has only at most $2k+1$ neighbors that are earlier than u in $\preccurlyeq$. Therefore, for every vertex u of X we construct a list $L(u)$ consisting of all neighbors of u that are earlier than u in $\preccurlyeq$. These lists can be constructed by examining the edges of H^λ one by one, and for every next edge uv, either appending u to $L(v)$ or v to $L(u)$, depending whether u or v is earlier in $\preccurlyeq$. Then whether given u and v are adjacent can be decided in time $\mathcal{O}(k)$ by checking whether u appears on $L(v)$ or v appears on $L(u)$. □

Next, for a node $x \in V(T)$, we define $\mathsf{profile}(x) \subseteq \binom{\mathsf{bag}(x)}{2}$ to be the set consisting of all pairs $\{u, v\} \subseteq \mathsf{bag}(x)$ such that in G there is path connecting u and v that is internally disjoint with X. (Note that this definition concerns u and v both belonging and not belonging to X.) Note that if $x \in T_{\mathrm{pref}}$, we have $\mathsf{bag}(x) \subseteq X$ and $\mathsf{profile}(x)$ consists of all edges of $\mathsf{torso}_G(X) = H^\lambda$ with both endpoints in $\mathsf{bag}(x)$. Consequently, for such x we can compute $\mathsf{profile}(x)$ in time $\ell^{\mathcal{O}(1)}$ using the data structure of Claim 3.7.10.

Next, we show that knowing the profile of a parent we can compute the profile of a child.

Claim 3.7.11. *Suppose x is the parent of y in T and $y \notin T_{\mathrm{pref}}$. Then given $\mathsf{profile}(x)$, one can compute $\mathsf{profile}(y)$ in time $2^{\mathcal{O}((c+\ell)^2)}$.*

Proof. Let J be the graph on vertex set $\mathsf{bag}(y)$ whose edge set is the union of

- $\mathsf{edges}(y)$;
- all edges present in $\mathsf{profile}(x)$ with both endpoints in $\mathsf{adh}(x)$; and
- all edges present in $\mathsf{torso}_{G_z}(X_z \cup \mathsf{adh}(z))$ that have both endpoints in $\mathsf{adh}(z)$, for every child z of y.

Note that J can be constructed in time $2^{\mathcal{O}((c+\ell)^2)}$ using Claim 3.7.9. Now, it is straightforward to see that $\mathsf{profile}(y)$ consists of all pairs $\{u, v\} \subseteq \mathsf{bag}(y)$ such that in J there is a path connecting u and v that is internally disjoint with $X \cap \mathsf{bag}(y)$. Using this observation, $\mathsf{profile}(y)$ can be now constructed in time $\ell^{\mathcal{O}(1)}$, because J has at most $\ell+1$ vertices. □

Having established Claim 3.7.11, we can finally describe the procedure that finds blockages. The procedure inspects the appendices $a \in A$ one by one, and upon inspecting a it finds all blockages that are descendants of a. To this end, we start a depth-first search in T_a from a. At all times, together with the node $x \in V(T_a)$ which is currently processed in the search, we also store $\mathsf{profile}(x)$. Initially, $\mathsf{profile}(a)$ can be computed using Claim 3.7.11, where the profile of the parent of a – which belongs to T_{pref} – can be computed in time $\ell^{\mathcal{O}(1)}$, as argued. When the search enters a node y from its parent x, $\mathsf{profile}(y)$ can be computed from $\mathsf{profile}(x)$ using Claim 3.7.11 again. Observe that knowing $\mathsf{profile}(x)$, it can be determined in time $(c+\ell)^{\mathcal{O}(1)}$ whether x is a blockage:

- If $\mathsf{bag}(x) \subseteq X$ and $\mathsf{profile}(x) = \binom{\mathsf{bag}(x)}{2}$, then x is a clique blockage.
- If $\mathsf{bag}(x) \cap X \neq \emptyset$ and there exists $u \in \mathsf{bag}(x) \setminus X$ such that $\{u, v\} \in \mathsf{profile}(x)$ for all $v \in \mathsf{bag}(x) \setminus \{u\}$, then x is a component blockage.

Consequently, if x is a blockage, we output x and do not pursue the search further. Otherwise, if x is not a blockage, the search recurses to the children of x.

Clearly, the total number of nodes visited by the search is bounded by $|\mathsf{Exploration}(T_{\mathrm{pref}}, X)|$, and for each node we use time $2^{\mathcal{O}((c+\ell)^2)}$. Hence, the algorithm outputs all blockages in total time $2^{\mathcal{O}((c+\ell)^2)}|\mathsf{Exploration}(T_{\mathrm{pref}}, X)|$. □

3.8 Proof of Theorems 1.3.1 and 1.3.3

In this section we complete the proof of Theorems 1.3.1 and 1.3.3.

Let us roughly sketch the proof of Theorem 1.3.1. Recall that by Lemma 3.2.5 there exists a data structure that, for an initially empty dynamic graph G, updated by edge insertions and deletions, maintains an annotated tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of G of width at most $6k+5$ under the promise that the treewidth of G never grows above k. Now we progressively improve the data structure to become more resilient to queries causing $\mathrm{tw}(G)$ to exceed k, and then add support for testing arbitrary CMSO_2 properties:

- The first improvement (Lemma 3.8.1) allows the data structure to detect the updates increasing the treewidth of G above k, which enables us to reject them (i.e., refuse to perform such updates and leave the state of the data structure intact). This is achieved by: (a) maintaining a dynamic graph G', equal to G after each update, and an additional instance of the data structure of Lemma 3.2.5 maintaining a tree decomposition of G' of width at most $6k+11$ under the promise that $\mathrm{tw}(G') \leqslant k+1$; (b) also maintaining the Bodlaender–Kloks automaton (Lemma 3.7.4) that, given the augmented tree decomposition of G' of width at most $6k+11$, verifies whether the treewidth of G is at most k; (c) applying each update to G' first, using the Bodlaender–Kloks automaton in order to verify whether $\mathrm{tw}(G') \leqslant k$ after the update, and, depending on whether this is the case, accepting or rejecting the update.

- The next improvement (Lemma 3.8.3) allows the data structure to actually process the updates that would increase $\mathrm{tw}(G)$ above k. At each point of time, the data structure will maintain the information on whether $\mathrm{tw}(G) \leqslant k$ and an annotated tree decomposition of the most recent snapshot of G of treewidth at most k. This is a fairly standard application of the technique of *delaying invariant-breaking insertions* by Eppstein et al. [EGIS96]. This improvement resolves the former assertion of Theorem 1.3.1.

- In order to resolve the latter assertion of the theorem (i.e., the maintenance of arbitrary CMSO_2 properties), we use the data structure of Lemma 3.8.3 along with the prefix-rebuilding data structure of Lemma 3.2.4.

All three steps should be considered standard and fairly straightforward; for instance, the first two steps are used to perform an analogous improvement to the data structure of Majewski et al. [MPS23] maintaining CMSO_2 properties of graphs with bounded feedback vertex number.

Lemma 3.8.1. *There is a data structure that for an integer $k \in \mathbb{N}$, fixed upon initialization, and a dynamic graph G, updated by edge insertions and deletions, maintains an annotated tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of G of width at most $6k+5$ using prefix-rebuilding updates, only accepting the updates if the treewidth of the updated graph is at most k. More precisely, at every point in time the graph has treewidth at most k and the data structure contains an annotated tree decomposition of G of width at most $6k+5$. The data structure can be initialized on k and an edgeless n-vertex graph G in time $2^{k^{\mathcal{O}(1)}} \cdot n$, and then every update:*

- *takes amortized time $2^{k^{\mathcal{O}(1)} \cdot \sqrt{\log n \log\log n}}$;*
- *if the treewidth of the graph after the update would be larger than k, then the update is rejected and* "Treewidth too large" *is returned;*
- *otherwise, the update is accepted and the data structure returns a sequence of prefix-rebuilding updates used to obtain the annotated tree decomposition of the new graph from $(T, \mathsf{bag}, \mathsf{edges})$.*

Proof. We implement the required data structure by setting up three instances of already existing data structures:

- $\mathbb{D}_k$: the data structure from Lemma 3.2.5 maintaining an annotated tree decomposition of a dynamic graph G_k of width at most $6k+5$ using prefix-rebuilding updates under the promise that $\mathrm{tw}(G_k) \leqslant k$;
- $\mathbb{D}_{k+1}$: an analogous data structure, maintaining an annotated tree decomposition of a dynamic graph G_{k+1} of width at most $6k+11$ using prefix-rebuilding updates under the promise that $\mathrm{tw}(G_{k+1}) \leqslant k+1$;
- $\mathbb{BK}$: a $(6k+11)$-prefix-rebuilding data structure with overhead $2^{\mathcal{O}(k^3)}$ maintaining whether the dynamic graph maintained by an annotated tree decomposition has treewidth at most k; an existence of such a data structure follows from the combination of Lemmas 3.7.4 and 3.7.7.

All the data structures are initialized with an edgeless graph on n vertices. Between the updates to the data structure, we maintain the following invariant: All three data structures store the description of the same dynamic graph of treewidth at most k; and moreover, the annotated tree decompositions stored by $\mathbb{D}_{k+1}$ and $\mathbb{BK}$ are identical – each prefix-rebuilding update performed by $\mathbb{D}_{k+1}$ is also applied to $\mathbb{BK}$. On each successful update, the data structure returns all prefix-rebuilding updates applied on the annotated tree decomposition stored in $\mathbb{D}_k$.

It remains to show how the updates are handled.

- Assume an edge uv is removed from the graph. Note that edge removals cannot increase the treewidth of the maintained graph, so the removal can be safely relayed to all data structures.
- When an edge uv is to be added to the graph, we first update $\mathbb{D}_{k+1}$ by adding the edge. As the addition of an edge to a graph may increase its treewidth by at most one, $\mathbb{D}_{k+1}$ will handle this update. If, after the update, $\mathbb{BK}$ confirms that the treewidth of the stored graph is still at most k, we accept the update by adding uv also to the graph stored by $\mathbb{D}_k$.

  On the other hand, if $\mathbb{BK}$ returns that the treewidth of the graph after the update is larger then k, then we reject the update: We update $\mathbb{D}_{k+1}$ by removing uv from the stored graph and return "Treewidth too large".[13]

It is easy to show that the updates maintain the prescribed invariants and that on each query to the data structure, $\mathbb{D}_k$ and $\mathbb{D}_{k+1}$ are updated a constant number of times, so the required time complexity bounds follow for these data structures. It also follows from Lemma 3.2.5 that the total size of prefix-rebuilding updates performed by $\mathbb{D}_{k+1}$ over the first q queries is $2^{k^{\mathcal{O}(1)}} \cdot n + 2^{k^{\mathcal{O}(1)} \cdot \sqrt{\log n \log\log n}} \cdot q$. As each update of $\mathbb{D}_{k+1}$ is also applied to $\mathbb{BK}$ (a prefix-rebuilding data structure with overhead $2^{\mathcal{O}(k^3)}$), the first q updates to the data structure are processed by $\mathbb{BK}$ in time $2^{k^{\mathcal{O}(1)}} \cdot n + 2^{k^{\mathcal{O}(1)} \cdot \sqrt{\log n \log\log n}} \cdot q$. Thus, the amortized time complexity of the data structure is proved. □

We now strengthen the data structure from Lemma 3.8.1 so as to accept the edge insertions increasing the treewidth of the graph above k. To this end, we use the aforementioned technique of postponing invariant-breaking insertions. Here, we use a formulation of the technique by Chen et al. [CCD+20].

Suppose U is a universe. A family of subsets $\mathcal{F} \subseteq 2^U$ is said to be *downward closed* if $F \in \mathcal{F}$ implies $E \in \mathcal{F}$ for all $E \subseteq F$. Assume that $\mathbb{F}$ is a data structure maintaining a dynamic subset $X \subseteq U$ under insertions and removals of elements. We say that $\mathbb{F}$:

- *strongly supports $\mathcal{F}$ membership* if $\mathbb{F}$ supports a boolean query $\mathsf{Member}()$ which returns whether $X \in \mathcal{F}$;
- *weakly supports $\mathcal{F}$ membership* if $\mathbb{F}$ maintains an invariant that $X \in \mathcal{F}$, rejecting the insertions that would break the invariant.

Then, the following statement provides a way of turning weak support for $\mathcal{F}$ membership into the strong support:

Lemma 3.8.2 ([CCD+20], Lemma 11.1)**.** *Suppose U is a universe and we have access to a dictionary $\mathbb{L}$ on U. Let $\mathcal{F} \subseteq 2^U$ be downward closed and suppose that there is a data structure $\mathbb{D}$ weakly supporting $\mathcal{F}$ membership.*

Then there is a data structure $\mathbb{D}'$ strongly supporting $\mathcal{F}$ membership, where each $\mathsf{Member}()$ query takes $\mathcal{O}(1)$ time and each update uses amortized $\mathcal{O}(1)$ time and amortized $\mathcal{O}(1)$ calls to operations on $\mathbb{L}$ and $\mathbb{D}$. Moreover, $\mathbb{F}'$ maintains an instance of $\mathbb{F}$ and whenever $\mathsf{Member}() = \mathsf{true}$, then $\mathbb{F}$ stores the same set $X \in \mathcal{F}$ as $\mathbb{F}'$.

[13] Note that even though the update is rejected in this case and the graphs stored by $\mathbb{D}_k$ and $\mathbb{D}_{k+1}$ do not change, the annotated tree decomposition stored by $\mathbb{D}_{k+1}$ may ultimately be modified by the update.

The last statement of Lemma 3.8.2 was not stated formally; however, it is immediate from the proof in [CCD+20].

We proceed to show how Lemma 3.8.2 can be applied to the data structure of Lemma 3.8.1.

Lemma 3.8.3. *There is a data structure that for an integer $k \in \mathbb{N}$, fixed upon initialization, and a dynamic graph G, updated by edge insertions and deletions, maintains:*

- *an annotated tree decomposition $(T, \mathsf{bag}, \mathsf{edges})$ of width at most $6k+5$ of the most recent snapshot of G of treewidth at most k;*
- *a boolean information on whether $\mathrm{tw}(G) \leqslant k$.*

The data structure can be initialized on k and an edgeless n-vertex graph G in time $2^{k^{\mathcal{O}(1)}} \cdot n$, and then every update:

- *takes amortized time $2^{k^{\mathcal{O}(1)}} \cdot \sqrt{\log n \log\log n}$;*
- *returns* "Treewidth too large" *if the treewidth of the graph after the update is larger than k; otherwise, returns the sequence of prefix-rebuilding updates used to modify $(T, \mathsf{bag}, \mathsf{edges})$.*

Proof. Following the notation of Chen et al., define the universe $U := \binom{V(G)}{2}$. Moreover, define a family $\mathcal{F} \subseteq 2^U$ as follows: $X \in \mathcal{F}$ if and only if the graph H with $V(H) = V(G)$ and $E(H) = X$ has treewidth at most k. It is immediate that $\mathcal{F}$ is downward closed. Let also $\mathbb{F}$ be the data structure of Lemma 3.8.1. Observe that $\mathbb{F}$ weakly supports $\mathcal{F}$: $\mathbb{F}$ handles edge additions and removals, accepting them exactly when the treewidth of the graph after the update is at most k. Therefore, by Lemma 3.8.2, there exists a data structure $\mathbb{F}'$ strongly supporting $\mathcal{F}$. Moreover, $\mathbb{F}'$ maintains an instance of $\mathbb{F}$ and whenever the treewidth of the maintained graph is at most k, then $\mathbb{F}$ stores the same graph as $\mathbb{F}'$.

Now, our data structure maintains the instances of $\mathbb{F}$ and $\mathbb{F}'$ as above and the sequence s of prefix-rebuilding updates applied to $\mathbb{F}$ since the last time the treewidth of G was k or less. Each update is relayed verbatim to $\mathbb{F}'$; in turn, $\mathbb{F}'$ updates $\mathbb{F}$ through edge insertions and deletions, causing $\mathbb{F}$ to rebuild its annotated tree decomposition using a sequence of prefix-rebuilding updates. Such updates are appended to s. If, after the update to our data structure, $\mathbb{F}'$ returns $\mathsf{Member}() = \mathsf{false}$, then we return "Treewidth too large". Otherwise, we return the sequence s of prefix-rebuilding updates and clear s before the next query.

It is straightforward to show that the described data structure satisfies all the requirements of the lemma. □

The proof of Theorem 1.3.1 is now direct:

Proof of Theorem 1.3.1. The first part of the statement of the theorem is immediate from Lemma 3.8.3. Now, assume we are given a CMSO_2 formula φ and we are to verify whether $G \models \varphi$ whenever $\mathrm{tw}(G) \leqslant k$. To this end, we invoke Lemma 3.2.4 and instantiate a $(6k+5)$-prefix-rebuilding data structure $\mathbb{M}$ with overhead $\mathcal{O}_{k,\varphi}(1)$ that can be queried whether $G \models \varphi$ in worst-case time $\mathcal{O}_{k,\varphi}(1)$. Now, whenever the data structure of Lemma 3.8.3 returns that $\mathrm{tw}(G) \leqslant k$, it returns a sequence of prefix-rebuilding updates, which we immediately forward to $\mathbb{M}$. Then, we query $\mathbb{M}$ to verify whether $G \models \varphi$.

From Lemma 3.8.3 it immediately follows that the total size of all prefix-rebuilding updates over the first q queries is at most $2^{k^{\mathcal{O}(1)}} \cdot n + 2^{k^{\mathcal{O}(1)}} \cdot \sqrt{\log n \log\log n} \cdot q$. Hence, $\mathbb{M}$ processes the first q queries in time $\mathcal{O}_{k,\varphi}(2^{k^{\mathcal{O}(1)}} \cdot n + 2^{k^{\mathcal{O}(1)}} \cdot \sqrt{\log n \log\log n} \cdot q)$, which satisfies the postulated amortized time complexity bounds. □

Also, Theorem 1.3.3 follows immediately by the same proof as above by additionally using a prefix-rebuilding data structure for $\mathsf{LinCMSO}_2$ from Lemma 3.7.3.

3.9 Conclusions

We presented a data structure for the dynamic treewidth problem that achieves amortized update time $2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$. The obvious open question is to improve this complexity. It is plausible that some optimization of the current approach could result in shaving off the $\sqrt{\log\log n}$ factor in the exponent, but complexity of the form $2^{\mathcal{O}_k(\sqrt{\log n})}$ seems inherent to the recursive approach presented in Section 3.5. Nevertheless, we conjecture that it should be possible to achieve update time that is polylogarithmic in n for every fixed k, that is, of the form $\log^{\mathcal{O}_k(1)} n$, or maybe even $\mathcal{O}_k(\log^c n)$ for some universal constant c.

The ultimate goal would be to get closer to the $\mathcal{O}(\log n)$ bound achieved by Bodlaender for the case $k = 2$ [Bod93a].

Apart from the above, we hope that our result may open new directions in the design of parameterized dynamic data structures. More precisely, dynamic programming on tree decompositions is used as a building block in multiple different techniques in algorithm design, so Theorem 1.3.1 may be useful for designing dynamic counterparts of those techniques. Here is a list some possible directions.

- Theorem 1.3.1 provides the dynamic variant of the most basic formulation of Courcelle's Theorem. It seems that dynamic variants of the optimization formulation and the counting formulation, due to Arnborg et al. [ALS91], just follow from combining the data structure of Theorem 1.3.1 with suitably constructed automata. It would be interesting to see whether Theorem 1.3.1 can be used also in the context of more general problems concerning CMSO_2 queries on graphs of bounded treewidth, for instance the problem of query enumeration; see e.g. [Bag06, KS13].

- As mentioned in the Introduction, *bidimensionality* is a basic technique used in the design of parameterized algorithms in planar graphs, or more generally, graphs excluding a fixed minor; see [CFK+15, Section 7.7]. As it is based on solving the problem efficiently on graphs of bounded treewidth, one could investigate whether a dynamic counterpart could be developed using Theorem 1.3.1.

- Related to the point above, it would be interesting to see whether Theorem 1.3.1 could be used to design a dynamic counterpart of Baker's technique [Bak94]. This thesis already contains such a counterpart (see Chapter 5 for the approximation schemes for Independent Set and Dominating Set in planar and apex-minor-free classes of graphs), but interestingly, the framework of dynamic treewidth is not used there at all. The complexity guarantees of the data structures in Chapter 5 are also worse than these promised by Theorem 1.3.1. Hence it would be interesting to understand whether the concepts of dynamic treewidth could be used to improve the running times of these approximation schemes.

- The *irrelevant vertex technique* is a classic principle in parameterized algorithms, based on iteratively deleting vertices from a graph while not changing the answer to the problem, until the graph in question has bounded treewidth and can be tackled directly using dynamic programming. A very recent result of Korhonen, Pilipczuk, and Stamoulis [KPS24] heavily utilizes our data structure for dynamic treewidth to efficiently implement this technique for the classical problems of Minor Testing and Vertex Disjoint Paths: They can detect whether an n-vertex, m-edge (static) graph contains a graph H as a minor in time $\mathcal{O}_H(n^{1+\varepsilon})$, and they can solve a (static) instance of Vertex Disjoint Paths with k terminal pairs in time $\mathcal{O}_k(m^{1+\varepsilon})$. Previously, the best known algorithms for both problems had quadratic dependency on the number of vertices in the graph.

  Still, while certainly challenging, it does not seem impossible to derive dynamic variants of some of the algorithms obtained using the irrelevant vertex technique. A concrete candidate here would be the Vertex Disjoint Paths problem on planar graphs, which admits a relatively simple and well-understood irrelevant vertex rule [AKK+17].

- *Meta-kernelization*, due to Bodlaender et al. [BFL+16], is a powerful meta-technique for obtaining small kernels for parameterized problems on topologically-constrained graphs. The fundamental concept in meta-kernelization is *protrusion*: a portion of the graph in question that induces a subgraph of bounded treewidth and communicates with the rest of the graph through a bounded-size interface. Kernelization algorithms obtained through meta-kernelization use reduction rules based on *protrusion replacement*: finding a large protrusion, understanding it using tree-decomposition-based dynamic programming, and replacing it with a small gadget of the same functionality. It would be interesting to see if in some basic settings, Theorem 1.3.1 could be applied in combination with meta-kernelization to obtain data structures for maintaining small kernels in dynamic graphs. Dominating Set on planar graphs would probably be the first problem to look into, as it admits a simple kernelization algorithm that predates (and inspired) meta-kernelization [AFN04]. We remark that dynamic kernelization has already been studied as a way to obtain dynamic parameterized data structures [AMV20, IO14].

Chapter 4

Dynamic rankwidth

This chapter presents the results of our work on dynamic rank decompositions [KS24]. Our main goal is the following result about computing rankwidth exactly. The result has been stated in the introductory part of the thesis; for convenience, we restate it below.

Theorem 1.3.6 ([KS24]). *There is an algorithm that, given an n-vertex m-edge graph G and an integer k, in time $\mathcal{O}_k(n \cdot 2^{\sqrt{\log n}\log\log n}) + \mathcal{O}(m)$, either outputs a rank decomposition of G of width at most k or determines that the rankwidth of G is larger than k. The algorithm also outputs a $(2^{k+1}-1)$-expression for cliquewidth of G within the same running time.*

Moreover, every fixed graph problem that can be expressed in $\mathsf{LinCMSO}_1$ can be solved in time $\mathcal{O}_k(n \cdot 2^{\sqrt{\log n}\log\log n}) + \mathcal{O}(m)$ on graphs of rankwidth k.

Theorem 1.3.6 improves upon the $\mathcal{O}_k(n^2)$ time algorithm of Fomin and Korhonen [FK22], and is a subpolynomial $2^{\sqrt{\log n}\log\log n} = n^{o(1)}$ factor away from concluding the long line of work on computing rankwidth in the setting where k is bounded [OS06, OS07, CO07, Oum08a, HO08, JKO21, FK22]. Moreover, if the average degree of the input graph is higher than $f(k) \cdot 2^{\sqrt{\log n}\log\log n}$ for the function $f(k)$ hidden by the $\mathcal{O}_k(\cdot)$-notation (which is a very natural case when we are interested in rankwidth), then our algorithm works in truly linear $\mathcal{O}(m)$ time.

As both the main ingredient in proving Theorem 1.3.6 and a contribution of independent interest, we give a data structure for efficiently maintaining rank decompositions of dynamic graphs under edge insertions and deletions, under the promise that the rankwidth of the graph never grows above a given parameter k. The data structure can also maintain any finite-state dynamic programming scheme on the rank decomposition. We formalize this by stating that it can maintain the value of any $\mathsf{LinCMSO}_1$ sentence on the graph. In particular, we prove the following.

Theorem 1.3.4 ([KS24]). *There is a data structure that is initialized with an integer k and an empty n-vertex dynamic graph G, and maintains a rank decomposition of G of width at most $4k$ under edge insertions and deletions, under the promise that the rankwidth of G never exceeds k. The initialization time is $\mathcal{O}_k(n\log^2 n)$ and the amortized update time is $2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$.*

Further, the data structure can be initialized with a $\mathsf{LinCMSO}_1$ sentence φ and it can maintain the value of φ on G. In such a case, the initialization time of the data structure is $\mathcal{O}_{k,\varphi}(n\log^2 n)$ and the amortized update time is $2^{\mathcal{O}_{k,\varphi}(\sqrt{\log n \log\log n})}$.

To prove Theorem 1.3.6, we use a bit more technical version of Theorem 1.3.4 where we assert that the rank decomposition held by the data structure is represented as an "annotated rank decomposition" (defined in Section 4.3). Even after this the proof of Theorem 1.3.6 requires nontrivial additional work.

As rankwidth generalizes treewidth, the high-level approach of our Theorem 1.3.4 is similar to the high-level approach of the data structure of Chapter 3 (yielding a similar running time). However, making the approach work for rankwidth requires developing an extensive amount of new machinery for rankwidth, with several new algorithmic and structural insights. We provide a comparison between techniques in Theorem 1.3.4 and in the result of Chapter 3 at the end of Section 4.1.1. We note that also formally speaking, Theorem 1.3.4 is a generalization the result of Chapter 3: The setting of treewidth and CMSO_2 logic can be reduced to the setting of rankwidth and CMSO_1 logic by considering instead of a graph G the graph G' obtained by subdividing every edge of G once and adding two degree-1 vertices adjacent to each non-subdivision vertex. Then every edge update of G can be simulated by two edge updates of G',

the rankwidth of G' is at most the treewidth of G plus one, and every CMSO_2 sentence about G can be translated into a CMSO_1 sentence about G'.

Recalling the framework of *edge update sentences* from Section 1.3.1, we generalize Theorem 1.3.4 to support *dense* graph updates as follows:

Theorem 1.3.5 ([KS24]). *The data structure of Theorem 1.3.4, when furthermore initialized with a given integer d, can also support the following operations:*

- $\mathsf{Update}(\overline{e})$*: Given an edge update sentence $\overline{e}$ of length at most d, either returns that the graph resulting from applying $\overline{e}$ to G would have rankwidth more than k, or applies $\overline{e}$ to update G. Runs in $|\overline{e}| \cdot 2^{\mathcal{O}_{k,d}(\sqrt{\log n \log\log n})}$ amortized time.*
- $\mathsf{LinCMSO}_1(\varphi, X_1, \ldots, X_p)$*: Given a $\mathsf{LinCMSO}_1$ sentence φ of length at most d with p free set variables and p vertex subsets $X_1, \ldots, X_p \subseteq V(G)$, returns the value of φ on $(G, X_1, \ldots, X_p)$. Runs in time $\mathcal{O}_d(1)$ if $X_1, \ldots, X_p = \emptyset$, and in time $\sum_{i=1}^{p} |X_i| \cdot 2^{\mathcal{O}_{k,d}(\sqrt{\log n \log\log n})}$ otherwise.*

We discuss further extensions of Theorems 1.3.4 and 1.3.6 in Section 4.11.

Organization. We start by giving an overview of our proofs in Section 4.1. We discuss additional notation and preliminary results in Section 4.2. Then, we give our framework of annotated rank decompositions and prefix-rebuilding operations in Section 4.3 (compare it to the treewidth counterparts from Section 3.2). In Section 4.4 we give the main tools for maintaining rank decompositions of dynamic graphs, although delaying significant ingredients to Sections 4.8 and 4.9; here, we rely on the tools already implemented for dynamic treewidth in Sections 3.3 to 3.5. In Section 4.5 we introduce rank decomposition automata – a rankwidth variant of tree decomposition automata from Section 3.7 – and give results about them, though deferring some standard results to Section 4.10. Then, in Section 4.6 we finish the proofs of Theorems 1.3.4 and 1.3.5, and in Section 4.7 we prove Theorem 1.3.6.

In the following part of the chapter we establish several auxiliary results related to rank decompositions that we use in the proofs of Theorems 1.3.4 to 1.3.6. In Section 4.8 we prove a result called the Dealternation Lemma for rankwidth – which is an analog of the Dealternation Lemma for treewidth by Bojańczyk and Pilipczuk ([BP22], see also Section 3.3.1) – and which is used in Section 4.4. In Section 4.9 we give results related to computing exact rankwidth by dynamic programming, which are used in Sections 4.4, 4.6 and 4.7. We believe that the results in Sections 4.8 and 4.9 are of independent interest. In Section 4.10, we show how rank decomposition automata can emulate cliquewidth decompositions (*k-expressions*) and argue that our formalism of automata in Section 4.5 can solve CMSO_1 verification and optimization problems efficiently. Finally, we conclude in Section 4.11.

4.1 Overview

In this section we give an overview of our algorithms. We start by giving an overview of the proof of Theorem 1.3.4 in Section 4.1.1. We omit many important ingredients, of which two major ones we overview in Sections 4.1.2 and 4.1.3. We overview the proof of Theorem 1.3.6 in Section 4.1.4.

4.1.1 Dynamic rankwidth

Suppose we maintain a dynamic n-vertex graph G under edge insertions and deletions, and the rankwidth of G is guaranteed to stay at most k. Our goal will be to maintain a *rooted rank decomposition* $\mathcal{T} = (T, \lambda)$ of G of width at most $4k$ and height at most $h = 2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$. Rooted rank decompositions are defined like rank decompositions, except the tree T is a rooted binary tree. Even the existence of rank decompositions with such parameters is not immediately obvious, but indeed Courcelle and Kanté [CK07] show that a rank decomposition of width k can be turned into a rooted rank decomposition of width at most $2k$ and height $\mathcal{O}(\log n)$.

It is essential for our algorithm to also maintain dynamic programming schemes on the rank decomposition. We need this to support both the $\mathsf{LinCMSO}_1$ queries and various internal operations of our data structure. We will formalize dynamic programming as automata processing the tree T, so that the state of a node can be computed in $\mathcal{O}_k(1)$ time from the states of its children. For this, we need to store additional information about the graph G in the rank decomposition, for which we next define *annotated rank decompositions*; compare this notion to annotated tree decompositions described in Section 3.2.

For an edge xy of the tree T, we denote by $\mathcal{L}(\mathcal{T})[\vec{xy}] \subseteq V(G)$ the vertices of G that are mapped to leaves of T closer to x than y. If $\mathcal{T}$ has width $\leqslant 4k$, there exists a set $R \subseteq \mathcal{L}(\mathcal{T})[\vec{xy}]$ with $|R| \leqslant 2^{4k}$ so that for every $v \in \mathcal{L}(\mathcal{T})[\vec{xy}]$ exists $r \in R$ so that $N(r) \setminus \mathcal{L}(\mathcal{T})[\vec{xy}] = N(v) \setminus \mathcal{L}(\mathcal{T})[\vec{xy}]$. We define that such R is a *representative* of $\mathcal{L}(\mathcal{T})[\vec{xy}]$, and a minimal such R a *minimal representative* of $\mathcal{L}(\mathcal{T})[\vec{xy}]$. An annotated rank decomposition stores for every oriented edge $\vec{xy}$ of T a minimal representative $\mathcal{R}(\vec{xy})$ of the set $\mathcal{L}(\mathcal{T})[\vec{xy}]$. It also stores for every edge xy of T the bipartite graph $\mathcal{E}(xy) = G[\mathcal{R}(\vec{xy}), \mathcal{R}(\vec{yx})]$, encoding the adjacencies between $\mathcal{L}(\mathcal{T})[\vec{xy}]$ and $\mathcal{L}(\mathcal{T})[\vec{yx}]$. Furthermore, for every (oriented) path xyz of length 3 in T, it stores the function $\mathcal{F}(xyz)\colon \mathcal{R}(\vec{xy}) \to \mathcal{R}(\vec{yz})$, mapping each $v \in \mathcal{R}(\vec{xy})$ to $r \in \mathcal{R}(\vec{yz})$ so that $N(r) \setminus \mathcal{L}(\mathcal{T})[\vec{yz}] = N(v) \setminus \mathcal{L}(\mathcal{T})[\vec{yz}]$. It can be shown that an annotated rank decomposition of G uniquely defines the graph G. Indeed in our algorithm we do not store the graph G explicitly; we only maintain an annotated rank decomposition of it.

Then, the slightly more formal definition of a *rank decomposition automaton* is that it is a tree automaton working on T, where the state of a node x can be computed from the states of its children and the annotations $\mathcal{R}, \mathcal{E}, \mathcal{F}$ around x in $\mathcal{O}_k(1)$ time. Note that the total size of the annotations around x is $\mathcal{O}_k(1)$. A significant part of our result is to show that various dynamic programming routines on rank decompositions and cliquewidth expressions can be formulated as rank decomposition automata, similar to how dynamic programming schemes on tree decompositions can be stated as tree decomposition automata (Section 3.7). Formal definitions about annotated rank decompositions are in Section 4.3 and about rank decomposition automata in Section 4.5.

After this detour to dynamic programming, let us return to the problem of dynamic maintenance of an annotated rank decomposition $\mathcal{T}$ of G. Suppose $\mathcal{T}$ has width at most $4k$ and height at most h, and there comes an update to insert or delete an edge between two vertices u and v, which turns G into G'. Let T_{pref} be the minimal prefix of T that contains $\lambda(u)$ and $\lambda(v)$. We have that $|T_{\mathrm{pref}}| \leqslant 2 \cdot h \leqslant 2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$. Now, we can turn $\mathcal{T}$ into an annotated rank decomposition of G' by only re-computing annotations inside T_{pref}, which can be done in $\mathcal{O}_k(|T_{\mathrm{pref}}|)$ time. The states of the maintained automata also need to be re-computed only for nodes in T_{pref}, which also works in $\mathcal{O}_k(|T_{\mathrm{pref}}|)$ time. Therefore, we manage to update $\mathcal{T}$ in the desired time bound. The only issue is that the width of $\mathcal{T}$ could increase in this process.

We observe that the width of $\mathcal{T}$ can increase only by one, and moreover, only the widths of edges of $\mathcal{T}$ inside T_{pref} can increase, so suppose now that $\mathcal{T}$ has width $4k+1$ and all edges of width $4k+1$ are inside T_{pref}. Since the edges of $\mathcal{T}$ incident to $\lambda(u)$ and $\lambda(v)$ have width at most 1 at all times, we can now remove both $\lambda(u)$ and $\lambda(v)$ from T_{pref} and maintain that all edges of width $4k+1$ are inside T_{pref}. To reduce the width, we design a *refinement operation*, that takes as input a prefix T_{pref} of $\mathcal{T}$ not containing any leaves of $\mathcal{T}$ such that all edges incident to a vertex outside of T_{pref} have width at most $4k$ and, in some sense, locally re-computes the decomposition for the prefix T_{pref}. More accurately, the goal is that applying refinement to T_{pref} reduces the width back to at most $4k$, assuming G has rankwidth at most k, and runs in amortized time proportional to $|T_{\mathrm{pref}}|$. The refinement can increase the height of $\mathcal{T}$ by $\mathcal{O}(\log n)$. To not allow the height of $\mathcal{T}$ to spiral out of control, we then use a *height reduction scheme*, that by repeatedly applying refinement decreases the height back to at most h.

A keen reader may observe that the blueprint of the data structure above closely resembles the implementation of the data structure for dynamic treewidth presented in Chapter 3. This intuition is, of course, correct; in fact, we borrow the implementation of the height reduction scheme verbatim from Section 3.5. On the other hand, the implementation of the refinement operation needs to be slightly amended for dynamic rankwidth since tree and rank decompositions are structurally slightly different: In a tree decomposition of a graph, each vertex of the original graph is placed in a connected subtree of the decomposition, but in the case of rank decomposition, vertices are stored only in the leaves of the decomposition. Hence some technical details – such as the definition of a closure or a refinement of a prefix of the decomposition – must change to accommodate rank decompositions.

Then we delve into the details of the refinement operation. Let T_{pref} be the given prefix, and let us define $\vec{\mathsf{App}}(T_{\mathrm{pref}})$ as the set of oriented edges $\vec{xy}$ of T with $x \notin T_{\mathrm{pref}}$ and $y \in T_{\mathrm{pref}}$. Then we say that a *closure* of T_{pref} is a partition $\mathcal{C}$ of $V(G)$, so that for each $C \in \mathcal{C}$ there exists $\vec{xy} \in \vec{\mathsf{App}}(T_{\mathrm{pref}})$ with $C \subseteq \mathcal{L}(\mathcal{T})[\vec{xy}]$. A rank decomposition of a closure $\mathcal{C}$ is a pair $\mathcal{T}^\star = (T^\star, \lambda^\star)$, where $T^\star$ is a cubic tree with $|\mathcal{C}|$ leaves and $\lambda^\star$ is a bijection from $\mathcal{C}$ to the leaves of $T^\star$. The width of $\mathcal{T}^\star$ can be naturally defined analogously to the definition for rank decompositions of graphs. Before giving any arguments about how to find closures $\mathcal{C}$ with desirable properties, let us describe how the refinement operation uses a closure $\mathcal{C}$ to transform $\mathcal{T}$ into a new annotated rank decomposition $\mathcal{T}'$.

Assume $\mathcal{T}^\star$ is a rank decomposition of $\mathcal{C}$ of width at most $2k$. We use the method of [CK07] to turn $\mathcal{T}^\star$ into a rooted rank decomposition $\mathcal{T}^{\star\star}$ of width at most $4k$ and height at most $\mathcal{O}(\log n)$. Then, for each $C \in \mathcal{C}$, we construct from $\mathcal{T}$ a rooted rank decomposition $\mathcal{T}^C$ with $|C|$ leaves corresponding to C

by repeatedly deleting all leaves not corresponding to vertices in C and contracting degree-2 nodes. In particular, if $\mathcal{T}$ contains an edge xy with $\mathcal{L}(\mathcal{T})[\vec{xy}] \cap C \neq \emptyset$ and $\mathcal{L}(\mathcal{T})[\vec{yx}] \cap C \neq \emptyset$, then $\mathcal{T}^C$ contains an edge $x'y'$ with $\mathcal{L}(\mathcal{T})[\vec{x'y'}] = \mathcal{L}(\mathcal{T})[\vec{xy}] \cap C$ and $\mathcal{L}(\mathcal{T})[\vec{y'x'}] = \mathcal{L}(\mathcal{T})[\vec{yx}] \cap C$ (and all of edges of $\mathcal{T}^C$ are like that). Then we construct $\mathcal{T}'$ by taking $\mathcal{T}^{\star\star}$ and identifying the root of each $\mathcal{T}^C$ with the leaf of $\mathcal{T}^{\star\star}$ corresponding to C. Without assuming anything about $\mathcal{C}$, we can deduce that the height of $\mathcal{T}'$ is at most $\mathcal{O}(\log n)$ more than the height of $\mathcal{T}$ and all edges of $\mathcal{T}'$ corresponding to edges of $\mathcal{T}^{\star\star}$ have width at most $4k$. However, we do not know anything about the widths of edges not in $\mathcal{T}^{\star\star}$, and we do not know how this transformation could be implemented efficiently.

The first requirement for efficient implementation of this transformation is that $|\mathcal{C}|$ is not too large. We prove the existence of a closure with an even stronger property. We say that $\mathcal{C}$ is a *k-closure* if the rankwidth of $\mathcal{C}$ is at most $2k$. We also say that $\mathcal{C}$ is *c-small* if for all $\vec{xy} \in \vec{\mathsf{App}}(T_{\mathrm{pref}})$ there are at most c parts $C \in \mathcal{C}$ with $C \subseteq \mathcal{L}(\mathcal{T})[\vec{xy}]$. We prove the following lemma in Section 4.4; compare it to the Small Closure Lemma for treewidth (Lemma 3.3.2).

Lemma 4.1.1 (Lemma 4.4.2)**.** *For every $k, \ell \in \mathbb{N}$ there exists $c \in \mathbb{N}$ so that if $\mathcal{T}$ has width at most ℓ and G has rankwidth at most k, then for any prefix T_{pref} of $\mathcal{T}$ there exists a c-small k-closure $\mathcal{C}$ of T_{pref}.*

As $|\vec{\mathsf{App}}(T_{\mathrm{pref}})| \leqslant \mathcal{O}(|T_{\mathrm{pref}}|)$, this implies $|\mathcal{C}| \leqslant \mathcal{O}_k(|T_{\mathrm{pref}}|)$. In Section 4.1.2, we will highlight the main tool we develop for proving Lemma 4.1.1 – the *Dealternation Lemma* for rankwidth, analogous to the Dealternation Lemma for treewidth of Bojańczyk and Pilipczuk ([BP22], also cited in a slightly weaker form in Lemma 3.3.3) – and overview its proof, which is fully presented in Section 4.8.

We then require one more property of $\mathcal{C}$, which will be useful in both addressing the issue of the widths of the edges of $\mathcal{T}'$ coming from the decompositions $\mathcal{T}^C$ and in the efficient implementation of the refinement. For a node x of $\mathcal{T}$ with parent p we denote $\mathcal{L}(\mathcal{T})[x] = \mathcal{L}(\mathcal{T})[\vec{xp}]$, and when x is the root $\mathcal{L}(\mathcal{T})[x] = V(G)$. We say that $\mathcal{C}$ *cuts* a node x of $\mathcal{T}$ if more than one part in $\mathcal{C}$ intersects $\mathcal{L}(\mathcal{T})[x]$, and define $\mathsf{cut}(\mathcal{C})$ to be the set of nodes cut by $\mathcal{C}$. Note that $\mathsf{cut}(\mathcal{C})$ is a prefix of T and $T_{\mathrm{pref}} \subseteq \mathsf{cut}(\mathcal{C})$. We define that $\mathcal{C}$ is a *minimal* c-small k-closure if among all c-small k-closures of T_{pref}, $\mathcal{C}$ primarily minimizes $\sum_{C \in \mathcal{C}} \mathsf{cutrk}(C)$ and secondarily minimizes $|\mathsf{cut}(\mathcal{C})|$. Here, $\mathsf{cutrk}(C)$ denotes the rank of the $|C| \times |V(G) \setminus C|$ matrix describing adjacencies between C and $V(G) \setminus C$. We again encourage the reader to compare the definition of minimal closures above to its treewidth analog in Section 3.3.2 (Definition 5).

We observe that if $\mathcal{T}$ has a node $x \in V(T) \setminus T_{\mathrm{pref}}$ and $C \in \mathcal{C}$ intersects $\mathcal{L}(\mathcal{T})[x]$, then in $\mathcal{T}'$ there is a node x^C with $\mathcal{L}(\mathcal{T}')[x^C] = \mathcal{L}(\mathcal{T})[x] \cap C$, and moreover all nodes of $\mathcal{T}'$ coming from the decompositions $\mathcal{T}^C$ can be characterized like this. Therefore, to prove that $\mathcal{T}'$ has width at most $4k$, it suffices to prove that for all such x it holds that $\mathsf{cutrk}(\mathcal{L}(\mathcal{T})[x] \cap C) \leqslant 4k$. We prove the following stronger statement in Section 4.4.

Lemma 4.1.2 (Lemmas 4.4.3 and 4.4.6)**.** *Let $\mathcal{C}$ be a minimal c-small k-closure of T_{pref}, $C \in \mathcal{C}$, and $x \in V(T) \setminus T_{\mathrm{pref}}$ with $\mathcal{L}(\mathcal{T})[x] \cap C \neq \emptyset$. Then $\mathsf{cutrk}(\mathcal{L}(\mathcal{T})[x] \cap C) \leqslant \mathsf{cutrk}(\mathcal{L}(\mathcal{T})[x])$, with equality only if $\mathcal{L}(\mathcal{T})[x] \subseteq C$.*

The statement above relates to the Closure Linkedness Lemma (Lemma 3.3.7) from dynamic treewidth.

As $\mathsf{cutrk}(\mathcal{L}(\mathcal{T})[x]) \leqslant 4k$ for $x \in V(T) \setminus T_{\mathrm{pref}}$, this implies that $\mathcal{T}'$ has width at most $4k$ when $\mathcal{C}$ is minimal. The proof of Lemma 4.1.2 makes use of the submodularity of the cutrk function. It can be considered to be a rankwidth analog of the techniques developed for improving tree decompositions by Korhonen and Lokshtanov [KL23]. Let us then assume that $\mathcal{C}$ is a minimal c-small k-closure.

We then use the fact that in Lemma 4.1.2 the equality holds only if $\mathcal{L}(\mathcal{T})[x] \subseteq C$. The nodes of $\mathcal{T}$ can be partitioned into three groups based on T_{pref} and $\mathcal{C}$ – those in T_{pref}, those in $\mathsf{cut}(\mathcal{C}) \setminus T_{\mathrm{pref}}$, and those in $V(T) \setminus \mathsf{cut}(\mathcal{C})$. If $x \in V(T) \setminus \mathsf{cut}(\mathcal{C})$, then there exists $C \in \mathcal{C}$ so that $\mathcal{L}(\mathcal{T})[x] \subseteq C$. In this case the resulting decomposition $\mathcal{T}'$ will contain the exactly same subtree rooted at x as $\mathcal{T}$, so maximal such subtrees can be copied from $\mathcal{T}$ to $\mathcal{T}'$ by changing just one pointer, copying also the annotations and the automata states, and the number of them is $\mathcal{O}(|\mathsf{cut}(\mathcal{C})|)$. If $x \in \mathsf{cut}(\mathcal{C}) \setminus T_{\mathrm{pref}}$, then a node x^C corresponding to x is constructed for every $C \in \mathcal{C}$ that intersects $\mathcal{L}(\mathcal{T})[x]$. Because $\mathcal{C}$ is c-small, there are at most c such nodes x^C, and by Lemma 4.1.2 for all of them it holds that $\mathsf{cutrk}(\mathcal{L}(\mathcal{T}')[x^C]) < \mathsf{cutrk}(\mathcal{L}(\mathcal{T})[x])$. Thus, we can think that we replace each node in $\mathsf{cut}(\mathcal{C}) \setminus T_{\mathrm{pref}}$ by at most c nodes that each has smaller width (the width of a node is the width of the edge between it and its parent), which motivates to use the following potential function for amortized analysis:

$$\Phi(\mathcal{T}) = \sum_{x \in V(T)} (2c)^{\mathsf{width}_{\mathcal{T}}(x)} \cdot \mathsf{height}_{\mathcal{T}}(x),$$

where $\mathsf{height}_{\mathcal{T}}(x)$ denotes the height of x in $\mathcal{T}$, i.e., the distance from x to the deepest leaf in its subtree. Let us not focus on the $\mathsf{height}(x)$ factor at this point, but note that by the above discussion, the factor $(2c)^{\mathsf{width}_{\mathcal{T}}(x)}$ achieves that for every $x \in \mathsf{cut}(\mathcal{C}) \setminus T_{\mathrm{pref}}$,

$$\sum_{C \in \mathcal{C} \mid \mathcal{L}(\mathcal{T})[x] \cap C \neq \emptyset} (2c)^{\mathsf{width}_{\mathcal{T}'}(x^C)} \cdot \mathsf{height}_{\mathcal{T}'}(x^C) < (2c)^{\mathsf{width}_{\mathcal{T}}(x)} \cdot \mathsf{height}_{\mathcal{T}}(x),$$

implying that the potential decreases proportionally to the number of nodes in $\mathsf{cut}(\mathcal{C}) \setminus T_{\mathrm{pref}}$, which justifies implementing the refinement operation in time proportional to $|\mathsf{cut}(\mathcal{C})|$. Before going into more analysis of the potential and the height reduction, let us discuss this implementation.

Given T_{pref}, we wish to find in time $\mathcal{O}_k(|\mathsf{cut}(\mathcal{C})|)$ some representation of a minimal c-small k-closure $\mathcal{C}$ of T_{pref}. We observe that for each oriented edge $\vec{xy} \in \vec{\mathsf{App}}(\mathsf{cut}(\mathcal{C}))$, there is unique $C \in \mathcal{C}$ so that $\mathcal{L}(\mathcal{T})[\vec{xy}] \subseteq C$. Therefore, we define the *appendix edge partition* $\mathsf{aep}(\mathcal{C})$ of $\mathcal{C}$ to be the partition of $\vec{\mathsf{App}}(\mathsf{cut}(\mathcal{C}))$ into $|\mathcal{C}|$ parts naturally corresponding to $\mathcal{C}$. As $|\vec{\mathsf{App}}(\mathsf{cut}(\mathcal{C}))| \leqslant \mathcal{O}(|\mathsf{cut}(\mathcal{C})|)$, we can represent $\mathsf{aep}(\mathcal{C})$ in $\mathcal{O}(|\mathsf{cut}(\mathcal{C})|)$ space. Also, a rank decomposition $\mathcal{T}^\star$ of $\mathcal{C}$ can be represented in $\mathcal{O}(|\mathsf{cut}(\mathcal{C})|)$ space by associating the leaves with the parts of $\mathsf{aep}(\mathcal{C})$. We compute these objects by the following lemma, which we prove in Section 4.9 and overview in Section 4.1.3. This lemma resembles a similar statement we have proved for dynamic treewidth (Lemma 3.3.23).

Lemma 4.1.3 (Informal statement of Lemma 4.4.7)**.** *By maintaining an automaton on $\mathcal{T}$, we can support an operation that given a prefix T_{pref}, in time $\mathcal{O}_k(|\mathsf{cut}(\mathcal{C})|)$ returns $\mathsf{cut}(\mathcal{C})$, $\mathsf{aep}(\mathcal{C})$, and a rank decomposition $\mathcal{T}^\star$ of $\mathcal{C}$ of width at most $2k$, for some minimal c-small k-closure $\mathcal{C}$ of T_{pref}, or concludes that the rankwidth of G is more than k.*

After turning $\mathcal{T}^\star$ into a log-height decomposition $\mathcal{T}^{\star\star}$, we can compute based on $\mathcal{T}^{\star\star}$ and $\mathsf{aep}(\mathcal{C})$ a "recipe" of size $\mathcal{O}(|\mathsf{cut}(\mathcal{C})|)$ on how the subtrees of $\mathcal{T}$ hanging below edges $\vec{xy} \in \vec{\mathsf{App}}(\mathsf{cut}(\mathcal{C}))$ should be re-arranged to transform $\mathcal{T}$ into $\mathcal{T}'$. Even after this, the problem of turning $\mathcal{T}$ into an annotated rank decomposition $\mathcal{T}'$ efficiently turns out to not be straightforward, as we need to compute the annotations for $\mathcal{T}'$. In Section 4.3 we give a divide-and-conquer type algorithm for computing these annotations based on the recipe in $\mathcal{O}_k(|\mathsf{cut}(\mathcal{C})| \log n)$ time (Lemmas 4.3.8 and 4.3.11). This concludes the overview on how $\mathcal{T}$ is transformed into $\mathcal{T}'$ in $\mathcal{O}_k(|\mathsf{cut}(\mathcal{C})| \log n)$ time.

We then return to the potential function $\Phi(\mathcal{T})$. The main idea of the amortized analysis of our algorithm is that each edge update can increase the potential by at most $\mathcal{O}_k(h^2)$ (recall that h is the bound on the height of $\mathcal{T}$) and the refinement operation can increase the potential by at most $\mathcal{O}_k(h|T_{\mathrm{pref}}| \log n)$ and decreases it proportionally to $|\mathsf{cut}(\mathcal{C}) \setminus T_{\mathrm{pref}}|$. The fact that edge updates increase the potential by at most $\mathcal{O}_k(h^2)$ is straightforward from the facts that the update affects the widths of at most $\mathcal{O}(h)$ nodes, and the contribution of each node to potential is at most $\mathcal{O}_k(h)$.

The analysis of the potential change caused by refinement is based on case-analysis of nodes of $\mathcal{T}'$: If a node is in a subtree directly copied from $\mathcal{T}$ to $\mathcal{T}'$, then nothing changes. If a node is of type x^C for $x \in \mathsf{cut}(\mathcal{C}) \setminus T_{\mathrm{pref}}$ and $C \in \mathcal{C}$, then its potential can be charged from the potential of the corresponding node x as argued earlier, and this even decreases the potential proportionally to $|\mathsf{cut}(\mathcal{C}) \setminus T_{\mathrm{pref}}|$. If a node comes from $\mathcal{T}^{\star\star}$, then its height is initially $\mathcal{O}(\log n)$, but can increase when we attach trees $\mathcal{T}^C$ as its descendants. We observe that each $\mathcal{T}^C$ can increase the height of at most $\mathcal{O}(\log n)$ such nodes, so the total potential of such nodes is bounded by $\mathcal{O}_k(|\mathcal{T}^{\star\star}| \log n) + \sum_{C \in \mathcal{C}} \mathcal{O}_k(\mathsf{height}(\mathcal{T}^C) \log n) \leqslant \mathcal{O}_k(h|T_{\mathrm{pref}}| \log n)$ (recall that $|\mathcal{C}| \leqslant \mathcal{O}_k(|T_{\mathrm{pref}}|)$). These arguments imply that if the height of $\mathcal{T}$ stays at most h, then the amortized running time of each update is $\mathcal{O}_k(h^2 \log^2 n)$. It remains to give the height reduction scheme to maintain this height bound.

Suppose the height of $\mathcal{T}$ increased above h by an application of the refinement operation. We wish to argue that whenever the height is more than h, there is a prefix T_{pref}, so that if we apply the refinement on T_{pref}, the potential $\Phi(\mathcal{T}')$ of the resulting decomposition is smaller than $\Phi(\mathcal{T})$, and moreover, the running time of the refinement operation is $\mathcal{O}_k((\Phi(\mathcal{T}) - \Phi(\mathcal{T}')) \cdot \log n)$. For this, we prove the following more fine-grained bound on $\Phi(\mathcal{T}')$:

$$\Phi(\mathcal{T}') \leqslant \Phi(\mathcal{T}) - \sum_{x \in T_{\mathrm{pref}}} \mathsf{height}_{\mathcal{T}}(x) - |\mathsf{cut}(\mathcal{C})| + \log n \cdot \mathcal{O}_k\left(|T_{\mathrm{pref}}| + \sum_{\vec{xy} \in \vec{\mathsf{App}}(T_{\mathrm{pref}})} \mathsf{height}_{\mathcal{T}}(x)\right).$$

This is not very hard to deduce from the construction of $\mathcal{T}'$ and the arguments for bounding the potential change given earlier, but we omit giving a more detailed argument here. Then, it suffices to prove that if $\mathcal{T}$ has height more than h, we can find a nonempty prefix T_{pref} so that according to the above formula,

$\Phi(\mathcal{T}') \leqslant \Phi(\mathcal{T}) - |\mathsf{cut}(\mathcal{C})|$. For this, we use the following result, which we have already proved for the purposes of height reduction in the data structure for dynamic treewidth.

Lemma 3.5.2. *Let $c \geqslant 2$ and T be a binary tree with at most N nodes. If the height of T is at least $2^{\Omega(\sqrt{\log N \log c})}$, then there exists a nonempty prefix T_{pref} of T so that*

$$c \cdot \left(|T_{\mathrm{pref}}| + \sum_{a \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(a) \right) \leqslant \sum_{x \in T_{\mathrm{pref}}} \mathsf{height}_T(x). \tag{3.13}$$

Moreover, if we can access the height of each node of T in constant time, then such T_{pref} can be computed in time $\mathcal{O}(|T_{\mathrm{pref}}|)$.

By plugging in $N = \mathcal{O}(n)$ and $c = f(k)\log n$ for a suitable function $f(k)$, the existence of a desired prefix T_{pref} follows whenever $\mathsf{height}(\mathcal{T}) \geqslant 2^{\Omega(\sqrt{\log n \log(f(k)\log n)})} \geqslant 2^{\Omega_k(\sqrt{\log n \log\log n})}$, which is the claimed bound for h. Then, the height-reduction scheme consists of applying refinement operations on such prefixes T_{pref} until the height is decreased below h. As the running time is proportional to the potential decrease, these operations are "free" from the viewpoint of amortized analysis – exactly the same way as in Chapter 3. This concludes the overview of our dynamic algorithm, up to the Dealternation Lemma and the proof of Lemma 4.1.3, which we will overview in Sections 4.1.2 and 4.1.3, respectively.

Comparison to dynamic treewidth. Our approach for dynamic rankwidth is inspired by the approach for dynamic treewidth. In particular, we design a refinement operation with similar properties to the refinement counterpart for treewidth, so that we can then use the height-reduction scheme encapsulated in Lemma 3.5.2 to control the height of the decomposition. As the combinatorics of treewidth and rankwidth are different, the definitions and structural results used for our refinement operation are different from those of the refinement operation of treewidth. In particular, the concept of closures and Lemmas 4.1.1 and 4.1.2, along with the Dealternation Lemma, are novel structural results about rankwidth. Somewhat surprisingly, in the end our rankwidth version of the refinement operation turned out to be more elegant than the treewidth version, which has a more complicated construction of the resulting decomposition $\mathcal{T}'$, resulting also in a more complicated analysis of the potential. From the more low-level side, manipulating rank decompositions and maintaining automata on them is much more complicated and less researched task than that on tree decompositions. We consider the concept of annotated rank decompositions, along with the efficient algorithms for manipulating them (particularly Lemma 4.3.8), an important contribution of this result, which we will highlight further in Section 4.1.3.

An astute reader might have observed an interesting detail differentiating the statements of Theorem 1.3.1 (the data structure for dynamic treewidth) and Theorem 1.3.6 (the data structure for dynamic rankwidth): In the treewidth case, we are able to monitor whether the treewidth of the dynamic graph is below k at any given point of time, and present a marker "Treewidth too large" instead of the approximate tree decomposition whenever the treewidth exceeds k. This is expressly not the case for dynamic rankwidth – we are forced to assume that the rankwidth of the dynamic graph stays below k after each update. This disparity is caused by the limited power of the technique of *postponing invariant-breaking insertions* of Eppstein et al. [EGIS96]. The details follow below.

In Section 3.8, we designed an efficient data structure (call it a *rejecting data structure*) that maintained an approximate tree decomposition of a dynamic graph G of treewidth at most k, rejecting all updates whose application would increase the treewidth of G above k. A data structure of Theorem 1.3.1 is then implemented as follows: Observe crucially that edge insertions may only increase the treewidth of the dynamic graph and edge removals may only decrease the treewidth. (In other words, the class of graphs of treewidth at most k is *subgraph-closed*.) So we maintain a rejecting data structure for the treewidth of G. Suppose now that we want to add an edge uv to the graph, and we detect that this insertion would increase the treewidth of G above k (because the rejecting data structure prevents us from adding the edge). We *delay* the insertion of the edge to the rejecting data structure by storing uv in a *queue of invariant-breaking insertions*. We also expose the marker "Treewidth too large". Now, the treewidth of the graph with the edge uv added will not drop below k until some edge is removed from G. So, after each removal of an edge from G (which our rejecting data structure can always apply successfully), we process the queue, performing the delayed insertions until either we successfully insert all postponed edges (in which case the treewidth of the graph has dropped below k and the marker "Treewidth too large" can be replaced with the tree decomposition maintained by the rejecting data structure) or some insertion fails (in which case we pause processing the delayed updates and conclude that the treewidth of

G is still above k). At all points of time: (i) the rejecting data structure maintains a tree decomposition of width at most $\mathcal{O}(k)$, (ii) the treewidth of the current graph is at most k if and only if the queue of invariant-breaking insertions is empty, and (iii) the tree decomposition in the rejecting data structure is precisely a tree decomposition of G whenever the queue is empty. Hence Theorem 1.3.1 holds.

It turns out that it is straightforward to produce a counterpart of a rejecting data structure for rankwidth. However, it is not the case anymore that the class of graphs of rankwidth at most k is subgraph-closed – removing an edge from a graph may actually *increase* its rankwidth. Even worse, given a dynamic graph G of rankwidth at most k and a sequence of edge insertions and removals in G whose application ultimately produces another graph of rankwidth at most k, it is not clear whether there exists an order of these insertions and removals that keeps the rankwidth of G below k or even $\mathcal{O}(k)$ at all times; and in the case such a sequence exists, it is not clear how to determine such an order efficiently. Therefore, the technique of [EGIS96] unfortunately does not apply to the setting of rankwidth.

4.1.2 Dealternation Lemma

We now overview a crucial combinatorial result regarding optimum-width rank decompositions that lies at the heart of the dynamic rankwidth data structure: the Dealternation Lemma for rankwidth, proved in Section 4.8. This variant of Dealternation Lemma is a natural variant of the Dealternation Lemma for treewidth proved by Bojańczyk and Pilipczuk [BP22], which we use in the data structure for treewidth in a slightly weaker form (Lemma 3.3.3). Later, we will sketch how the Dealternation Lemma is used in the proof of Lemma 4.1.1.

Our Dealternation Lemma essentially states the following: Whenever $\mathcal{T}^b$ is some rooted rank decomposition of a graph G of unoptimal (but bounded) width, there exists a rank decomposition $\mathcal{T}$ of optimum width in which every subtree $\mathcal{L}(\mathcal{T}^b)[x] \subseteq V(G)$ of $\mathcal{T}^b$ can be decomposed into a bounded number of "simple" pieces of $\mathcal{T}$.

Formally, if $\mathcal{T} = (T, \lambda)$ is a rooted rank decomposition of G and $F \subseteq V(G)$, we say that F is a *tree factor* of $\mathcal{T}$ if $F = \mathcal{L}(\mathcal{T})[v]$ for some $v \in V(T)$, and a *context factor* if F is nonempty and of the form $F_1 \setminus F_2$, where both F_1 and F_2 are tree factors. Then the Dealternation Lemma reads as follows:

Lemma 4.1.4 (Lemma 4.4.1). *There exists a function $f(\ell)$ so that if G is a graph and $\mathcal{T}^b = (T^b, \lambda^b)$ is a rooted rank decomposition of G of width ℓ, then there exists a rooted rank decomposition $\mathcal{T}$ of G of optimum width so that for every node $x \in V(T^b)$, the set $\mathcal{L}(\mathcal{T}^b)[x]$ can be partitioned into a disjoint union of at most $f(\ell)$ factors of $\mathcal{T}$.*

We invite the reader to compare the statement of the Dealternation Lemma above to its treewidth counterpart in Lemma 3.3.3.

Subspace arrangements. In the following sections, we make heavy use of a generalization of the notion of rankwidth to linear spaces over finite fields. Let $\mathbb{F}$ be a finite field; throughout this chapter we assume $\mathbb{F} = \mathrm{GF}(2)$. For two linear subspaces $V_1, V_2 \subseteq \mathbb{F}^d$, let $V_1 + V_2$ denote their sum and $V_1 \cap V_2$ denote their intersection. For $V \subseteq \mathbb{F}^d$, let $\dim(V)$ denote the dimension of V. Any family $\mathcal{V} = \{V_1, \ldots, V_n\}$ of linear subspaces of $\mathbb{F}^d$ is called a *subspace arrangement*. For convenience, let $\langle \mathcal{V} \rangle \coloneqq V_1 + \ldots + V_n$. Let also $\mathbf{e}_1, \ldots, \mathbf{e}_d$ denote the canonical basis of $\mathbb{F}^d$.

A rank decomposition (or more properly, a *branch decomposition*) $\mathcal{T} = (T, \lambda)$ of $\mathcal{V}$ is defined as for graphs or partitions, only that we assign subspaces V_i to the leaves of T. For an edge xy of T, let $\mathcal{L}(\mathcal{T})[\vec{xy}] \subseteq \mathcal{V}$ denote the subfamily of linear subspaces assigned to the leaves of T closer to x than y. If $\mathcal{T}$ is rooted, we define $\mathcal{L}(\mathcal{T})[x]$ analogously to Section 4.1.1. Then the *width* of an edge xy is $\dim(\langle \mathcal{L}(\mathcal{T})[\vec{xy}] \rangle \cap \langle \mathcal{L}(\mathcal{T})[\vec{yx}] \rangle)$, and then the width of $\mathcal{T}$ and the rankwidth of $\mathcal{V}$ are defined naturally. The definitions of tree and context factors also lift naturally to the setting of rooted rank decompositions of subspace arrangements.

As observed in [JKO17], any undirected graph G can be converted to an equivalent subspace arrangement $\mathcal{V}$ as follows: Let $V(G) = \{v_1, \ldots, v_n\}$. Then for $i \in [n]$ define V_i to be the subspace of $\mathbb{F}^n$ spanned by the two vectors $\mathbf{e}_i$ and $\sum_{v_j \in N(v_i)} \mathbf{e}_j$, and set $\mathcal{V} = \{V_1, \ldots, V_n\}$. Then, for any rank decomposition $\mathcal{T}$ of G of width ℓ, the isomorphic rank decomposition $\mathcal{T}'$ of $\mathcal{V}$ has width 2ℓ. So the rankwidth of $\mathcal{V}$ is equal to twice the rankwidth of G. Hence, the Dealternation Lemma can be rephrased in the language of subspace arrangements:

Lemma 4.1.5 (Lemma 4.8.5). *There exists a function $f(\ell)$ so that if G is a graph and $\mathcal{T}^b = (T^b, \lambda^b)$ is a rooted rank decomposition of G of width ℓ, then there exists a rooted rank decomposition $\mathcal{T}$ of G of*

optimum width so that for every node $x \in V(T^b)$, the set $\mathcal{L}(\mathcal{T}^b)[x]$ can be partitioned into a disjoint union of at most $f(\ell)$ factors of $\mathcal{T}$.

For convenience, we will henceforth write $\mathcal{V}_x$ as a shorthand for $\mathcal{L}(\mathcal{T}^b)[x]$. From now on we will only focus on the proof of Lemma 4.1.5.

Outline of the proof of the Dealternation Lemma. The proof of Lemma 4.1.5 is inspired by its treewidth counterpart ([BP22], Lemma 3.3.3): Similarly to how their Dealternation Lemma can be viewed as a purely combinatorial understanding of the treewidth algorithm by Bodlaender and Kloks [BK96], our proof relies heavily on the combinatorial understanding of the rankwidth algorithm by Jeong, Kim and Oum [JKO21]. In fact, as a starting point of the proof, we invoke their result:

Theorem 4.1.6 ([JKO21, Proposition 4.6])**.** *Let $\mathcal{T}^b = (T^b, \lambda^b)$ be a rooted rank decomposition of a subspace arrangement $\mathcal{V}$. Then there exists a rooted rank decomposition $\mathcal{T} = (T, \lambda)$ of the same subspace arrangement $\mathcal{V}$ of optimum width that is "totally pure" with respect to $\mathcal{T}^b$.*

We delay the precise definition of "totally pure" to Section 4.8.1. Intuitively though, $\mathcal{T}$ is totally pure with respect to $\mathcal{T}^b$ if $\mathcal{T}$ excludes, for all $x \in V(T^b)$, specific local "complicated" patterns defined in terms of $\mathcal{V}_x$ and $\mathcal{V} \setminus \mathcal{V}_x$. We now lift Theorem 4.1.6 to show that, in fact, for all $x \in V(T^b)$ the entire decomposition $\mathcal{T}$ admits a simple and bounded-size description in terms of $\mathcal{V}_x$ and $\mathcal{V} \setminus \mathcal{V}_x$.

Let $v \in V(T)$ and $x \in V(T^b)$. We say that v is: (i) *x-full* if $\mathcal{L}(\mathcal{T})[v] \subseteq \mathcal{V}_x$; (ii) *$x$-empty* if $\mathcal{L}(\mathcal{T})[v]$ is disjoint from $\mathcal{V}_x$; and (iii) *x-mixed* otherwise. Now, a nonleaf node v of T is an *x-leaf point* if one child of v is x-empty and the other is x-full; and v is an *x-branch point* if both children of v are x-mixed. We define the *x-mixed skeleton* of $\mathcal{T}$ as a (possibly empty) rooted tree T^{M} with $V(T^{\mathsf{M}})$ comprising the x-leaf points and the x-branch points of T, with two vertices $u, v \in V(T^{\mathsf{M}})$ connected by an edge if the simple path between u and v in T is internally disjoint from $V(T^{\mathsf{M}})$ (see Figure 4.2). We then prove that:

Lemma 4.1.7 (Lemma 4.8.13)**.** *There exists a function $f(\ell)$ so that if $\mathcal{T}^b = (T^b, \lambda^b)$ is a rooted rank decomposition of $\mathcal{V}$ of width ℓ and $\mathcal{T}$ is an optimum-width decomposition of $\mathcal{V}$ that is totally pure with respect to $\mathcal{T}^b$, then, for every $x \in V(T^b)$, the x-mixed skeleton of $\mathcal{T}$ has at most $f(\ell)$ nodes.*

We omit the proof in this overview; however, the proof proceeds by selecting any rank decomposition $\mathcal{T}$ of $\mathcal{V}$ of optimum width that is totally pure with respect to $\mathcal{T}^b$ (its existence is asserted by Theorem 4.1.6) and verifying that it actually satisfies all the conditions of Lemma 4.1.7. However, the work is far from done as $\mathcal{T}$ might not meet the requirements of the Dealternation Lemma.

Fix $x \in V(T^b)$. Observe that every x-full node v yields a tree factor $\mathcal{L}(\mathcal{T})[v] \subseteq \mathcal{V}_x$ (and every tree factor that is a subset of $\mathcal{V}_x$ is like this). On the other hand consider a vertical path $v_1 v_2 \dots v_{p+1}$ in $\mathcal{T}$ without any x-leaf points or x-branch points such that v_{p+1} is x-mixed. For $i \in [p]$, let v_i' be the child of v_i different than v_{i+1}. It can be shown that each node v_i' is either x-full or x-empty. Also whenever, for $1 \leqslant a \leqslant b \leqslant p$, the nodes $v_a', \dots, v_b'$ are x-full, then $\mathcal{L}(\mathcal{T})[v_a] \setminus \mathcal{L}(\mathcal{T})[v_{b+1}] \subseteq \mathcal{V}_x$ is a context factor (and all context factors that are subsets of $\mathcal{V}_x$ are like this). This creates an issue: If, for example, $\mathcal{L}(\mathcal{T})[v_i']$ is x-full for odd $i \in [p]$ and x-empty for even $i \in [p]$ (in other words, the sequence $v_1', \dots, v_p'$ alternates between x-full and x-empty nodes), we will not be able to partition $\mathcal{L}(\mathcal{T}^b)[x]$ into fewer than $\frac{p}{2} - O(1)$ factors. Hence our strategy is to improve the decomposition by *dealternating* all such heavily alternating paths – that is, reorder the nodes along the path so as to bunch the x-full nodes v_i' into a small number of contiguous blocks, bounded by some constant $c_\ell \geqslant 1$ dependent only on ℓ. This reordering is highly nontrivial – utmost care needs to be taken to avoid increasing the width of the decomposition – and its implementation adapts to the setting of rankwidth the toolchain of [BP22], which in turn encapsulates the technique of *typical sequences* of [BK96]. After this is done, we show that we can partition $\mathcal{L}(\mathcal{T}^b)[x]$ into at most $\mathcal{O}(|V(T^{\mathsf{M}})| \cdot c_\ell)$ factors of $\mathcal{T}$, where T^{M} is the x-mixed skeleton of $\mathcal{T}$. Then we repeat the process for each $x \in V(T^b)$. So for $V(T^b) = \{x_1, \dots, x_n\}$, we perform n phases, where in the jth phase, we perform the dealternation as above to produce a partition of $\mathcal{L}(\mathcal{T}^b)[x_j]$ into a small number of factors of $\mathcal{T}$.

This strategy comes with nontrivial requirements: (1) dealternation should not increase the width of the decomposition, (2) for $1 \leqslant i < j \leqslant n$, the reordering performed during the jth phase should preserve all factors in the already constructed partition of $\mathcal{L}(\mathcal{T}^b)[x_i]$, and (3) for $1 \leqslant j < i \leqslant n$, the jth phase should not blow up the size of the x_i-mixed skeleton of $\mathcal{T}$. While it appears hard to ensure all these conditions at once, this feat can fortunately be achieved. Hence, using our approach, we ultimately arrive at the following improvement step:

Lemma 4.1.8 (Local Dealternation Lemma, Lemma 4.8.14)**.** *There exists a function $f(t,\ell)$ such that the following holds. Let $x \in V(T^b)$ and assume that the x-mixed skeleton of $\mathcal{T}$ has t nodes. Then there exists a rooted rank decomposition $\mathcal{T}'$ of $\mathcal{V}$ of optimum width such that:*

- *the set $\mathcal{L}(\mathcal{T}^b)[x]$ is a disjoint union of at most $f(t,\ell)$ factors of $\mathcal{T}'$;*
- *for every $y \in V(T^b)$, the y-mixed skeletons of $\mathcal{T}$ and $\mathcal{T}'$ are equal; and*
- *if y is not an ancestor of x and $F \subseteq \mathcal{L}(\mathcal{T}^b)[y]$ is a factor of $\mathcal{T}$, then F is also a factor of $\mathcal{T}'$.*

Lemma 4.1.8 is proved in Section 4.8.6. Then the Dealternation Lemma follows by sorting the nodes of $V(T^b)$ in the order of nonincreasing distance from the root of T^b and applying Lemma 4.1.8 for each $x \in V(T^b)$ in this order.

Dealternation Lemma to Lemma 4.1.1. We now briefly describe how the Dealternation Lemma implies Lemma 4.1.1; the description is inspired by the proof of the Small Closure Lemma for treewidth (Lemma 3.3.2). Fix $k, \ell \in \mathbb{N}$ and let $c := f(\ell)$, where the function f is as in the statement of the Dealternation Lemma. Let $\mathcal{T}$ be a rank decomposition of G of width ℓ and assume that G has width k. By applying Lemma 4.1.4, let $\mathcal{T}' = (T', \lambda')$ be a rooted rank decomposition of G of width k so that for every node $t \in V(T)$ the set $\mathcal{L}(\mathcal{T})[t]$ can be partitioned into a disjoint union of at most c factors of $\mathcal{T}'$. Then for each $a \in \mathsf{App}_T(T_{\mathrm{pref}})$ let $\mathcal{C}_a$ be the partition of $\mathcal{L}(\mathcal{T})[a]$ into at most c parts that are factors of $\mathcal{T}'$, and let $\mathcal{C} = \bigcup_{a \in \mathsf{App}_T(T_{\mathrm{pref}})} \mathcal{C}_a$. It remains to show that $(G[\mathcal{C}], \mathcal{C})$ has rankwidth at most $2k$. The bound on the rankwidth of $(G[\mathcal{C}], \mathcal{C})$ can be shown in several ways: For instance, one can construct a rank decomposition $\mathcal{T}'' = (T'', \lambda'')$ of $\mathcal{C}$ from $\mathcal{T}'$ by: (1) setting $T'' := T'$, (2) choosing for every $C \in \mathcal{C}$ an arbitrary vertex $v \in C$ and assigning $\lambda''(C) := \lambda'(v)$, and (3) removing from T'' all leaves without any assigned parts of $\mathcal{C}$ and contracting degree-2 vertices. It then can be proved that such a constructed $\mathcal{T}''$ has width at most $2k$. Therefore, $\mathcal{C}$ is a k-closure of T_{pref}, and its c-smallness follows directly from the construction.

4.1.3 Automata for optimum-width decompositions

We then overview an algorithmic result displaying the strength of the model of rank decomposition automata; namely that there exists a rank decomposition automaton computing the exact value of the rankwidth of the underlying graph.

Lemma 4.1.9 (Informal)**.** *Fix integers $k \leqslant \ell$. Suppose $\mathcal{T}$ is an annotated rank decomposition of width at most ℓ of a dynamic graph G. By maintaining an automaton on $\mathcal{T}$, we can support an operation that returns whether the rankwidth of G is at most k.*

We will then use Lemma 4.1.9 to show that given an annotated rank decomposition of small (but possibly unoptimal) width of a graph, we can efficiently construct a rank decomposition of this graph of optimum width:

Lemma 4.1.10 (Lemma 4.9.12)**.** *There is an algorithm that, given as input an annotated rank decomposition $\mathcal{T}$ of width ℓ of a graph G and an integer k, in time $\mathcal{O}_\ell(|\mathcal{T}|)$ either determines that G has rankwidth larger than k, or outputs a (non-annotated) rank decomposition (T, λ) of G of width at most k. Moreover, the resulting decomposition can be annotated in time $\mathcal{O}_\ell(|\mathcal{T}| \log |\mathcal{T}|)$.*

Later, we will show how both Lemmas 4.1.9 and 4.1.10 are used in the proof of Lemma 4.1.3 announced in Section 4.1.1, i.e., that we can maintain an automaton on $\mathcal{T}$ supporting the following operation: given a prefix T_{pref} of $\mathcal{T}$, find a minimal c-small k-closure $\mathcal{C}$ of T_{pref}. Moreover, Lemma 4.1.10 is crucially used in the proof of Theorem 1.3.6; we overview that result in Section 4.1.4.

The results in this section build on (and improve upon) an algorithm of Jeong, Kim, and Oum [JKO21] for computing rankwidth exactly in $\mathcal{O}_k(n^3)$ time by using a dynamic programming procedure that can be regarded as a rankwidth analog of the Bodlaender-Kloks dynamic programming for treewidth [BK96]. Lemma 4.1.10 showcases the strength of annotated rank decompositions, as the corresponding algorithm for rank decompositions given in [JKO21] works in $\mathcal{O}_\ell(|\mathcal{T}|^2)$ time.

Exact rankwidth automaton. The automaton announced in the statement of Lemma 4.1.9 effectively reimplements the subroutine of Branch-Width Compression from the cubic-time rankwidth algorithm of Jeong, Kim and Oum [JKO21]: Given a subspace arrangement $\mathcal{V}$, $|\mathcal{V}| = n$, comprising subspaces of $\mathbb{F}^n$ and a rank decomposition $\mathcal{T}^b$ of $\mathcal{V}$ of width at most ℓ, find a rank decomposition of $\mathcal{V}$ of width at most k

if one exists. However, due to the fact that their algorithm operates on subspaces of n-dimensional linear spaces explicitly, their subroutine works in time $\mathcal{O}_k(n^2)$ – and even this complexity is only achieved after a cubic-time preprocessing of $\mathcal{V}$. In the restricted case of rankwidth of graphs, we are able to break the quadratic time barrier by manipulating the implicit representations of these spaces and optimize the time complexity of our implementation to linear, and even represent the algorithm as an automaton on $\mathcal{T}^b$.

We now give a short overview of Branch-Width Compression in [JKO21]. Recall that $\mathcal{V}_x = \mathcal{L}(\mathcal{T}^b)[x]$ for $x \in V(T^b)$. The *boundary space* at x is $B_x := \langle \mathcal{V}_x \rangle \cap (\mathcal{V} \setminus \mathcal{V}_x)$; so we have $\ell = \max_{x \in V(T^b)} \dim(B_x)$. Let $\mathfrak{B}_x$ be an ordered basis of B_x – any sequence of $\dim(B_x)$ vectors of B_x spanning B_x. Then any vector of B_x can be uniquely represented in the basis $\mathfrak{B}_x$ using $\dim(B_x)$ bits as a linear combination of vectors of $\mathfrak{B}_x$, and any subspace of B_x can be represented using at most $\dim(B_x)^2$ bits as a span of at most $\dim(B_x)$ vectors of B_x. If x is a nonleaf node with two children c_1, c_2, then we define $B'_x = B_x + B_{c_1} + B_{c_2}$, and we let $\mathfrak{B}'_x$ to be an ordered basis of B'_x whose prefix is $\mathfrak{B}_x$. In the algorithm, all subspaces of B_x are represented in the basis $\mathfrak{B}_x$, and all subspaces of B'_x are represented in the basis $\mathfrak{B}'_x$. For any node x of T^b with parent p, let $M_{x\vec{p}}$ be the transition matrix from the basis $\mathfrak{B}_x$ to the basis $\mathfrak{B}'_p$, i.e., the unique $|\mathfrak{B}'_p| \times |\mathfrak{B}_x|$ matrix such that, for every vector $\mathbf{v} \in \mathbb{F}^{\dim(B_x)}$, we have $\sum_{i=1}^{\dim(B_x)} \mathbf{v}_i(\mathfrak{B}_x)_i = \sum_{i=1}^{\dim(B'_p)} (M_{x\vec{p}}\mathbf{v})_i(\mathfrak{B}'_p)_i$. Note that $M_{x\vec{p}}$ can be represented using $\dim(B_x) \cdot \dim(B'_p) = \mathcal{O}_\ell(1)$ bits, even though B_x and B'_p are subspaces of a highly-dimensional space $\mathbb{F}^n$.

In the algorithm the authors compute, for every $x \in V(T^b)$, the *full set at x of width k with respect to $\mathcal{T}^b$*, denoted $\mathrm{FS}_k(x)$, which is a family of objects representing heavily compressed versions of rank decompositions of $\mathcal{V}_x$ that are totally pure with respect to $\mathcal{T}^b$.[14] They show that:

- for a leaf l of T^b, the set $\mathrm{FS}_k(l)$ can be constructed in time $\mathcal{O}_\ell(1)$ given only $\dim(B_l)$;
- for a nonleaf x of T^b with two children c_1, c_2, the set $\mathrm{FS}_k(x)$ can be constructed in time $\mathcal{O}_\ell(1)$ given $\mathrm{FS}_k(c_1)$, $\mathrm{FS}_k(c_2)$, the transition matrices $M_{c_1\vec{x}}$, $M_{c_2\vec{x}}$ and the value $\dim(B_x)$;
- $\mathrm{FS}_k(r) \neq \emptyset$ for the root r of T^b if and only if the rankwidth of $\mathcal{V}$ is at most k; and
- if $\mathrm{FS}_k(r) \neq \emptyset$, then a rank decomposition of $\mathcal{V}$ of width at most k can be reconstructed in time $\mathcal{O}_\ell(n)$ from the values $\mathrm{FS}_k(x)$ for $x \in V(T^b)$ and the transition matrices $M_{x\vec{p}}$.

So, assuming access to the transition matrices $M_{x\vec{p}}$ for all nonroot $x \in V(T^b)$ with parent p, the entire Branch-Width Compression can be implemented in time $\mathcal{O}_\ell(n)$. However, it seems quite hard to determine these matrices efficiently from a general subspace arrangement $\mathcal{V}$: [JKO21] determines the ordered bases $\mathfrak{B}_x$, $\mathfrak{B}'_x$ explicitly and computes the transition matrices $M_{x\vec{p}}$ from these bases afterwards. This approach unfortunately requires $\Omega(n^2)$ time and space since we need $\Omega(n^2)$ bits of memory to simply store all the ordered bases. However, in the setting of rank decompositions of graphs, we can work around this issue using annotated rank decompositions. The following lemma (not proved here) encapsulates the key technical idea of our approach.

Lemma 4.1.11 (informal statement of Lemma 4.9.6)**.** *Suppose $\mathcal{T}$ is a rooted annotated rank decomposition of a graph G and $\mathcal{T}^b$ is the isomorphic rank decomposition of the subspace arrangement $\mathcal{V}$ equivalent to G. Then there exist two families of ordered bases $\{\mathfrak{B}_x\}_{x \in V(T^b)}$, $\{\mathfrak{B}'_x\}_{x \in V(T^b)}$, such that for every $x \in V(T^b)$ with parent p and children c_1, c_2, we can uniquely determine the transition matrices $M_{c_1\vec{x}}$, $M_{c_2\vec{x}}$ and the value $\dim(B_x)$ in time $\mathcal{O}_\ell(1)$ from the annotations of $\mathcal{T}$ around x.*

Recalling the model of rank decomposition automata defined before, observe that we can encode the algorithm of Jeong, Kim and Oum as a rank decomposition automaton running on $\mathcal{T}$:

Lemma 4.1.12 (informal statement of Lemma 4.9.11)**.** *There exists a rank decomposition automaton such that, for any graph G with annotated rank decomposition $\mathcal{T}^b$ of width ℓ, the state of the automaton at node $x \in V(\mathcal{T}^b)$ is exactly $\mathrm{FS}_k(x)$. Each state of the automaton can be evaluated in time $\mathcal{O}_\ell(1)$.*

This essentially resolves Lemma 4.1.9. With the help of the rank decomposition reconstruction subroutine from [JKO21], our algorithm can also output a non-annotated rank decomposition of G of width at most k in linear time, yielding the first part of Lemma 4.1.10. By Lemma 4.3.8, the output decomposition of Lemma 4.1.10 can be annotated in $\mathcal{O}_\ell(|\mathcal{T}| \log |\mathcal{T}|)$ time using a divide-and-conquer type algorithm.

Closure automaton. We then briefly sketch the proof of Lemma 4.1.3 as an application of Lemma 4.1.12: Assuming we maintain appropriate automaton on a decomposition $\mathcal{T}$, we can support an operation that given a prefix T_{pref}, returns an encoding of a minimal c-small k-closure $\mathcal{C}$ of T_{pref}. The proof is ideologically similar to that of Lemma 3.3.23, but with a different set of technical challenges.

[14] This mirrors an analogous definition of a full set in the work of Bodlaender and Kloks [BK96].

The automaton we will construct and maintain is a *closure automaton*. For fixed c and k it computes, for all edges $\vec{xy}$ of $\mathcal{T}$, the family $\mathsf{reps}^{c,k}(\vec{xy})$ of all partitions $\mathcal{C}_{\vec{xy}}$ of $\mathcal{L}(\mathcal{T})[\vec{xy}]$ into at most c parts that can be extended to a minimal c-small k-closure of some prefix T_{pref} with $\vec{xy} \in \vec{\mathsf{App}}(T_{\text{pref}})$. The main challenge is how to represent $\mathcal{C}_{\vec{xy}}$ – storing the partition of $\mathcal{L}(\mathcal{T})[\vec{xy}]$ explicitly is obviously impractical, and even storing $\mathsf{aep}(\mathcal{C}_{\vec{xy}})$ turns out to be too expensive in our algorithm. Instead, for every set $C \in \mathcal{C}_{\vec{xy}}$ we only keep a carefully selected minimal representative of C. Then, with some extensive bookkeeping, we can compute the family $\mathsf{reps}^{c,k}(\vec{xy})$ for all $\vec{xy}$ in time $\mathcal{O}_{c,k}(1)$.

Then, given a prefix T_{pref}, we want to find a closure $\mathcal{C}$ of T_{pref} such that: (i) for every $\vec{xy} \in \vec{\mathsf{App}}(T_{\text{pref}})$, (the representation of) the subfamily of $\mathcal{C}$ restricted to $\mathcal{L}(\mathcal{T})[\vec{xy}]$ belongs to $\mathsf{reps}^{c,k}(\vec{xy})$, (ii) the rankwidth of $\mathcal{C}$ is at most $2k$. This can be achieved in time $\mathcal{O}_{c,k}(|T_{\text{pref}}|)$ by using the exact rankwidth automaton from Lemma 4.1.12 and applying on it standard dynamic programming techniques on automata. With enough care, this dynamic programming allows us to find a *minimal* closure $\mathcal{C}$. Then, restoring the objects $\mathsf{cut}(\mathcal{C})$, $\mathsf{aep}(\mathcal{C})$ and the rank decomposition of $\mathcal{C}$ of width at most $2k$ are straightforward (even if technical) tasks that can be done in total time $\mathcal{O}_{c,k}(|\mathsf{cut}(\mathcal{C})|)$.

4.1.4 Almost-linear time algorithm for rankwidth

Then we show how to compute a rank decomposition of an n-vertex, m-edge graph G of width at most k in time $\mathcal{O}_k(n \cdot 2^{\sqrt{\log n}\log\log n}) + \mathcal{O}(m)$, if such a decomposition exists (Theorem 1.3.6). The full exposition of this algorithm can be found in Section 4.7.

In this section we assume that the input graph G is bipartite, with the bipartition $V(G) = A \cup B$; in Section 4.7.2 we show that the general case can be reduced to the bipartite case by using a construction of Courcelle [Cou06]. Also assume that G has rankwidth at most k. Let $X \triangle Y = (X \cup Y) \setminus (X \cap Y)$ denote the symmetric difference of sets. We say that two vertices $u, v \in V(G)$ are *twins* if $N(u) = N(v)$, and *c-near-twins* for $c \in \mathbb{N}$ if $|N(u) \triangle N(v)| \leqslant c$. The main idea of our algorithm is to exploit the presence of many twins and near-twins in bipartite graphs of small rankwidth.

Consider the following auxiliary problem, which we call Twin Flipping. As input we are given an annotated rank decomposition $\mathcal{T}$ of width at most k of a bipartite graph $G = (A, B, E)$, $E \subseteq A \times B$; a set $X \subseteq A$ with the property that every vertex of X has a twin in $A \setminus X$; and a set of pairs $F \subseteq X \times B$. Let $n = |A| + |B|$ and assume $|F| \leqslant \mathcal{O}_k(n)$. The task is to construct an annotated rank decomposition of $G' := (A, B, E \triangle F)$ of width at most k, assuming it exists. Define a function $T(n)$ with the property that Twin Flipping can be solved in time $\mathcal{O}_k(T(n))$. Then we have:

Lemma 4.1.13. $T(n) \leqslant n \cdot 2^{o(\sqrt{\log n}\log\log n)}$.

Sketch of the proof. Consider G to be a dynamic graph described by an annotated rank decomposition, initially $\mathcal{T}$, maintained by the dynamic rankwidth data structure of Theorem 1.3.4. Then for each $(u, v) \in F$ in the lexicographic order, flip the adjacency between u and v (add the edge uv to G if not present, remove it otherwise). It can be shown that the rankwidth of the dynamic graph never grows above $k + 1$ during this process, so the data structure can perform the initialization and all the updates in time $n \cdot 2^{\mathcal{O}_k(\sqrt{\log n \log\log n})} = \mathcal{O}_k(n \cdot 2^{o(\sqrt{\log n}\log\log n)})$[15], maintaining a 4-approximate decomposition, which can be finally turned into optimal decomposition by Lemma 4.1.10. □

We also define another auxiliary problem, Twin Detection: Construct an efficient data structure that, when initialized with a bipartite graph $G = (A, B, E)$ with $B = \{v_1, \ldots, v_{|B|}\}$, supports the following query: given a set $X \subseteq A$ and a subinterval $[\ell, r]$ of $[1, |B|]$, return the partition of X into the equivalence classes of twins in the induced subgraph $G[X, \{v_\ell, \ldots, v_r\}]$. In Lemma 4.7.10 we propose such a data structure with initialization time $\mathcal{O}(n + m)$ and query time $\mathcal{O}(|X| \log n)$. The implementation uses as a black box a linear-time suffix array construction algorithm of [KSB06].

In the third and final auxiliary problem, Near-Twin Pairing, we get as input an annotated rank decomposition $\mathcal{T}$ of width at most k of a bipartite graph $G = (A, B, E)$ with $|B| \geqslant 2$. On output we should produce: (i) $t = \max(1, \frac{|B|}{\mathcal{O}_k(1)})$ pairwise disjoint pairs of vertices $(u_1, v_1), \ldots, (u_t, v_t)$ of B such that u_i and v_i are $\mathcal{O}_k(\frac{|A|}{|B|})$-near-twins for all $i \in [t]$, and (ii) the sets $N(u_i) \triangle N(v_i)$ for each $i \in [t]$. We show in Lemma 4.7.9 the solution of this problem in time $\mathcal{O}_k(n)$.

[15] The fact that the data structure can be efficiently initialized with an annotated rank decomposition $\mathcal{T}$ is not stated explicitly in Theorem 1.3.4, but this follows readily from the discussion in Section 4.1.1 and we actually prove this in Lemma 4.6.3.

We also use the following straightforward fact: If $\mathcal{T}$ is an annotated rank decomposition of G of width k, and a graph $G^\star$ is created from G by cloning a vertex v (creating a new vertex $v^\star$ such that $N_{G^\star}(v^\star) = N_G(v)$), then $\mathcal{T}$ can be transformed into an annotated rank decomposition $\mathcal{T}^\star$ of $G^\star$ of the same width in time $\mathcal{O}(1)$.

The main ingredient of our algorithm is the following result:

Lemma 4.1.14. *A decomposition of G of width at most k can be found in time $\mathcal{O}_k(T(n)\log^2 n) + \mathcal{O}(m)$.*

Sketch of the proof. In time $\mathcal{O}(n+m)$, initialize the data structure for Twin Detection on the input graph $G = (A, B, E)$. Also suppose $B = \{v_1, \ldots, v_{|B|}\}$. We now design a recursive algorithm that takes as input a subset $A' \subseteq A$ and a subinterval $[\ell, r] \subseteq [1, |B|]$, and returns an annotated rank decomposition of width k of $G[A', B']$, where $B' = \{v_\ell, \ldots, v_r\}$.

The base case is $\ell = r$; then the graph is a forest and we can construct its rank decomposition of width at most k in time $\mathcal{O}(|A'|)$. So suppose that $\ell < r$. We resolve this case in several steps.

Step 1: Filter out the twins. We query the data structure for Twin Detection on $X = A'$ and the interval $[\ell, r]$ in time $\mathcal{O}(|A'| \log n)$; the result of the query can be represented as a subset $A'' \subseteq A'$ with no twins in $G[A'', B']$, and a mapping $\eta : A' \to A''$ such that for every $v \in A' \setminus A''$, $\eta(v)$ is a twin of v in $G[A', B']$. Since A'' has no twins in $G[A'', B']$ and $G[A'', B']$ has rankwidth at most k, we can show that $|A''| \leqslant \mathcal{O}_k(|B'|)$; this statement is proved as Lemma 4.7.8, but has appeared before in various forms and generalizations [PP20, BFLP24]. Hence we will now only compute an annotated rank decomposition $\mathcal{T}$ of $G[A'', B']$ since it is straightforward to add the vertices of $A' \setminus A''$ to $\mathcal{T}$ as soon as $\mathcal{T}$ is constructed.

Step 2: Recurse on B. Let $\delta \approx \frac{1}{2}(\ell + r)$ and let $B_1 = \{v_\ell, \ldots, v_\delta\}$ and $B_2 = \{v_{\delta+1}, \ldots, v_r\}$. For each $i \in [2]$, we construct an annotated rank decomposition $\mathcal{T}_i$ of $G[A'', B_i]$ recursively.

Step 3: Merge the decompositions. The final step – merging $\mathcal{T}_1$ and $\mathcal{T}_2$ into an annotated rank decomposition $\mathcal{T}$ of $G[A'', B_1 \cup B_2]$ – is quite nontrivial. In fact, we will perform this step recursively by implementing a subroutine taking as input a subset $B_2' \subseteq B_2$ and a rank decomposition $\mathcal{T}_2'$ of $G[A'', B_2']$ of width at most k and returning an analogous decomposition $\mathcal{T}'$ of $G[A'', B_1 \cup B_2']$.

First, if $|B_2'| = 1$, then we model the problem at hand as an instance of Twin Flipping as follows: Assume $B_2' = \{v\}$. Choose an arbitrary vertex $u \in B_1$ in $G[A'', B_1]$ and clone it, naming the clone v. Denote the updated graph $G^\star$ and let $\mathcal{T}^\star$ be an annotated rank decomposition of $G^\star$. Then $\mathcal{T}'$ is exactly the result of the Twin Flipping problem for the graph $G^\star$ with sides $B_1 \cup \{v\}$ and A'', the decomposition $\mathcal{T}^\star$, the set $X = \{v\}$ and the set of edges $F = \{wv \mid w \in N(u) \triangle N(v)\}$. We can easily see that the time required to resolve case is $\mathcal{O}_k(T(|A''| + |B_1| + 1)) = \mathcal{O}_k(T(|B'|))$.

Now suppose $|B_2'| \geqslant 2$. Then by Near-Twin Pairing applied to the decomposition $\mathcal{T}_2'$ of $G[A'', B_2']$ we get $t = \max(1, \frac{|B_2'|}{\mathcal{O}_k(1)})$ pairwise disjoint pairs of vertices $(u_i, v_i), \ldots, (u_v, v_t)$ of B such that $|N(u_i) \triangle N(v_i)| \leqslant \mathcal{O}_k(\frac{|A''|}{|B_2'|})$ for each $i \in [t]$. Therefore, $\sum_{i=1}^{t} |N(u_i) \triangle N(v_i)| \leqslant \mathcal{O}_k(|A''|)$. Let $B_2^{\mathrm{del}} = B_2' \setminus \{v_1, \ldots, v_t\}$ and $\mathcal{T}_2^{\mathrm{del}}$ be the rank decomposition of $G[A'', B_2^{\mathrm{del}}]$, easily constructed from $\mathcal{T}_2'$. We run the subroutine recursively for $B_2^{\mathrm{del}} \subseteq B_2$ and $\mathcal{T}_2^{\mathrm{del}}$ and get the decomposition $\mathcal{T}^{\mathrm{del}}$ of $G[A'', B_1 \cup B_2^{\mathrm{del}}]$. Create a new graph $G^\star$ from $G[A'', B_1 \cup B_2^{\mathrm{del}}]$ by cloning, for each $i \in [t]$, the vertex u_i and naming the clone v_i; let also $\mathcal{T}^\star$ be the decomposition of $G^\star$. Finally, let $F = \{wv_i \mid i \in [t], w \in N(u_i) \triangle N(v_i)\}$ and apply Twin Flipping to the graph $G^\star$, its decomposition $\mathcal{T}^\star$, the set $X = \{v_1, \ldots, v_t\}$ and the set of flipped edges F, resulting in the sought decomposition $\mathcal{T}'$. Tracing all the steps described above, excluding the recursive call on the subset B_2^{del}, we find that that these steps can be performed in total time $\mathcal{O}_k(T(|A''| + |B_1| + |B_2'|)) = \mathcal{O}_k(T(|B'|))$.

Since each recursive call takes time $\mathcal{O}_k(T(|B'|))$ and B' decreases in size by a multiplicative factor of $1 - \frac{1}{\mathcal{O}_k(1)}$ on each level of recursion, we get that the recursion terminates after $\mathcal{O}_k(\log |B'|)$ levels and so the entire decomposition-merging subroutine takes total time $\mathcal{O}_k(T(|B'|) \log |B'|)$.

Summary. The recursive reconstruction of an annotated rank decomposition of $G[A', B']$, where $|B'| = r - \ell + 1$, takes time $\mathcal{O}_k(T(|B'|) \log |B'|)$, excluding the time spent in the two recursive calls for subsets of B'. Thus, the total running time of the entire recursive scheme across all levels of recursion is $\mathcal{O}_k(T(n) \log^2 n)$. Including the time required to instantiate the instance of Twin Detection, we get the final time complexity of $\mathcal{O}_k(T(n) \log^2 n) + \mathcal{O}(m)$. □

So Theorem 1.3.6 holds by Lemmas 4.1.13 and 4.1.14. Moreover, an $\mathcal{O}_k(n \log^{\mathcal{O}(1)} n)$ time algorithm for Twin Flipping would immediately imply an improved $\mathcal{O}_k(n \log^{\mathcal{O}(1)} n) + \mathcal{O}(m)$ time algorithm for finding rank decompositions of graphs of width at most k.

4.2 Preliminary results for rank decompositions

In this section we lift the definition of rank decompositions to handle the so-called *partitioned graphs*. Then we prove several additional preliminary results regarding tree decompositions.

Partitioned graphs and rank decompositions. A *partitioned graph* is a pair $(G, \mathcal{C})$, where G is a graph and $\mathcal{C}$ is a partition of $V(G)$. A *rank decomposition* of a partitioned graph $(G, \mathcal{C})$ is a pair $\mathcal{T} = (T, \lambda)$, where T is a cubic tree and λ is a bijection $\lambda\colon \mathcal{C} \to \vec{L}(T)$. A rank decomposition of a graph G is a rank decomposition of $(G, \mathsf{TrivPart}(V(G)))$, where $\mathsf{TrivPart}(V(G))$ denotes the partition of $V(G)$ into sets of size 1. The bijection λ is called the *leaf mapping*. In the case of graphs, we may treat λ as a function $\lambda\colon V(G) \to \vec{L}(T)$. We define that there is no rank decomposition of a partitioned graph with less than 2 parts or a graph with less than 2 vertices. For an oriented edge $\vec{xy} \in \vec{E}(T)$, we denote by $\mathcal{L}(\mathcal{T})[\vec{xy}] = \bigcup_{\vec{lp} \in \vec{L}(T)[\vec{xy}]} \lambda^{-1}(\vec{lp})$ the union of the parts of $\mathcal{C}$ that are mapped to leaf edges that are closer to x than y. A *rooted rank decomposition* of a partitioned graph is defined like a rank decomposition, but the tree T is a binary tree. When $\mathcal{T}$ is a rooted rank decomposition and $t \in V(T)$, we denote by $\mathcal{L}(\mathcal{T})[t] = \bigcup_{\vec{lp} \in \vec{L}(T)[t]} \lambda^{-1}(\vec{lp})$ the union of the parts of $\mathcal{C}$ that are mapped to descendants of t.

Let G be a graph and $A \subseteq V(G)$. We denote $\overline{A} = V(G) \setminus A$. We denote by $\mathsf{cutrk}_G(A)$ the rank of the $|A| \times |\overline{A}|$ 0-1-matrix over the binary field $\mathrm{GF}(2)$ describing adjacencies between vertices in A and vertices in $\overline{A}$ in G. The *width* of an edge $xy \in E(T)$ of a rank decomposition is $\mathsf{cutrk}_G(\mathcal{L}(\mathcal{T})[\vec{xy}]) = \mathsf{cutrk}_G(\mathcal{L}(\mathcal{T})[\vec{yx}])$, and the width of a rank decomposition is the maximum width of its edge. The rankwidth of a graph is the minimum width of a rank decomposition of it.

We will use the following properties of the cutrk_G function.

Lemma 4.2.1 ([OS06])**.** *For any graph G, the function $\mathsf{cutrk}_G\colon 2^{V(G)} \to \mathbb{N}$ is symmetric and submodular, that is,*

1. $\mathsf{cutrk}_G(A) = \mathsf{cutrk}_G(\overline{A})$ *for all* $A \subseteq V(G)$ *and*
2. *for all* $A, B \subseteq V(G)$ *it holds that* $\mathsf{cutrk}_G(A \cup B) + \mathsf{cutrk}_G(A \cap B) \leqslant \mathsf{cutrk}_G(A) + \mathsf{cutrk}_G(B)$.

We will refer to Item 1 as the symmetry of cutrk and to Item 2 as the submodularity of cutrk.

Let us also recall a known lemma that rank decompositions can be transformed into logarithmic height without increasing the width much (see Lemma 2.4.1). This lemma was shown by Courcelle and Kanté [CK07], but we reprove the result here in order to demonstrate the $\mathcal{O}(|V(T)| \log |V(T)|)$ running time and generalize it to the setting of rank decompositions of partitioned graphs.

Lemma 4.2.2. *There is an algorithm that given a (rooted) rank decomposition (T, λ) of a partitioned graph $(G, \mathcal{C})$ of width k, in time $\mathcal{O}(|V(T)| \log |V(T)|)$ returns a rooted rank decomposition of $(G, \mathcal{C})$ of height $\mathcal{O}(\log |V(T)|)$ and width at most $2k$.*

Proof. We assume that the components $C \in \mathcal{C}$ in the representation of λ are represented as pointers so that the representation of λ is of size $\mathcal{O}(|V(T)|)$. Let us also assume without loss of generality that (T, λ) is unrooted.

We will construct a binary tree T^* of height $\mathcal{O}(\log |V(T)|)$ so that

1. every node $t \in V(T^*)$ is labeled with a subtree $\delta(t)$ of T that contains at least one leaf of T,
2. for each $t \in V(T^*)$ there are at most two edges of T that have one endpoint in $V(\delta(t))$ and another endpoint in $V(T) \setminus V(\delta(t))$, and
3. if $\delta(t)$ contains at least two leaves of T, then t has two children c_1 and c_2 so that $L(T) \cap L(\delta(t))$ is the disjoint union of $L(T) \cap L(\delta(c_1))$ and $L(T) \cap L(\delta(c_2))$.

Before giving the algorithm to construct T^*, let us observe that T^* can be transformed into a rooted rank decomposition (T^*, λ^*) of $(G, \mathcal{C})$ of height $\mathcal{O}(\log |V(T)|)$ and width at most $2k$: Note that for each leaf $l \in L(T^*)$, the subtree $\delta(l)$ contains exactly one leaf of T, and these leaves of T are distinct for distinct leaves of T^*. Therefore, there is a natural bijection between $L(T)$ and $L(T^*)$, so we construct λ^* simply by following this bijection. This construction can be implemented in $\mathcal{O}(|V(T)|)$ time. Then, Item 2 implies that for each $t \in V(T^*)$ (except the root), it holds that $\mathcal{L}(T^*)[t] = \mathcal{L}(T)[\vec{xy}] \cap \mathcal{L}(T)[\vec{zw}]$ for some $\vec{xy}, \vec{zw} \in \vec{E}(T)$. Because of submodularity of cutrk_G, this implies that $\mathsf{cutrk}_G(\mathcal{L}(T^*)[t]) \leqslant \mathsf{cutrk}_G(\mathcal{L}(T)[\vec{xy}]) + \mathsf{cutrk}_G(\mathcal{L}(T)[\vec{zw}]) \leqslant 2k$, which implies that (T^*, λ^*) has width at most $2k$.

Then we describe an algorithm to construct such T^* in time $\mathcal{O}(|V(T)|\log|V(T)|)$. The algorithm constructs T^* recursively top-down, in particular, each recursive step takes a subtree $\delta(t)$ of T as an input and if it contains at least two leaves of T, constructs the subtrees $\delta(c_1)$ and $\delta(c_2)$ of T for the two children c_1 and c_2 of t, and recurses to c_1 and c_2. Alternatively, we can also construct subtrees $\delta(c_1), \delta(c_2), \delta(c_3), \delta(c_4)$ of T, where c_1 and c_2 will be the children of t, and c_3 and c_4 the children of c_1, and then recurse to c_2, c_3, and c_4.

Denote $X = L(T) \cap L(\delta(t))$. If there is at most one edge of T that has an endpoint in both $V(\delta(t))$ and $V(T) \setminus V(\delta(t))$, we pick an edge $xy \in E(\delta(t))$ so that $|X \cap L(\delta(t))[\vec{xy}]| \leqslant \frac{2}{3}|X|$ and $|X \cap L(\delta(t))[\vec{yx}]| \leqslant \frac{2}{3}|X|$, and let $\delta(c_1)$ and $\delta(c_2)$ be the two connected components of $\delta(t) - xy$. Such xy can be shown to exist by a simple walking argument on $\delta(t)$.

Then suppose there are two edges of T that have an endpoint in both $V(\delta(t))$ and $V(T) \setminus V(\delta(t))$. If both of them are incident to the same node x of $\delta(t)$, we can set $\delta(t) \coloneqq \delta(t) - \{x\}$ and apply the case of one edge. Therefore suppose one of them is incident to a node x of $\delta(t)$ and other to a node $y \neq x$ of $\delta(t)$. Note that both x and y have degree 2 in $\delta(t)$. Let $x = z_1, z_2, \ldots, z_\ell = y$ be the unique path between x and y in $\delta(t)$. Now, each node z_i on this path is incident to exactly one oriented edge $w_i\vec{z_i} \in \vec{E}(\delta(t))$ so that w_i is not on the path, and moreover, the sets $L(\delta(t))[w_i\vec{z_i}]$ form a partition of X. Let us pick the smallest r so that $\sum_{i=1}^{r} |L(\delta(t))[w_i\vec{z_i}]| \geqslant \frac{|X|}{3}$. First, if $z_r \in \{x, y\}$, we let $\delta(c_1)$ be the connected component of $\delta(t) - \{z_r\}$ that contains all vertices on the path except z_r, and $\delta(c_2)$ the connected component that is disjoint with the path. It can be observed that both of them satisfy Item 2. Moreover, we observe that $\delta(c_1)$ contains at most $\frac{2}{3}|X|$ leaves in X, and there is at most one edge of T that has endpoints in both $V(\delta(c_2))$ and $V(T) - V(\delta(c_2))$, namely the edge $w_r z_r$.

It remains to consider the case $z_r \notin \{x, y\}$. We first let $\delta(c_1)$ and $\delta(c_2)$ be the two connected components of $\delta(t) - z_r z_{r+1}$, with $x \in V(\delta(c_1))$ and $y \in V(\delta(c_2))$. Then, we let $\delta(c_3)$ and $\delta(c_4)$ be the two connected components of $\delta(c_1) - \{z_r\}$, with $x \in V(\delta(c_3))$. We observe that each of the constructed subtrees satisfy Item 2. Moreover, each of $\delta(c_2)$ and $\delta(c_3)$ contain at most $\frac{2}{3}|X|$ leaves in X, and there is at most one edge of T that has endpoints in both $V(\delta(c_4))$ and $V(T) - V(\delta(c_4))$, namely the edge $w_r z_r$.

Clearly, each recursive call of this algorithm can be implemented in $\mathcal{O}(|\delta(t)|)$ time. To obtain both the total time complexity $\mathcal{O}(|V(T)|\log|V(T)|)$ and the $\mathcal{O}(\log|V(T)|)$ height of T^*, it remains to bound the height of this recursion tree. We recall that if there is at most one edge that has endpoints in $V(\delta(t))$ and $V(T) - V(\delta(t))$, then the size of X shrinks by at least a factor $\frac{1}{3}$ when going to the children. Also, in the other two cases, the only case when we recurse to a child where the size of X does not shrink by a factor of $\frac{1}{3}$ is when there is only one edge of T with endpoints in both the subtree of this child and outside of it. We conclude that on any path of length 4 that goes from a node in T^* towards some leaf of T^*, the size of X must shrink by a factor of at least $\frac{1}{3}$, implying that the height of T^* is $\mathcal{O}(\log|V(T)|)$. □

Representatives. A *representative* of a set $A \subseteq V(G)$ in a graph G is a set $R_A \subseteq A$ so that for every $a \in A$ there exists $r \in R_A$ with $N_G(a) \setminus A = N_G(r) \setminus A$. Such set R_A is a *minimal representative* of A if no subset of it is a representative of A. A cut of a graph G is a pair (A, B) so that $V(G)$ is the disjoint union of A and B. We say that vertices $u, v \in A$ are twins over a cut (A, B) if $N(u) \cap B = N(v) \cap B$. A representative graph of a cut (A, B) is a bipartite graph $G[R_A, R_B]$, where R_A is a representative of A and R_B a representative of B. A minimal representative graph of a cut is defined by requiring R_A and R_B to be minimal representatives. We observe that $\mathsf{cutrk}_{G[R_A,R_B]}(R_A) = \mathsf{cutrk}_G(A)$.

We will need the following lemma about how minimal representative graphs are isomorphic to each other.

Lemma 4.2.3. *Let (A, B) be a cut of a graph G, R_A^1, R_A^2 minimal representatives of A, and R_B^1, R_B^2 minimal representatives of B. The graphs $G[R_A^1, R_B^1]$ and $G[R_A^2, R_B^2]$ are isomorphic to each other, and moreover if $R_B^1 = R_B^2$, then there is a unique isomorphism that is identity on $R_B^1 = R_B^2$.*

Proof. For every $v \in R_A^1$ there exists by definition exactly one $u \in R_A^2$ so that $N(v) \cap B = N(u) \cap B$, so we can map such u and v to each other, and similarly for R_B^1 and R_B^2. This is not necessarily the only isomorphism because both sides can be permuted, e.g., when G is a perfect matching between A and B. However, it becomes unique if we fix the mapping for one side. □

We also recall the following well-known lemma, which allows to make use of rank decompositions in dynamic programming.

Lemma 4.2.4. *Let $A \subseteq V(G)$ and R_A a minimal representative of A. Then $|R_A| \leqslant 2^{\mathsf{cutrk}_G(A)}$.*

Proof. Follows from the fact that a matrix of rank k over GF(2) can have at most 2^k distinct rows. □

4.3 Annotated rank decompositions and prefix rebuilding

In this section we introduce our notion of annotated rank decompositions and the notion of prefix-rebuilding updates to manipulate them. Definitions comprise a large part of this section, but we also give (slightly nontrivial) proofs on the implementations of these manipulations.

4.3.1 Annotated rank decompositions

An annotated rank decomposition is a tuple $\mathcal{T} = (T, U, \mathcal{R}, \mathcal{E}, \mathcal{F})$, where

- T is a cubic tree and U is a set,
- $\mathcal{R}$ is a function that maps each oriented edge $\vec{xy} \in \vec{E}(T)$ to a nonempty set $\mathcal{R}(\vec{xy}) \subseteq U$,
- U is the disjoint union of the sets $\mathcal{R}(\vec{lp})$ over the leaf edges $\vec{lp} \in \vec{L}(T)$,
- $\mathcal{E}$ is a function that maps each edge $xy \in E(T)$ to a bipartite graph $\mathcal{E}(xy)$ with bipartition $(\mathcal{R}(\vec{xy}), \mathcal{R}(\vec{yx}))$ and with no twins over this bipartition, and
- $\mathcal{F}$ is a function that maps each path of length three $xyz \in \mathcal{P}_3(T)$ to a *representative map* $\mathcal{F}(xyz) \colon \mathcal{R}(\vec{xy}) \to \mathcal{R}(\vec{yz})$.

For an oriented edge $\vec{xy} \in \vec{E}(T)$, we denote by $\mathcal{L}(\mathcal{T})[\vec{xy}] = \bigcup_{\vec{lp} \in \vec{L}(T)[\vec{xy}]} \mathcal{R}(\vec{lp})$ the union of the elements of U on the leaf edges that are closer to x than y. Let $(G, \mathcal{C})$ be a partitioned graph. We define that an annotated rank decomposition $\mathcal{T} = (T, U, \mathcal{R}, \mathcal{E}, \mathcal{F})$ *encodes* $(G, \mathcal{C})$ if

1. $\mathcal{C} = \{\mathcal{R}(\vec{lp}) \mid \vec{lp} \in \vec{L}(T)\}$, and in particular $V(G) = U$,
2. for all $C \in \mathcal{C}$ the graph $G[C]$ is edgeless,
3. for all $\vec{xy} \in \vec{E}(T)$ the set $\mathcal{R}(\vec{xy})$ is a minimal representative of $\mathcal{L}(\mathcal{T})[\vec{xy}]$ in G and $\mathcal{E}(xy) = G[\mathcal{R}(\vec{xy}), \mathcal{R}(\vec{yx})]$, and
4. for all $xyz \in \mathcal{P}_3(T)$ and $u \in \mathcal{R}(\vec{xy})$ it holds that $N_G(u) \cap \mathcal{R}(\vec{zy}) = N_G(\mathcal{F}(xyz)(u)) \cap \mathcal{R}(\vec{zy})$.

We will call these the properties ENC 1 to 4. Let us then prove that the partitioned graph encoded by $\mathcal{T}$ is uniquely defined by $\mathcal{T}$. The proof contains useful properties of annotated rank decompositions that will be implicitly used later.

Lemma 4.3.1. *If an annotated rank decomposition encodes a partitioned graph, then it uniquely determines the partitioned graph it encodes.*

Proof. Suppose $\mathcal{T} = (T, U, \mathcal{R}, \mathcal{E}, \mathcal{F})$ encodes $(G, \mathcal{C})$. The partition $\mathcal{C} = \{\mathcal{R}(\vec{lp}) \mid \vec{lp} \in \vec{L}(T)\}$ is uniquely defined by $\mathcal{T}$ by ENC 1. Let u, v be distinct vertices in $V(G) = U$. If u and v are in the same part of $\mathcal{C}$ then by ENC 2 there is no edge between u and v.

Then suppose $u \in \mathcal{R}(l_1\vec{p}_1)$ and $v \in \mathcal{R}(l_2\vec{p}_2)$ with $l_1 \neq l_2$. Let $x_1 = l_1, x_2, \ldots, x_{t-1}, x_t = l_2$ be the unique path in T between l_1 and l_2. For $i \in [t-2]$ let $u_i = \mathcal{F}(x_i x_{i+1} x_{i+2}) \circ \ldots \circ \mathcal{F}(x_1 x_2 x_3)(u)$, where $\circ$ denotes the function composition. Let us prove by induction that for every $i \in [t-2]$, it holds that

$$N_G(u) \cap \mathcal{R}(x_{i+2}\vec{x}_{i+1}) = N_G(u_i) \cap \mathcal{R}(x_{i+2}\vec{x}_{i+1}). \tag{4.1}$$

For $i = 1$ it holds by ENC 4. Then, for $i \geqslant 2$ we have by ENC 3 and induction assumption that

$$N_G(u) \cap \mathcal{L}(\mathcal{T})[x_{i+1}\vec{x}_i] = N_G(u_{i-1}) \cap \mathcal{L}(\mathcal{T})[x_{i+1}\vec{x}_i],$$

which implies

$$N_G(u) \cap \mathcal{R}(x_{i+2}\vec{x}_{i+1}) = N_G(u_{i-1}) \cap \mathcal{R}(x_{i+2}\vec{x}_{i+1})$$

because $\mathcal{R}(x_{i+2}\vec{x}_{i+1}) \subseteq \mathcal{L}(\mathcal{T})[x_{i+1}\vec{x}_i]$. This yields Eq. (4.1) by ENC 4 by applying the function $\mathcal{F}(x_i x_{i+1} x_{i+2})$ to $u_{i-1} \in \mathcal{R}(x_i\vec{x}_{i+1})$.

Now, $u_{t-2} \in \mathcal{R}(p_2\vec{l}_2)$ and u is adjacent to v if and only if u_{t-2} is adjacent to v, and therefore by ENC 3 we have that $uv \in E(G)$ if and only if $u_{t-2}v \in E(\mathcal{E}(p_2 l_2))$. □

We say that an annotated rank decomposition encodes a graph G if it encodes the partitioned graph $(G, \mathsf{TrivPart}(V(G)))$.

At this point, let us make a few remarks about the choices of these definitions. We note that it would have been natural to require an additional property that $\mathcal{R}(\vec{xy}) \subseteq \mathcal{R}(\vec{zx}) \cup \mathcal{R}(\vec{wx})$, where $zxy, wxy \in \mathcal{P}_3(T)$. However, this property turns out to be too strong in that in some cases we do not know if it could be maintained efficiently. We also note that an equivalent alternative to storing the functions $\mathcal{F}(xyz)$ would be to store the graphs $G[\mathcal{R}(\vec{xy}), \mathcal{R}(\vec{zy})]$: The function $\mathcal{F}(xyz)$ can be computed given $\mathcal{E}(yz)$ and $G[\mathcal{R}(\vec{xy}), \mathcal{R}(\vec{zy})]$, and conversely the graph $G[\mathcal{R}(\vec{xy}), \mathcal{R}(\vec{zy})]$ can be computed given $\mathcal{F}(xyz)$ and $\mathcal{E}(yz)$. We choose to store $\mathcal{F}(xyz)$ because it is more explicit for the purpose of tracking representatives along the decomposition.

We define $|\mathcal{T}| = |T|$. The width of an annotated rank decomposition is the maximum of $\mathsf{cutrk}_{\mathcal{E}(xy)}(\mathcal{R}(\vec{xy}))$. If the width of an annotated rank decomposition is ℓ, then by Lemma 4.2.4 we have $|\mathcal{R}(\vec{xy})| \leqslant 2^\ell$ for all oriented edges $\vec{xy}$. It follows that an annotated rank decomposition of width ℓ can be represented in space $2^{\mathcal{O}(\ell)}|\mathcal{T}|$.

Let $\mathcal{T}' = (T', \lambda)$ be a rank decomposition of $(G, \mathcal{C})$. We say that an annotated rank decomposition $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ *corresponds* to $\mathcal{T}'$ if $T = T'$ and for all $\vec{lp} \in \vec{L}(T)$ it holds that $\mathcal{R}(\vec{lp}) = \lambda^{-1}(\vec{lp})$. Note that there is a unique rank decomposition of $(G, \mathcal{C})$ that the annotated rank decomposition $\mathcal{T}$ corresponds to. We also observe that if $\mathcal{T}$ corresponds to $\mathcal{T}'$, then the widths of $\mathcal{T}$ and $\mathcal{T}'$ are equal. When talking about annotated rank decompositions we sometimes use definitions that are defined for rank decompositions but not explicitly for annotated rank decompositions, in which case these definitions refer to the rank decomposition that the annotated rank decomposition corresponds to.

We define a *rooted annotated rank decomposition* in the same way as an annotated rank decomposition, except the tree T is a binary tree instead of a cubic tree. If $r \in V(T)$ is the root and $x, y \in V(T)$ are its two children, then we require that $\mathcal{R}(\vec{xr}) = \mathcal{R}(\vec{ry})$, $\mathcal{R}(\vec{yr}) = \mathcal{R}(\vec{rx})$, and the functions $\mathcal{F}(xry)$ and $\mathcal{F}(yrx)$ are identity functions. We observe that an annotated rank decomposition of width ℓ can be turned in $\mathcal{O}_\ell(1)$ time into a corresponding rooted annotated rank decomposition, and vice versa.

We assume that the tree T of an annotated rank decomposition is represented by an adjacency list and the functions $\mathcal{R}$, $\mathcal{E}$, $\mathcal{F}$ as tables. For rooted annotated rank decompositions the adjacency list furthermore contains information on which adjacent node is the parent, and we also always store a pointer to the root node. We also assume that the representation contains a table so that given $u \in U$ we can find $\vec{lp} \in \vec{L}(T)$ so that $u \in \mathcal{R}(\vec{lp})$ in constant time.

4.3.2 Prefix-rebuilding updates

We will maintain a rooted annotated rank decomposition that encodes the dynamic graph G we are maintaining. All updates to the decomposition will be done via *prefix-rebuilding updates*, which informally speaking change a prefix of a rooted annotated rank decomposition, but keep everything else intact. The updates to the graph G will also be made via prefix-rebuilding updates to the decomposition. In particular, we will not maintain G explicitly, but instead G will be represented by the decomposition we are maintaining.

We then define prefix-rebuilding updates formally. An update that changes a rooted annotated rank decomposition $\mathcal{T} = (T, U, \mathcal{R}, \mathcal{E}, \mathcal{F})$ into another rooted annotated rank decomposition $\mathcal{T}' = (T', U, \mathcal{R}', \mathcal{E}', \mathcal{F}')$ is a prefix-rebuilding update if there exists a leafless prefix T_{pref} of T and a leafless prefix T'_{pref} of T' so that

- $T - T_{\text{pref}} = T' - T'_{\text{pref}}$,
- for all $\vec{xy} \in \vec{E}(T - T_{\text{pref}})$ it holds that $\mathcal{R}(\vec{xy}) = \mathcal{R}'(\vec{xy})$,
- for all $\vec{lp} \in \vec{L}(T)$ there exists $p' \in V(T')$ so that $\vec{lp'} \in \vec{L}(T')$ and $\mathcal{R}(\vec{lp}) = \mathcal{R}'(\vec{lp'})$,
- for all $xy \in E(T - T_{\text{pref}})$ it holds that $\mathcal{E}(xy) = \mathcal{E}'(xy)$, and
- for all $xyz \in \mathcal{P}_3(T - T_{\text{pref}})$ it holds that $\mathcal{F}(xyz) = \mathcal{F}'(xyz)$.

We say that such T_{pref} is the prefix of T associated with the update and T'_{pref} the prefix of T' associated with the update. We observe that a prefix-rebuilding update never changes the partition of U associated with the leaves of the decomposition. We also note that because both T and T' are binary trees with the same number of leaves, $|T_{\text{pref}}| = |T'_{\text{pref}}|$ must hold.

The purpose of prefix-rebuilding updates will be to argue that such updates, along with re-computing various auxiliary information stored in the decomposition, can be implemented in time proportional to $|T_{\text{pref}}|$ instead of time proportional to $|V(T)|$. For example, bottom-up dynamic programming on the decomposition would need to be recomputed only for the nodes in T'_{pref}. Next we introduce some definitions to more formally facilitate this.

We define the tuple of *annotations* of $\mathcal{T}'$ with respect to the prefix T'_{pref} to be the triple

$$\mathsf{Annot}(\mathcal{T}', T'_{\text{pref}}) = (\mathcal{R}'|_{\vec{E}(T')\setminus\vec{E}(T'-T'_{\text{pref}})}, \mathcal{E}'|_{E(T')\setminus E(T'-T'_{\text{pref}})}, \mathcal{F}'|_{\mathcal{P}_3(T')\setminus\mathcal{P}_3(T'-T_{\text{pref}})}).$$

Then, we say that the *description* of the prefix-rebuilding update that changes $\mathcal{T}$ into $\mathcal{T}'$ is the triple

$$\overline{u} = (T_{\text{pref}}, T^\star, \mathsf{Annot}(\mathcal{T}', T'_{\text{pref}})),$$

where T_{pref} and $\mathsf{Annot}(\mathcal{T}', T'_{\text{pref}})$ are as defined above, and $T^\star = T'[T'_{\text{pref}} \cup \mathsf{App}_{T'}(T'_{\text{pref}})]$. Note that $T'_{\text{pref}} = V(T^\star) \setminus L(T^\star)$ and $L(T^\star) = \mathsf{App}_T(T_{\text{pref}}) = \mathsf{App}_{T'}(T'_{\text{pref}}) = \mathsf{App}_{T^\star}(T'_{\text{pref}})$. We observe that the resulting rooted annotated rank decomposition $\mathcal{T}'$ is uniquely determined by $\mathcal{T}$ and $\overline{u}$. We denote $|\overline{u}| = |T_{\text{pref}}|$ and observe that if $\mathcal{T}'$ has width ℓ, then $\overline{u}$ can be represented in space $\mathcal{O}_\ell(|\overline{u}|)$.

Next we show that rooted annotated rank decompositions can be maintained efficiently under prefix-rebuilding updates.

Lemma 4.3.2. *Suppose a representation of a rooted annotated rank decomposition $\mathcal{T}$ is already stored. Then, given a description $\overline{u}$ of a prefix-rebuilding update that changes $\mathcal{T}$ into $\mathcal{T}'$ of width ℓ, the representation of $\mathcal{T}$ can be turned into a representation of $\mathcal{T}'$ in time $\mathcal{O}_\ell(|\overline{u}|)$.*

Proof. Let $\mathcal{T} = (T, U, \mathcal{R}, \mathcal{E}, \mathcal{F})$, $\mathcal{T}' = (T', U, \mathcal{R}', \mathcal{E}', \mathcal{F}')$, and $\overline{u} = (T_{\text{pref}}, T^\star, \mathsf{Annot}(\mathcal{T}', T'_{\text{pref}}))$. We first use $T^\star$ to compute for all $a \in \mathsf{App}_T(T_{\text{pref}})$ the parent $\pi(a)$ of a in $T^\star$, which is also the parent of a in T'. Then, we construct T' by taking $T^\star$ and for each $a \in \mathsf{App}_T(T_{\text{pref}})$ attaching the subtree of T rooted at a as a child of $\pi(a)$. This can be done by $\mathcal{O}(1)$ pointer changes for each such a, so we constructed T' in time $\mathcal{O}(|T_{\text{pref}}|)$. In this process also the annotations in the subtrees below such appendices a are preserved, so to construct the rest of the annotations of $\mathcal{T}'$ we just copy the annotations from $\mathsf{Annot}(\mathcal{T}', T'_{\text{pref}})$ in $\mathcal{O}_\ell(|T_{\text{pref}}|)$ time. □

4.3.3 Prefix-rebuilding data structures

To formalize the notion of a rooted annotated rank decomposition that maintains some auxiliary information under prefix-rebuilding updates, we define *prefix-rebuilding data structures*. For $\ell \in \mathbb{N}$, an ℓ-prefix-rebuilding data structure with overhead τ is a data structure that maintains a rooted annotated rank decomposition $\mathcal{T}$ of width at most ℓ that encodes a dynamic graph G, and supports the following queries:

- $\mathsf{Initialize}(\mathcal{T})$: Initialize the data structure with the given rooted annotated rank decomposition $\mathcal{T}$. Assumes that $\mathcal{T}$ encodes a graph G and the width of $\mathcal{T}$ is at most ℓ. Runs in time $\mathcal{O}(\tau \cdot |\mathcal{T}|)$.
- $\mathsf{Update}(\overline{u})$: Given a description $\overline{u}$ of a prefix-rebuilding update that changes $\mathcal{T}$ into $\mathcal{T}'$, apply this update to $\mathcal{T}$. Assumes that $\mathcal{T}'$ encodes a graph G' and the width of $\mathcal{T}'$ is at most ℓ. Runs in time $\mathcal{O}(\tau \cdot |\overline{u}|)$.

Note that an ℓ-prefix-rebuilding data structure with overhead $\tau = \mathcal{O}_\ell(1)$ supporting the two queries mentioned above can be readily implemented by Lemma 4.3.2. The purpose of this definition is to give a template for data structures that implement also other queries in addition to the aforementioned two. As an immediate example, let us give a prefix-rebuilding data structure for maintaining the $\mathsf{height}_T(t)$ function.

Lemma 4.3.3. *Let $\ell \in \mathbb{N}$. There exists an ℓ-prefix-rebuilding data structure with overhead $\mathcal{O}_\ell(1)$ that maintains a rooted annotated rank decomposition $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ that encodes a dynamic graph G, and additionally supports the following query:*

- $\mathsf{Height}(t)$*: Given a node $t \in V(T)$, returns* $\mathsf{height}_T(t)$ *in time* $\mathcal{O}(1)$.

Proof. In the $\mathsf{Initialize}(\mathcal{T})$ query we compute $\mathsf{height}_T(t)$ by bottom-up dynamic programming for every node $t \in V(T)$. This runs in $\mathcal{O}_\ell(|\mathcal{T}|)$ time. The $\mathsf{Update}(\overline{u})$ query is implemented by first using Lemma 4.3.2 to construct $\mathcal{T}' = (T', U, \mathcal{R}', \mathcal{E}', \mathcal{F}')$, and then computing $\mathsf{height}_{T'}(t)$ for all $t \in T'_{\text{pref}}$ by bottom-up dynamic programming, where T'_{pref} is the prefix of T' associated with the update. This runs in $\mathcal{O}_\ell(|\overline{u}|)$ time. Then the $\mathsf{Height}(t)$ query can be implemented by simply returning the already stored height of the node t. □

Let us then clarify our assumptions about prefix-rebuilding data structures. We assume that the stored decomposition $\mathcal{T}$ always encodes a graph. We also assume that the data structure explicitly represents the current decomposition $\mathcal{T}$ at all times, so that we can access it and for example retrieve a copy of $\mathcal{T}$ in $\mathcal{O}_\ell(|\mathcal{T}|)$ time.

As the final lemma of this subsection we give a prefix-rebuilding data structure for making certain straightforward manipulations of descriptions of prefix-rebuilding updates.

Lemma 4.3.4. *Let $\ell \in \mathbb{N}$. There exists an ℓ-prefix-rebuilding data structure with overhead $\mathcal{O}_\ell(1)$ that maintains a rooted annotated rank decomposition $\mathcal{T}$ that encodes a dynamic graph G, and additionally supports the following queries:*

- $\mathsf{Reverse}(\overline{u})$*: Given a description $\overline{u}$ of a prefix-rebuilding update that changes $\mathcal{T}$ into $\mathcal{T}'$, return a description of a prefix-rebuilding update that changes $\mathcal{T}'$ into $\mathcal{T}$. Runs in time $\mathcal{O}_\ell(|\overline{u}|)$.*
- $\mathsf{Compose}(\overline{u}_1, \overline{u}_2)$*: Given two descriptions of prefix-rebuilding updates, $\overline{u}_1$ and $\overline{u}_2$, so that $\overline{u}_1$ changes $\mathcal{T}$ into $\mathcal{T}'$ and $\overline{u}_2$ changes $\mathcal{T}'$ into $\mathcal{T}''$ and both $\mathcal{T}'$ and $\mathcal{T}''$ have width at most ℓ, return a description of a prefix-rebuilding update that changes $\mathcal{T}$ into $\mathcal{T}''$. Runs in time $\mathcal{O}_\ell(|\overline{u}_1| + |\overline{u}_2|)$.*

Proof. We maintain $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ by using Lemma 4.3.2.

The $\mathsf{Reverse}$ query is implemented as follows. Let $\overline{u} = (T_{\mathrm{pref}}, T^\star, \mathsf{Annot}(\mathcal{T}', T'_{\mathrm{pref}}))$, where $T'_{\mathrm{pref}} = V(T^\star) \setminus L(T^\star)$. We observe that now

$$\overline{u}^r = (T'_{\mathrm{pref}}, T[T_{\mathrm{pref}} \cup \mathsf{App}_T(T_{\mathrm{pref}})], \mathsf{Annot}(\mathcal{T}, T_{\mathrm{pref}}))$$

is a description of a prefix-rebuilding update that changes $\mathcal{T}'$ into $\mathcal{T}$, and it can be computed from $\overline{u}$ and $\mathcal{T}$ in $\mathcal{O}_\ell(|\overline{u}|)$ time.

The $\mathsf{Compose}$ query is implemented as follows. Let $\overline{u}_1 = (T^1_{\mathrm{pref}}, T^\star_1, \mathsf{Annot}(\mathcal{T}', T^{1'}_{\mathrm{pref}}))$ and $\overline{u}_2 = (T^2_{\mathrm{pref}}, T^\star_2, \mathsf{Annot}(\mathcal{T}'', T^{2'}_{\mathrm{pref}}))$, where $T^{i'}_{\mathrm{pref}} = V(T^\star_i) \setminus L(T^\star_i)$ for $i \in [2]$. Let $T^\circ_{\mathrm{pref}} = T^1_{\mathrm{pref}} \cup (T^2_{\mathrm{pref}} \setminus T'^1_{\mathrm{pref}})$ and $T^{\circ'}_{\mathrm{pref}} = T^{2'}_{\mathrm{pref}} \cup (T^{1'}_{\mathrm{pref}} \setminus T^2_{\mathrm{pref}})$.

We use the $\mathsf{Reverse}$ query to compute a description $\overline{u}^r_1$ that turns $\mathcal{T}'$ into $\mathcal{T}$, then we use Lemma 4.3.2 to turn $\mathcal{T}$ into $\mathcal{T}'$, then again $\mathsf{Reverse}$ to compute a description $\overline{u}^r_2$ that turns $\mathcal{T}''$ into $\mathcal{T}'$, and then Lemma 4.3.2 to turn $\mathcal{T}'$ into $\mathcal{T}'' = (T'', V(G), \mathcal{R}'', \mathcal{E}'', \mathcal{F}'')$. This runs in time $\mathcal{O}_\ell(|\overline{u}_1| + |\overline{u}_2|)$. Then, we observe that

$$\overline{u}_\circ = (T^\circ_{\mathrm{pref}}, T''[T^{\circ'}_{\mathrm{pref}} \cup \mathsf{App}_T(T^{\circ'}_{\mathrm{pref}})], \mathsf{Annot}(\mathcal{T}'', T^{\circ'}_{\mathrm{pref}}))$$

is a description of a prefix-rebuilding update that changes $\mathcal{T}$ into $\mathcal{T}''$. We can compute $\overline{u}_\circ$ from $\mathcal{T}''$, $\overline{u}_1$, and $\overline{u}_2$ in $\mathcal{O}_\ell(|T^\circ_{\mathrm{pref}}| + |T^{\circ'}_{\mathrm{pref}}|) = \mathcal{O}_\ell(|\overline{u}_1| + |\overline{u}_2|)$ time. We return $\overline{u}_\circ$ and finally turn $\mathcal{T}''$ back into $\mathcal{T}$ with $\overline{u}^r_1$ and $\overline{u}^r_2$. □

4.3.4 Prefix-rearrangement descriptions

In our algorithm we wish to re-arrange rooted annotated rank decompositions by prefix-rebuilding updates without worrying about the details on what happens to the annotations $\mathcal{R}$, $\mathcal{E}$, and $\mathcal{F}$ stored in them. In this subsection we show that prefix-rebuilding updates that are described without the tuple of new annotations and which do not change the graph encoded by the decomposition can be efficiently turned into prefix-rebuilding updates with descriptions as defined in Section 4.3.2. In particular, we introduce *prefix-rearrangement descriptions* as a more high-level versions of descriptions of prefix-rebuilding updates, and show that they can be turned efficiently into descriptions of prefix-rebuilding updates.

Let $\mathcal{T} = (T, U, \mathcal{R}, \mathcal{E}, \mathcal{F})$ be a rooted annotated rank decomposition that encodes a graph G. We define that a prefix-rearrangement description is a pair $\overline{u} = (T_{\mathrm{pref}}, T^\star)$, where T_{pref} is a leafless prefix of T and $T^\star$ is a binary tree with $L(T^\star) = \mathsf{App}_T(T_{\mathrm{pref}})$. A prefix-rebuilding update *corresponds* to $(T_{\mathrm{pref}}, T^\star)$ if it changes $\mathcal{T}$ into a rooted annotated rank decomposition $\mathcal{T}' = (T', U, \mathcal{R}', \mathcal{E}', \mathcal{F}')$ so that

- $\mathcal{T}'$ encodes G,
- T_{pref} is the prefix of T associated with the update, and
- $T^\star = T'[T'_{\mathrm{pref}} \cup \mathsf{App}_{T'}(T'_{\mathrm{pref}})]$, where T'_{pref} is the prefix of T' associated with the update.

In other words, a prefix-rearrangement description is like a prefix-rebuilding description but it does not contain the triple of new annotations, and it is required to maintain the graph G encoded by the decomposition. It can be observed that $\mathcal{T}$ and the prefix-rearrangement description uniquely determine the

resulting tree T', and in particular the rank decomposition to which $\mathcal{T}'$ corresponds, but not necessarily the annotations in $\mathcal{T}'$.

We again denote $|\overline{u}| = |T_{\text{pref}}|$. The rest of this subsection is devoted to showing that given a prefix-rearrangement description $\overline{u}$, a description of a prefix-rebuilding update that corresponds to $\overline{u}$ can be computed in $\mathcal{O}_\ell(|\overline{u}| \log |\overline{u}|)$ time, where ℓ is the maximum of the widths of $\mathcal{T}$ and $\mathcal{T}'$. We start with several auxiliary lemmas. In these lemmas we mostly manipulate unrooted annotated rank decompositions.

We first observe that we can efficiently remove leaves from an annotated rank decomposition.

Lemma 4.3.5. *There is an algorithm that given an annotated rank decomposition $\mathcal{T}$ of width ℓ that encodes a partitioned graph $(G, \mathcal{C})$ and a subset $\mathcal{C}' \subseteq \mathcal{C}$ with $|\mathcal{C}'| \geqslant 2$, in time $\mathcal{O}_\ell(|\mathcal{T}|)$ returns an annotated rank decomposition of width at most ℓ that encodes $(G[\mathcal{C}'], \mathcal{C}')$.*

Proof. Let $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ and $G' = G[\mathcal{C}']$. We will construct an annotated rank decomposition $\mathcal{T}'$ that encodes $(G', \mathcal{C}')$ and has width at most ℓ.

First we construct for all $\vec{xy} \in \vec{E}(T)$ a set $\mathcal{R}'''(\vec{xy}) \subseteq \mathcal{L}(\mathcal{T})[\vec{xy}] \cap V(G')$ that is a minimal representative of $\mathcal{L}(\mathcal{T})[\vec{xy}] \cap V(G')$ in G, along with functions $\phi(\vec{xy})\colon \mathcal{R}'''(\vec{xy}) \to \mathcal{R}(\vec{xy})$ that satisfy $N_G(u) \cap \mathcal{R}(\vec{yx}) = N_G(\phi(\vec{xy})(u)) \cap \mathcal{R}(\vec{yx})$ for all $u \in \mathcal{R}'''(\vec{xy})$. These can be computed by dynamic programming that follows the mapping $\mathcal{F}$ by two depth-first searches on $\mathcal{T}$, first computing for edges pointing towards an arbitrarily chosen root, and second for edges pointing away from the root. Then we construct the graph $\mathcal{E}'''(xy) = G[\mathcal{R}'''(\vec{xy}), \mathcal{R}'''(\vec{yx})]$ for each $xy \in E(T)$ from $\mathcal{E}(xy)$ with the help of the functions $\phi(\vec{xy})$ and $\phi(\vec{yx})$. Then, by using $\mathcal{E}'''(xy)$ we can compute for each $\vec{xy}$ a subset $\mathcal{R}''(\vec{xy}) \subseteq \mathcal{R}'''(\vec{xy})$ that is a minimal representative of $\mathcal{L}(\mathcal{T})[\vec{xy}] \cap V(G')$ in G' (instead of G). We also compute the graphs $\mathcal{E}''(xy) = G[\mathcal{R}''(\vec{xy}), \mathcal{R}''(\vec{yx})]$ for each $xy \in E(T)$.

We also construct for all $xyz \in \mathcal{P}_3(T)$ a function $\mathcal{F}''(xyz)\colon \mathcal{R}''(\vec{xy}) \to \mathcal{R}''(\vec{yz})$ so that for all $u \in \mathcal{R}''(\vec{xy})$ it holds that $N_{G'}(u) \cap \mathcal{R}''(\vec{zy}) = N_{G'}(\mathcal{F}(xyz)(u)) \cap \mathcal{R}''(\vec{zy})$. This can be constructed by first using $\phi(\vec{xy})$ and $\mathcal{F}(xyz)$ to compute $v \in \mathcal{R}(\vec{yz})$ so that $N_G(v) \cap \mathcal{R}(\vec{zy}) = N_G(u) \cap \mathcal{R}(\vec{zy})$ and then using v, $\mathcal{E}(yz)$, and $\phi(\vec{zy})$ to compute $N_{G'}(u) \cap \mathcal{R}''(\vec{zy})$.

We observe that $\mathcal{T}'' = (T, V(G'), \mathcal{R}'', \mathcal{E}'', \mathcal{F}'')$ almost satisfies all the properties required to be an annotated rank decomposition that encodes G': the only issue is that some of the sets $\mathcal{R}''(\vec{xy})$ can be empty. We construct $\mathcal{T}'$ from $\mathcal{T}''$ by deleting all edges xy where either $\mathcal{R}''(\vec{xy})$ or $\mathcal{R}''(\vec{yx})$ is empty, deleting all thus created isolated nodes, and finally contracting all degree-2 nodes. Note that the annotations can be modified in a straightforward way when contracting.

Because $\mathcal{E}''(xy)$ is isomorphic to an induced subgraph of $\mathcal{E}(xy)$ for all $xy \in E(T)$, the width of the resulting decomposition is at most the width of $\mathcal{T}$. Also, because $|\mathcal{R}(\vec{xy})| \leqslant 2^\ell$ for all $\vec{xy} \in \vec{E}(T)$, the algorithm can be implemented in $\mathcal{O}_\ell(|\mathcal{T}|)$ time. □

Then, we will observe that a certain type of induced subgraph finding problem can be solved by dynamic programming on annotated rank decompositions. Let G and H be graphs, and γ a function $\gamma\colon V(G) \to 2^{V(H)}$. We say that H is a *labeled induced subgraph* of (G, γ) if G has an induced subgraph $G[X]$ so that $G[X]$ is isomorphic to H with an isomorphism $\phi\colon X \to V(H)$ so that $\phi(x) \in \gamma(x)$ for all $x \in X$. The pair (X, ϕ) will be called the *witness* of the labeled induced subgraph. The following lemma will be proven in Section 4.10.3 by encoding the problem in CMSO_1 logic.

Lemma 4.3.6. *There is an algorithm that given an annotated rank decomposition $\mathcal{T}$ of width ℓ that encodes a partitioned graph $(G, \mathcal{C})$, a graph H, and a function $\gamma\colon V(G) \to 2^{V(H)}$, in time $\mathcal{O}_{\ell,H}(|\mathcal{T}|)$ either returns a witness of H as a labeled induced subgraph of (G, γ) or returns that (G, γ) does not contain H as a labeled induced subgraph.*

Then we need an algorithm that, given an annotated rank decomposition that encodes a partitioned graph $(G, \mathcal{C})$ and a vertex $v \in V(G)$, outputs $N_G(v)$.

Lemma 4.3.7. *There is an algorithm that, given an annotated rank decomposition $\mathcal{T}$ of width ℓ that encodes a partitioned graph $(G, \mathcal{C})$ and a vertex $v \in V(G)$, in time $\mathcal{O}_\ell(|\mathcal{T}|)$ returns $N_G(v)$.*

Proof. We run a depth-first search that starts at the leaf edge $\vec{lp} \in \vec{L}(T)$ with $v \in \mathcal{R}(\vec{lp})$, and for each successor $\vec{xy}$ of $\vec{lp}$ computes $u \in \mathcal{R}(\vec{xy})$ that represents v by following the mapping $\mathcal{F}$ along the depth-first search. After this, the neighbors of v can be determined from the graphs $\mathcal{E}(l'p')$ of the leaf edges $\vec{l'p'} \in \vec{L}(T)$. As $|\mathcal{R}(\vec{xy})| \leqslant 2^\ell$ for all $\vec{xy} \in \vec{E}(T)$, both steps take $\mathcal{O}_\ell(|\mathcal{T}|)$ time. □

The following lemma will be the main lemma towards the main algorithm of this subsection. It performs the update in the setting when the prefix-rearrangement description completely describes the new tree. After that, we will reduce the general case to this.

Lemma 4.3.8. *There is an algorithm that given an annotated rank decomposition $\mathcal{T}'$ of width at most ℓ that encodes a partitioned graph $(G,\mathcal{C})$ and a rank decomposition (T,λ) of $(G,\mathcal{C})$ of width at most ℓ, in time $\mathcal{O}_\ell(|V(T)|\log|V(T)|)$ returns an annotated rank decomposition $\mathcal{T}$ that encodes $(G,\mathcal{C})$ and corresponds to (T,λ).*

Proof. The idea of the algorithm will be to work recursively by picking an edge $xy \in E(T)$ that corresponds to a balanced cut between the leaves of T, then in time $\mathcal{O}_\ell(|V(T)|)$ computing a minimal representative of the cut $(\mathcal{L}(T,\lambda)[\vec{xy}],\mathcal{L}(T,\lambda)[\vec{yx}])$ of G, then recursively constructing annotated rank decompositions on both sides of this cut, and finally combining them. To make this idea work, we need to keep the "boundary" of the subtree of T that we are currently working on small, and explicitly encode all adjacencies from a representative of the boundary to all other vertices.

More formally, we define a *decomposition-boundary-pair*: Let $\mathcal{T}'$ be an annotated rank decomposition that encodes a partitioned graph $(G',\mathcal{C}')$, B a graph with $V(G') \subseteq V(B)$ so that $B[V(G')]$ is edgeless, and $\mathcal{B}$ a partition of $V(B)\setminus V(G')$ so that for all $C \in \mathcal{B}$ the graph $B[C]$ is edgeless. We call the pair $(B,\mathcal{B})$ a *boundary representation* and the pair $(\mathcal{T}',(B,\mathcal{B}))$ a decomposition-boundary-pair. The pair $(\mathcal{T}',(B,\mathcal{B}))$ encodes a partitioned graph $(G,\mathcal{C})$ where $V(G) = V(B)$, $E(G) = E(B)\cup E(G')$, and $\mathcal{C} = \mathcal{B}\cup\mathcal{C}'$. In particular, the edges in the subgraph induced by $V(G')$ come from $\mathcal{T}'$, and the other edges come from B. Note that we allow $V(G') = V(B)$, in which case $\mathcal{B} = \emptyset$ and B is edgeless.

Then we give our algorithm. We will describe a recursive algorithm that takes as input

- an annotated rank decomposition $\mathcal{T}'$ of width at most ℓ and a boundary representation $(B,\mathcal{B})$ with $|\mathcal{B}| \leqslant 4$ and $|C| \leqslant 2^\ell$ for all $C \in \mathcal{B}$, so that the decomposition-boundary-pair $(\mathcal{T}',(B,\mathcal{B}))$ encodes a partitioned graph $(G,\mathcal{C})$, where $|\mathcal{C}| \geqslant 2$, and
- a rank decomposition (T,λ) of $(G,\mathcal{C})$ of width at most ℓ,

and outputs

- an annotated rank decomposition $\mathcal{T}$ that encodes $(G,\mathcal{C})$ and corresponds to (T,λ).

The base case is that $|\mathcal{C}| \leqslant 3$. In this case $|V(G)| \leqslant 3\cdot 2^\ell$, so we can first explicitly construct $(G,\mathcal{C})$ from $(\mathcal{T}',(B,\mathcal{B}))$, and then from $(G,\mathcal{C})$ and (T,λ) construct an annotated rank decomposition $\mathcal{T}$ that corresponds to (T,λ) in a straightforward way in time $\mathcal{O}_\ell(1)$.

Then we consider the case when $|\mathcal{C}| \geqslant 4$. Let us first pick the edge of T along which we do recursion. We say that a leaf of T is a *boundary leaf* if it corresponds to a part of $\mathcal{C}$ that is in $\mathcal{B}$. By our assumption there are at most 4 boundary leaves. If there are exactly 4 boundary leaves, we pick $xy \in E(T)$ so that both $L(T)[\vec{xy}]$ and $L(T)[\vec{yx}]$ contain 2 boundary leaves (note that this can always be done by a walking argument on the decomposition). Otherwise, we pick $xy \in E(T)$ so that $|L(T)[\vec{xy}]| \leqslant \frac{2}{3}|L(T)|$ and $|L(T)[\vec{yx}]| \leqslant \frac{2}{3}|L(T)|$ (this can also be done by a similar argument). In both of the cases, such xy can be found in time $\mathcal{O}(|V(T)|)$.

Let $(X,Y) = (\mathcal{L}(T,\lambda)[\vec{xy}],\mathcal{L}(T,\lambda)[\vec{yx}])$ be the cut of G corresponding to xy. Next we compute a minimal representative (R_X,R_Y) of (X,Y). Such a representative corresponds to a largest set of vertices $R \subseteq V(G)$ so that in the graph $G[R\cap X, R\cap Y]$ there are no twins over the bipartition $(R\cap X, R\cap Y)$. Because the width of (T,λ) is at most ℓ, we have by Lemma 4.2.4 that $|R_X|,|R_Y| \leqslant 2^\ell$. Therefore, we compute such largest R by a combination of brute-force and Lemma 4.3.6: We guess a graph isomorphic to $G[R\cap X, R\cap Y]$ and how the vertices in $\bigcup\mathcal{B}$ are mapped into this graph. Then we use the graph B and the cut (X,Y) to compute for each vertex in $V(G')$ how it could be mapped to this graph so that it is consistent with the already guessed mapping, and based on that construct an instance of labeled induced subgraph and apply Lemma 4.3.6 with $\mathcal{T}'$ to find such $R \subseteq V(G)$. Note that multiple such R could exist, but we pick arbitrarily a single one found by this procedure. As $|R| \leqslant 2\cdot 2^\ell$ and $|\bigcup\mathcal{B}| \leqslant 4\cdot 2^\ell$, the running time of this step is $\mathcal{O}_\ell(|V(T)|)$.

Then we describe the recursive call. We describe the call only for the X-side of the cut, but it is analogous for the side of Y, with notation using Y in the subscript instead of X. Let $\mathcal{C}_X$ be the partition obtained from $\mathcal{C}$ by first removing all parts that are subsets of Y, and then inserting the part R_Y. Let also $G_X = G[X\cup R_Y]$. Because X and Y are nonempty, we have that $|\mathcal{C}_X| \geqslant 2$. Then, a rank decomposition (T_X,λ_X) of $(G_X,\mathcal{C}_X)$ is obtained from (T,λ) by cutting along xy, taking the side with X in the leaves, and mapping R_Y to the new leaf created by this cutting, and all other parts of $\mathcal{C}_X$ to the same leaves they were previously mapped. The new leaf to which R_Y is mapped will be called y, so T_X is an induced subgraph of T. (Similarly, T_Y is an induced subgraph of T, with $V(T_X)\cap V(T_Y) = \{x,y\}$.) Because R_Y is a representative of Y it follows that the width of (T_X,λ_X) is at most ℓ. Both $\mathcal{C}_X$ and (T_X,λ_X) can be constructed in $\mathcal{O}_\ell(|V(T)|)$ time.

We will recursively call the algorithm with (T_X, λ_X), and for this we must construct a decomposition-boundary-pair that encodes $(G_X, \mathcal{C}_X)$. To deal with technicalities, if $|\mathcal{C}_X| \leqslant 5$, we actually do not apply a recursive call but instead construct $(G_X, \mathcal{C}_X)$ explicitly in time $\mathcal{O}_\ell(|V(T)|)$ by using Lemma 4.3.7, and construct the annotations for (T_X, λ_X) in a straightforward way in time $\mathcal{O}_\ell(1)$. Then, assume $|\mathcal{C}_X| \geqslant 6$. The new boundary representation $(B_X, \mathcal{B}_X)$ is constructed by first removing all vertices in Y from B and from all sets in $\mathcal{B}$, then inserting to $\mathcal{B}$ the set R_Y as a new part, and then inserting to the graph B the vertices R_Y and all edges between R_Y and X, which can be computed in time $\mathcal{O}_\ell(|V(T)|)$ by Lemma 4.3.7 and the fact that $|R_Y| \leqslant 2^\ell$. The fact that $|R_Y| \leqslant 2^\ell$ also implies that the assumption that all parts of $\mathcal{B}_X$ have size at most 2^ℓ holds. We also have to argue that $|\mathcal{B}_X| \leqslant 4$. If $|\mathcal{B}| \leqslant 3$, this holds by the fact that we inserted only one new part. If $|\mathcal{B}| = 4$, then by the selection of xy there are two parts of $\mathcal{B}$ that are subsets of Y, and in fact in this case we have $|\mathcal{B}_X| \leqslant 3$. We will use this fact also later in the analysis of the overall time complexity. The annotated rank decomposition $\mathcal{T}'_X$ of the decomposition-boundary-pair is constructed from $\mathcal{T}'$ in $\mathcal{O}_\ell(|V(T)|)$ time by applying Lemma 4.3.5, in particular, by deleting the parts that are subsets of Y. Here we use $|\mathcal{C}_X| \geqslant 6$ to guarantee that $\mathcal{T}'_X$ has at least two leaves. We then observe that $(\mathcal{T}'_X, (B_X, \mathcal{B}_X))$ is a decomposition-boundary pair that encodes $(G_X, \mathcal{C}_X)$ and satisfies all assumptions required by the recursion.

Then, let $\mathcal{T}_X = (T_X, V(G_X), \mathcal{R}_X, \mathcal{E}_X, \mathcal{F}_X)$ and $\mathcal{T}_Y = (T_Y, V(G_Y), \mathcal{R}_Y, \mathcal{E}_Y, \mathcal{F}_Y)$ be the annotated rank decompositions obtained by the recursive calls. We describe the construction of the annotated rank decomposition $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$. First, for every $\vec{e} \in \vec{E}(T_X) \setminus \{\vec{xy}\}$ we set $\mathcal{R}(\vec{e}) \coloneqq \mathcal{R}_X(\vec{e})$, and for every $\vec{e} \in \vec{E}(T_Y) \setminus \{\vec{yx}\}$ we set $\mathcal{R}(\vec{e}) \coloneqq \mathcal{R}_Y(\vec{e})$. Observe that this sets representatives for all oriented edges of T, and that $\mathcal{R}(\vec{xy}) = R_X$ and $\mathcal{R}(\vec{yx}) = R_Y$. Then, for every $e \in E(T_X) \setminus \{xy\}$ we set $\mathcal{E}(e) \coloneqq \mathcal{E}_X(e)$ and for every $e \in E(T_Y) \setminus \{xy\}$ we set $\mathcal{E}(e) \coloneqq \mathcal{E}_Y(e)$. We set $\mathcal{E}(xy) \coloneqq G[R_X, R_Y]$, which can be computed in time $\mathcal{O}_\ell(|V(T)|)$ by Lemma 4.3.7.

At this point, we note that from the fact that (R_X, R_Y) is a minimal representative of (X, Y), and by induction on the recursion, it follows that for all $\vec{ab} \in \vec{E}(T)$, the set $\mathcal{R}(\vec{ab})$ is a minimal representative of $\mathcal{L}(\mathcal{T})[\vec{ab}]$, and that for all $ab \in E(T)$, $\mathcal{E}(ab) = G[\mathcal{R}(\vec{ab}), \mathcal{R}(\vec{ba})]$. In particular, $\mathcal{T}$ satisfies the property ENC 3. Also the properties ENC 1 and 2 are clearly satisfied.

Then we construct $\mathcal{F}$. First, for every $abc \in \mathcal{P}_3(T_X)$ so that $c \neq y$ we have $\mathcal{R}_X(\vec{ab}) = \mathcal{R}(\vec{ab})$ and $\mathcal{R}_X(\vec{bc}) = \mathcal{R}(\vec{bc})$, so we can set $\mathcal{F}(abc) \coloneqq \mathcal{F}_X(abc)$. Analogously, for every $abc \in \mathcal{P}_3(T_Y)$ so that $c \neq x$ we set $\mathcal{F}(abc) \coloneqq \mathcal{F}_Y(abc)$. Then consider $txy \in \mathcal{P}_3(T_X)$ for arbitrary such $t \in V(T_X)$. We have that $\mathcal{E}_X(xy) = G[\mathcal{R}_X(\vec{xy}), R_Y]$ and $\mathcal{R}_X(\vec{xy})$ is a minimal representative of $\mathcal{L}(\mathcal{T})[\vec{xy}]$. By Lemma 4.2.3, $\mathcal{E}_X(xy)$ is isomorphic to $G[R_X, R_Y]$ with an isomorphism that is identity on R_Y, and such an isomorphism is unique. We find such isomorphism $\phi\colon \mathcal{R}_X(\vec{xy}) \cup R_Y \to R_X \cup R_Y$ in $\mathcal{O}_\ell(1)$ time. Then we construct $\mathcal{F}(txy)$ by letting $\mathcal{F}(txy)(r) = \phi(\mathcal{F}_X(txy)(r))$ for all $r \in \mathcal{R}(\vec{tx})$. For $tyx \in \mathcal{P}_3(T_Y)$ we construct $\mathcal{F}(tyx)$ analogously.

It can be observed that this construction can be implemented in $\mathcal{O}_\ell(|V(T)|)$ time. It remains to show that the constructed annotated rank decomposition $\mathcal{T}$ indeed encodes $(G, \mathcal{C})$ and corresponds to (T, λ). We observe that by construction $\mathcal{T}$ corresponds to (T, λ). For showing that $\mathcal{T}$ encodes $(G, \mathcal{C})$, it remains to show ENC 4.

Claim 4.3.9. *For all $abc \in \mathcal{P}_3(T)$ and $u \in \mathcal{R}(\vec{ab})$, we have $N_G(u) \cap \mathcal{R}(\vec{cb}) = N_G(\mathcal{F}(abc)(u)) \cap \mathcal{R}(\vec{cb})$.*

Proof of the claim. For all $abc \in \mathcal{P}_3(T)$ except of form $abc \in \{txy, tyx\}$ this holds because it holds for $\mathcal{T}_X$ and $\mathcal{T}_Y$ and the graphs G_X and G_Y are induced subgraphs of G.

Then consider the case $abc = txy$. Recall that $R_Y = \mathcal{R}(\vec{yx}) = \mathcal{R}_X(\vec{yx})$ and let ϕ be the unique isomorphism from $\mathcal{E}_X(xy) = G[\mathcal{R}_X(\vec{xy}), R_Y]$ to $G[R_X, R_Y]$ that is identity on R_Y. We have that

$$\begin{aligned} N_G(u) \cap R_Y &= N_G(\mathcal{F}_X(txy)(u)) \cap R_Y && \text{(by ENC 4 on } \mathcal{T}_X) \\ &= N_G(\phi(\mathcal{F}_X(txy)(u))) \cap R_Y && \text{(by isomorphism)} \\ &= N_G(\mathcal{F}(txy)(u)) \cap R_Y && \text{(by construction)} \end{aligned}$$

The case of $abc = tyx$ is similar. ◁

This concludes the proof that the output of the algorithm is as claimed.

Then we analyze the time complexity of the algorithm. We already analyzed that a single recursive call takes $\mathcal{O}_\ell(|V(T)|)$ time. It remains to observe that if $|\mathcal{B}| = 4$, then in the child calls it holds that $|\mathcal{B}_X|, |\mathcal{B}_Y| \leqslant 3$, and that if $|\mathcal{B}| \leqslant 3$, then in the child calls it holds that $|L(T_X)|, |L(T_Y)| \leqslant \frac{2}{3}|L(T)| + 1$. Because T is a cubic tree we have $|V(T)| = \mathcal{O}(|L(T)|)$, and therefore a standard analysis of divide-and-conquer algorithms gives the total time complexity $\mathcal{O}_\ell(|V(T)| \log |V(T)|)$. □

Then we will present one more auxiliary lemma that will be used in reducing the general case to the case of Lemma 4.3.8.

Let $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ be a rooted annotated rank decomposition that encodes a partitioned graph $(G, \mathcal{C})$. Given a leafless connected node set $T_{\text{conn}} \subseteq V(T) \setminus L(T)$, we denote by $\mathsf{RepPart}(\mathcal{T}, T_{\text{conn}})$ the partition $\{\mathcal{R}(\vec{ap}) \mid \vec{ap} \in \vec{\mathsf{App}}_T(T_{\text{conn}})\}$ naturally associated with the appendix edges of T_{conn}. Obtaining an annotated rank decomposition of the partitioned graph $(G[\mathsf{RepPart}(\mathcal{T}, T_{\text{conn}})], \mathsf{RepPart}(\mathcal{T}, T_{\text{conn}}))$ from $\mathcal{T}$ is almost straightforward, but we have to deal with a technical issue arising from the fact that some representatives in $\mathcal{T}$ inside the subtree $T[T_{\text{conn}}]$ are not necessarily in $\bigcup \mathsf{RepPart}(\mathcal{T}, T_{\text{conn}})$.

Lemma 4.3.10. *Let $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ be a rooted annotated rank decomposition of width ℓ that encodes $(G, \mathcal{C})$ and whose representation is already stored. There is an algorithm that given a leafless connected node set $T_{\text{conn}} \subseteq V(T) \setminus L(T)$, in time $\mathcal{O}_\ell(|T_{\text{conn}}|)$ returns an annotated rank decomposition of width at most ℓ that encodes the partitioned graph $(G[\mathsf{RepPart}(\mathcal{T}, T_{\text{conn}})], \mathsf{RepPart}(\mathcal{T}, T_{\text{conn}}))$.*

Proof. We denote $(G', \mathcal{C}) = (G[\mathsf{RepPart}(\mathcal{T}, T_{\text{conn}})], \mathsf{RepPart}(\mathcal{T}, T_{\text{conn}}))$.

Let $T' = T[T_{\text{conn}} \cup \mathsf{App}_T(T_{\text{conn}})]$ be the subtree of T induced by T_{conn} and its appendices. We construct $\mathcal{T}' = (T', V(G'), \mathcal{R}|_{\vec{E}(T')}, \mathcal{E}|_{E(T')}, \mathcal{F}|_{\mathcal{P}_3(T')})$ in a straightforward way in $\mathcal{O}_\ell(|T_{\text{conn}}|)$ time. It can be observed that $\mathcal{T}'$ is almost an annotated rank decomposition that encodes $(G', \mathcal{C})$: the only issue is that some representatives are not from the set $V(G')$. This issue can be fixed by finding for every representative $u \in \mathcal{R}(\vec{xy})$ with $u \notin V(G')$ a representative $u' \in V(G')$ with $N(u') \cap \mathcal{L}(\mathcal{T})[\vec{yx}] = N(u) \cap \mathcal{L}(\mathcal{T})[\vec{yx}]$, and replacing u with u' in $\mathcal{R}(\vec{xy})$, $\mathcal{E}(xy)$, and in the representative maps that concern the edge $\vec{xy}$. This can be done in $\mathcal{O}_\ell(|T_{\text{conn}}|)$ time by a 2-phase dynamic programming that first finds such representatives u' on oriented edges pointing towards the root, and then on oriented edges pointing towards the leaves. Finally, it is straightforward to turn the obtained rooted annotated rank decomposition into unrooted. □

Then we give the main algorithm of this subsection.

Lemma 4.3.11. *There exists an ℓ-prefix-rebuilding data structure that maintains a rooted annotated rank decomposition $\mathcal{T}$ and additionally supports the following query:*

- $\mathsf{Translate}(T_{\text{pref}}, T^\star)$*: Given a prefix-rearrangement description $(T_{\text{pref}}, T^\star)$ on the decomposition $\mathcal{T}$, in time $\mathcal{O}_{\ell,\ell'}(|T_{\text{pref}}| \log |T_{\text{pref}}|)$ returns a description of a corresponding prefix-rebuilding update, where ℓ' is the width of the resulting rooted annotated rank decomposition.*

Proof. We maintain the rooted annotated rank decomposition $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ that encodes a graph G by making use of Lemma 4.3.2. It remains to describe how the $\mathsf{Translate}(T_{\text{pref}}, T^\star)$ query is implemented. Throughout the proof we will use $\pi \colon \vec{\mathsf{App}}_T(T_{\text{pref}}) \to \vec{L}(T^\star)$ to denote the bijection that maps an appendix edge $\vec{ap} \in \vec{\mathsf{App}}_T(T_{\text{pref}})$ to the corresponding edge $\vec{ap'} \in \vec{L}(T^\star)$.

Consider the partitioned graph $(G^\star, \mathcal{C}^\star) = (G[\mathsf{RepPart}(\mathcal{T}, T_{\text{pref}})], \mathsf{RepPart}(\mathcal{T}, T_{\text{pref}}))$. By applying Lemma 4.3.10, we obtain an annotated rank decomposition $\mathcal{T}^{\star\star}$ of width at most ℓ that encodes $(G^\star, \mathcal{C}^\star)$. Then let $\lambda^\star \colon \mathcal{C}^\star \to \vec{L}(T^\star)$ be the function that maps $\mathcal{R}(\vec{ap})$ to $\pi(\vec{ap})$ for all appendix edges $\vec{ap} \in \vec{\mathsf{App}}_T(T_{\text{pref}})$. We observe that $(T^\star, \lambda^\star)$ is a rooted rank decomposition of $(G^\star, \mathcal{C}^\star)$ of width at most ℓ'. Then we apply Lemma 4.3.8 with $\mathcal{T}^{\star\star}$ and $(T^\star, \lambda^\star)$ to obtain a rooted annotated rank decomposition $\mathcal{T}^\star = (T^\star, V(G^\star), \mathcal{R}^\star, \mathcal{E}^\star, \mathcal{F}^\star)$ that encodes $(G^\star, \mathcal{C}^\star)$ and corresponds to $(T^\star, \lambda^\star)$. Note that even though Lemma 4.3.8 works with unrooted decompositions, it is simple to make it work for rooted decompositions by unrooting $(T^\star, \lambda^\star)$ before applying it and then rooting the returned decomposition at the corresponding place. So far all the steps have taken $\mathcal{O}_{\ell,\ell'}(|T_{\text{pref}}| \log |T_{\text{pref}}|)$ time. It remains to attach the subtrees below the appendices of T_{pref} to $\mathcal{T}^\star$.

We consider an annotated rank decomposition $\mathcal{T}' = (T', V(G), \mathcal{R}', \mathcal{E}', \mathcal{F}')$ that is constructed as follows. We start with $\mathcal{T}^\star$, and then for every appendix edge $\vec{ap} \in \vec{\mathsf{App}}_T(T_{\text{pref}})$, we attach the subtree of T below $\vec{ap}$ to T' so that $\vec{ap}$ is identified with $\pi(\vec{ap}) = \vec{ap'}$, and also copy all annotations associated to that subtree in $\mathcal{T}$ to $\mathcal{T}'$. We do not copy the annotations on the edge ap, in particular, it will hold that $\mathcal{R}'(\vec{pa}) = \mathcal{R}^\star(\vec{pa})$ and $\mathcal{R}'(\vec{ap'}) = \mathcal{R}^\star(\vec{ap'}) = \mathcal{R}(\vec{ap})$.

The functions $\mathcal{F}'(tap')$ where $\vec{ap'} = \pi(\vec{ap})$ for $\vec{ap} \in \vec{\mathsf{App}}_T(T_{\text{pref}})$ and t is a child of a can be copied from $\mathcal{T}$ in the natural way, so it remains to construct the functions $\mathcal{F}'(p'at)$. It holds that $\mathcal{R}(\vec{ap}) = \mathcal{R}'(\vec{ap'})$ and both $\mathcal{E}(ap)$ and $\mathcal{E}'(ap')$ are representative graphs of $(\mathcal{L}(\mathcal{T})[\vec{ap}], \mathcal{L}(\mathcal{T})[\vec{pa}])$. Therefore, by Lemma 4.2.3 let $\phi \colon \mathcal{R}'(\vec{ap'}) \cup \mathcal{R}'(\vec{pa}) \to \mathcal{R}(\vec{ap}) \cup \mathcal{R}(\vec{pa})$ be the unique isomorphism between $\mathcal{E}'(ap')$ and $\mathcal{E}(ap)$ that is identity on $\mathcal{R}'(\vec{ap'}) = \mathcal{R}(\vec{ap})$. Such ϕ can be computed in $\mathcal{O}_\ell(1)$ time. We construct $\mathcal{F}'(p'at)$ by setting $\mathcal{F}'(p'at)(u) := \mathcal{F}(pat)(\phi(u))$ for all $u \in \mathcal{R}'(\vec{pa})$.

Clearly, this construction of $\mathcal{T}'$ can be implemented by a prefix-rebuilding update that corresponds to the given prefix-rearrangement description, and the description of this prefix-rebuilding update can be computed according to the previous discussion in $\mathcal{O}_{\ell,\ell'}(|T_{\text{pref}}| \log |T_{\text{pref}}|)$ time. It remains to show that $\mathcal{T}'$ encodes G.

The properties ENC 1 and 2 obviously hold. Then we show ENC 3.

Claim 4.3.12. *For all $\vec{xy} \in \vec{E}(T')$ it holds that $\mathcal{R}'(\vec{xy})$ is a minimal representative of $\mathcal{L}(\mathcal{T}')[\vec{xy}]$ and $\mathcal{E}'(xy) = G[\mathcal{R}'(\vec{xy}), \mathcal{R}'(\vec{yx})]$.*

Proof of the claim. First, suppose that $\vec{xy} \in \vec{E}(T') \setminus \vec{E}(T^\star)$. We have $\mathcal{L}(\mathcal{T}')[\vec{xy}] = \mathcal{L}(\mathcal{T})[\vec{xy}]$, $\mathcal{R}'(\vec{xy}) = \mathcal{R}(\vec{xy})$, and $\mathcal{E}'(xy) = \mathcal{E}(xy)$, so the claim holds because $\mathcal{T}$ encodes G.

Then suppose $\vec{xy} \in \vec{E}(T^\star)$. Because $\mathcal{T}^\star$ encodes $(G^\star, \mathcal{C}^\star)$, $\mathcal{R}'(\vec{xy})$ is a minimal representative of $\mathcal{L}(\mathcal{T}^\star)[\vec{xy}]$ in $G^\star$ and $\mathcal{E}'(xy) = G^\star[\mathcal{R}'(\vec{xy}), \mathcal{R}'(\vec{yx})]$. To obtain that $\mathcal{R}'(\vec{xy})$ is a minimal representative of $\mathcal{L}(\mathcal{T}')[\vec{xy}]$ and $\mathcal{E}'(xy) = G[\mathcal{R}'(\vec{xy}), \mathcal{R}'(\vec{yx})]$, it suffices to argue that $\mathcal{L}(\mathcal{T}^\star)[\vec{xy}]$ is a representative of $\mathcal{L}(\mathcal{T}')[\vec{xy}]$ in G and $G[\mathcal{L}(\mathcal{T}^\star)[\vec{xy}], \mathcal{L}(\mathcal{T}^\star)[\vec{yx}]] = G^\star[\mathcal{L}(\mathcal{T}^\star)[\vec{xy}], \mathcal{L}(\mathcal{T}^\star)[\vec{yx}]]$. The former follows from the fact that for each $\vec{ap} \in \vec{\mathsf{App}}_T(T_{\text{pref}})$ the set $\mathcal{R}(\vec{ap}) = \mathcal{R}'(\pi(\vec{ap}))$ is a representative of $\mathcal{L}(\mathcal{T})[\vec{ap}] = \mathcal{L}(\mathcal{T}')[\pi(\vec{ap})]$. The latter follows from the definition of $G^\star$ and the fact that for each $\vec{ap} \in \vec{\mathsf{App}}_T(T_{\text{pref}})$ either $\mathcal{R}(\vec{ap}) \subseteq \mathcal{L}(\mathcal{T}^\star)[\vec{xy}]$ or $\mathcal{R}(\vec{ap}) \subseteq \mathcal{L}(\mathcal{T}^\star)[\vec{yx}]$. ◁

The next claim will imply ENC 4.

Claim 4.3.13. *For all $xyz \in \mathcal{P}_3(T')$ and $u \in \mathcal{R}'(\vec{xy})$, it holds that*

$$N_G(u) \cap \mathcal{R}'(\vec{zy}) = N_G(\mathcal{F}'(xyz)(u)) \cap \mathcal{R}'(\vec{zy}).$$

Proof of the claim. When $y \notin V(T^\star)$ or when $xyz = tap'$ for $\vec{ap'} = \pi(\vec{ap})$ with $\vec{ap} \in \vec{\mathsf{App}}_T(T_{\text{pref}})$ this holds by the property ENC 4 of $\mathcal{T}$. Also when $xyz \in \mathcal{P}_3(T^\star)$ this holds by the property ENC 4 of $\mathcal{T}^\star$. It remains to consider the case of $xyz = p'at$ for $\vec{ap'} = \pi(\vec{ap})$ with $\vec{ap} \in \vec{\mathsf{App}}_T(T_{\text{pref}})$.

Let ϕ be the unique isomorphism between $\mathcal{E}'(ap')$ and $\mathcal{E}(ap)$ that is identity on $\mathcal{R}'(\vec{ap'}) = \mathcal{R}(\vec{ap})$, and recall that $\mathcal{R}'(\vec{ta}) = \mathcal{R}(\vec{ta})$. We have that $N_G(u) \cap \mathcal{R}(\vec{ap}) = N_G(\phi(u)) \cap \mathcal{R}(\vec{ap})$. Because $\mathcal{R}(\vec{ap})$ is a representative of $\mathcal{L}(\mathcal{T})[\vec{ap}]$, this implies that $N_G(u) \cap \mathcal{L}(\mathcal{T})[\vec{ap}] = N_G(\phi(u)) \cap \mathcal{L}(\mathcal{T})[\vec{ap}]$. Now, $\mathcal{R}'(\vec{ta}) \subseteq \mathcal{L}(\mathcal{T})[\vec{ta}] \subseteq \mathcal{L}(\mathcal{T})[\vec{ap}]$, so we get that

$$\begin{aligned}
N_G(u) \cap \mathcal{R}'(\vec{ta}) &= N_G(\phi(u)) \cap \mathcal{R}'(\vec{ta}) \\
&= N_G(\mathcal{F}(pat)(\phi(u))) \cap \mathcal{R}'(\vec{ta}) && \text{(by ENC 4 on } \mathcal{T}) \\
&= N_G(\mathcal{F}'(p'at)(u)) \cap \mathcal{R}'(\vec{ta}). && \text{(by construction of } \mathcal{F}')
\end{aligned}$$

◁

Hence the construction of $\mathcal{T}'$ is correct. □

4.3.5 Edge update descriptions

The dynamic graph G in our algorithm is represented by an annotated rank decomposition that encodes G, and therefore we use prefix-rebuilding updates to update G. In this section we give a higher-level formalism for describing edge updates, and show that it can be translated to corresponding descriptions of prefix-rebuilding updates efficiently.

Let $\mathcal{T} = (T, U, \mathcal{R}, \mathcal{E}, \mathcal{F})$ be a rooted annotated rank decomposition that encodes a graph G. An *edge update description* is a quadruple $\overline{u} = (W, T_{\text{pref}}, \mathcal{R}^\star, \mathcal{E}^\star)$, where

- $W \subseteq V(G)$,
- T_{pref} is a prefix of T so that if $\vec{lp} \in \vec{L}(T)$ and $\mathcal{R}(\vec{lp}) \subseteq W$ then $l \in T_{\text{pref}}$,
- $\mathcal{R}^\star$ is a function that maps each $\vec{xy} \in \vec{E}(T[T_{\text{pref}}])$ to a nonempty set $\mathcal{R}^\star(\vec{xy}) \subseteq \mathcal{L}(\mathcal{T})[\vec{xy}]$,
- $\mathcal{E}^\star$ is a function that maps each $xy \in E(T[T_{\text{pref}}])$ to a bipartite graph $\mathcal{E}^\star(xy)$ with bipartition $(\mathcal{R}^\star(\vec{xy}), \mathcal{R}^\star(\vec{yx}))$, each $xyz \in \mathcal{P}_3(T[T_{\text{pref}}])$ to a bipartite graph $\mathcal{E}^\star(xyz)$ with bipartition $(\mathcal{R}^\star(\vec{xy}), \mathcal{R}^\star(\vec{zy}))$, and each $xyz \in \mathcal{P}_3(T)$ with $x \in \mathsf{App}_T(T_{\text{pref}})$ and $y, z \in T_{\text{pref}}$ to a bipartite graph $\mathcal{E}^\star(xyz)$ with bipartition $(\mathcal{R}(\vec{xy}), \mathcal{R}^\star(\vec{zy}))$.

We say that $\overline{u}$ *describes* a graph G' if

- $V(G') = V(G)$,
- for all $u, v \in V(G)$ with $u \notin W$ or $v \notin W$ we have $uv \in E(G')$ if and only if $uv \in E(G)$,
- for all $\vec{xy} \in \vec{E}(T[T_{\text{pref}}])$ the set $\mathcal{R}^\star(\vec{xy})$ is a representative of $\mathcal{L}(\mathcal{T})[\vec{xy}]$ in G',
- for all $xy \in E(T[T_{\text{pref}}])$ it holds that $\mathcal{E}^\star(xy) = G'[\mathcal{R}^\star(\vec{xy}), \mathcal{R}^\star(\vec{yx})]$,
- for all $xyz \in \mathcal{P}_3(T[T_{\text{pref}}])$ it holds that $\mathcal{E}^\star(xyz) = G'[\mathcal{R}^\star(\vec{xy}), \mathcal{R}^\star(\vec{zy})]$, and
- for all $xyz \in \mathcal{P}_3(T)$ with $x \in \mathsf{App}_T(T_{\text{pref}})$ and $y, z \in T_{\text{pref}}$, $\mathcal{E}^\star(xyz) = G'[\mathcal{R}(\vec{xy}), \mathcal{R}^\star(\vec{zy})]$.

Note that $\mathcal{R}^\star(\vec{xy})$ is not required to be a minimal representative and the graphs in the image of $\mathcal{E}^\star$ are allowed to have twins over the bipartition.

We observe that if $\overline{u}$ describes some graph G', then G' is uniquely determined by $\overline{u}$ and G. In particular, by making use of the $\mathcal{E}^\star(xyz) = G'[\mathcal{R}^\star(\vec{xy}), \mathcal{R}^\star(\vec{zy})]$ graphs, the description $\overline{u}$ can be turned into an annotated rank decomposition that encodes $G'[W]$. We denote $|\overline{u}| = |T_{\text{pref}}|$. We define that the *width* of $\overline{u}$ is the maximum of $\mathsf{cutrk}_{\mathcal{E}^\star(xy)}(\mathcal{R}^\star(\vec{xy}))$ over all $\vec{xy} \in \vec{E}(T[T_{\text{pref}}])$. Note that if $\overline{u}$ has width ℓ then it can be represented in space $\mathcal{O}_\ell(|T_{\text{pref}}|)$.

We say that a prefix-rebuilding update *corresponds* to an edge update description $\overline{u}$ if $\overline{u}$ describes a graph G', the update turns $\mathcal{T}$ into a rooted annotated rank decomposition $\mathcal{T}' = (T', U', \mathcal{R}', \mathcal{E}', \mathcal{F}')$ so that $\mathcal{T}'$ encodes G' and $T' = T$, and the prefix of T associated with the update is $T_{\text{pref}} \setminus L(T)$. Note that such update can change the width of an edge $xy \in E(T)$ only if W intersects both $\mathcal{L}(\mathcal{T})[\vec{xy}]$ and $\mathcal{L}(\mathcal{T})[\vec{yx}]$, in particular, only if $xy \in E(T[T_{\text{pref}}])$. It follows that the width of $\mathcal{T}'$ is at most the maximum of the widths of $\mathcal{T}$ and $\overline{u}$.

We then give the algorithm to translate edge update descriptions into descriptions of prefix-rebuilding updates.

Lemma 4.3.14. *There exists an ℓ-prefix-rebuilding data structure with overhead $\mathcal{O}_\ell(1)$ that maintains a rooted annotated rank decomposition $\mathcal{T}$ that encodes a dynamic graph G and additionally supports the following query:*

- $\mathsf{Translate}(\overline{u})$*: Given an edge update description $\overline{u}$ of width ℓ' that describes a graph G', in time $\mathcal{O}_{\ell,\ell'}(|\overline{u}|)$ returns a description of a corresponding prefix-rebuilding update.*

Proof. We maintain the rooted annotated rank decomposition $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ that encodes G by making use of Lemma 4.3.2. It remains to describe how the $\mathsf{Translate}(\overline{u})$ query is implemented.

Denote $\overline{u} = (W, T_{\text{pref}}, \mathcal{R}^\star, \mathcal{E}^\star)$ and $\mathcal{T} = (T, U, \mathcal{R}, \mathcal{E}, \mathcal{F})$. We construct $\mathcal{T}' = (T, U, \mathcal{R}', \mathcal{E}', \mathcal{F}')$ as follows. First, for every $\vec{xy} \in \vec{E}(T[T_{\text{pref}}])$ we compute a set $\mathcal{R}^{\star\star}(\vec{xy}) \subseteq \mathcal{R}^\star(\vec{xy})$ so that $\mathcal{R}^{\star\star}(\vec{xy})$ is a minimal representative of $\mathcal{L}(\mathcal{T})[\vec{xy}]$ in G'. This can be computed in $\mathcal{O}_{\ell'}(1)$ time by using $\mathcal{E}^\star(xy)$. We also compute $\mathcal{E}^{\star\star}(xy) = \mathcal{E}^\star[\mathcal{R}^{\star\star}(\vec{xy}), \mathcal{R}^{\star\star}(\vec{yx})]$ for all $xy \in E(T[T_{\text{pref}}])$. Then we construct $\mathcal{R}'$ by setting $\mathcal{R}'(\vec{xy}) = \mathcal{R}^{\star\star}(\vec{xy})$ if $\vec{xy} \in \vec{E}(T[T_{\text{pref}}])$ and $\mathcal{R}'(\vec{xy}) = \mathcal{R}(\vec{xy})$ otherwise. We also construct $\mathcal{E}'$ by setting $\mathcal{E}'(xy) = \mathcal{E}^{\star\star}(xy)$ if $xy \in E(T[T_{\text{pref}}])$ and $\mathcal{E}'(xy) = \mathcal{E}(xy)$ otherwise.

Because all edges xy so that both $\mathcal{L}(\mathcal{T})[\vec{xy}]$ and $\mathcal{L}(\mathcal{T})[\vec{yx}]$ intersect W are in $E(T[T_{\text{pref}}])$, $\mathcal{T}'$ satisfies ENC 3. It remains to construct $\mathcal{F}'$.

When both xy and yz are not in $E(T[T_{\text{pref}}])$ we let $\mathcal{F}'(xyz) = \mathcal{F}(xyz)$. This satisfies ENC 4 because $\mathcal{R}'(\vec{xy}) = \mathcal{R}(\vec{xy})$, $\mathcal{R}'(\vec{yz}) = \mathcal{R}(\vec{yz})$, $\mathcal{E}'(xy) = \mathcal{E}(xy)$, and $\mathcal{E}'(yz) = \mathcal{E}(yz)$. Let $xyz \in \mathcal{P}_3(T[T_{\text{pref}}])$ and let $u \in \mathcal{R}'(\vec{xy})$. By using $\mathcal{E}^\star(xyz)$ we can compute $N_{G'}(u) \cap \mathcal{R}^{\star\star}(\vec{zy})$, and then find $v \in \mathcal{R}'(\vec{yz}) = \mathcal{R}^{\star\star}(\vec{yz})$ so that $N_{G'}(u) \cap \mathcal{R}^{\star\star}(\vec{zy}) = N_{G'}(v) \cap \mathcal{R}^{\star\star}(\vec{zy})$ and set $\mathcal{F}(xyz)(u) = v$. This clearly satisfies ENC 4. The same idea works for computing $\mathcal{F}(xyz)$ when x or z is not in T_{pref}.

We observe that this construction can be implemented with a prefix-rebuilding update so that $T_{\text{pref}} \setminus L(T)$ is the prefix of T associated with the update. Moreover, the description of the prefix-rebuilding update can be computed in $\mathcal{O}_{\ell,\ell'}(|T_{\text{pref}}|)$ time. □

4.4 Refinement

In this section we introduce the refinement operation that will be used for improving the rank decomposition, and give the height reduction scheme by using the refinement operation.

4.4.1 Closures

The main graph-theoretic ingredient of the refinement operation is the concept of *closures*.

Let $\mathcal{T} = (T, \lambda)$ be a rooted rank decomposition of a graph G, T_{pref} a leafless prefix of T, and k a positive integer. A *k-closure* of T_{pref} is a partition $\mathcal{C}$ of $V(G)$ so that

1. for each $C \in \mathcal{C}$ there exists $a \in \mathsf{App}_T(T_{\text{pref}})$ so that $C \subseteq \mathcal{L}(\mathcal{T})[a]$, and
2. the partitioned graph $(G[\mathcal{C}], \mathcal{C})$ has rankwidth at most $2k$.

We will show that if G has rankwidth at most k, then for any T_{pref} there exists a k-closure with specific properties. This will be then used in the refinement operation.

Small closures. We say that a k-closure $\mathcal{C}$ is c-small for some integer c if for every $a \in \mathsf{App}_T(T_{\text{pref}})$ there exist at most c parts $C \in \mathcal{C}$ with $C \subseteq \mathcal{L}(\mathcal{T})[a]$. In this subsection we show that if G has rankwidth k and $\mathcal{T}$ has width ℓ, then there exists a $f(\ell)$-small k-closure of any prefix T_{pref} of T. For this we will first prove the Dealternation Lemma for rankwidth, which will be an analogue of a similar lemma for treewidth given in [BP22]. We postpone the proof of this lemma to Section 4.8, but let us state it here.

We say that a set $F \subseteq V(G)$ is a *tree factor* of $\mathcal{T}$ if $F = \mathcal{L}(\mathcal{T})[t]$ for some node $t \in V(T)$. Similarly, we say that $F \subseteq V(G)$ is a *context factor* of $\mathcal{T}$ if it is not a tree factor but it can be written as $F = F_1 \setminus F_2$, where F_1 and F_2 are tree factors of $\mathcal{T}$. A set $F \subseteq V(G)$ is a *factor* of $\mathcal{T}$ if it is either a tree factor or a context factor of $\mathcal{T}$.

Lemma 4.4.1. *There exists a function $f(\ell)$ so that if G is a graph of rankwidth k and $\mathcal{T}$ a rooted rank decomposition of G of width ℓ, then there exists a rooted rank decomposition $\mathcal{T}'$ of G of width k so that for every node $t \in V(T)$, the set $\mathcal{L}(\mathcal{T})[t]$ can be partitioned into a disjoint union of $f(\ell)$ factors of $\mathcal{T}'$.*

Next we use the Dealternation Lemma to prove the existence of $f(\ell)$-small k-closures.

Lemma 4.4.2. *There exists a function $f(\ell)$, so that if G is a graph of rankwidth k, $\mathcal{T} = (T, \lambda)$ is a rooted rank decomposition of G of width ℓ, and T_{pref} a leafless prefix of T, then there exists a $f(\ell)$-small k-closure $\mathcal{C}$ of T_{pref}.*

Proof. By applying Lemma 4.4.1, let $\mathcal{T}' = (T', \lambda')$ be a rooted rank decomposition of G of width k so that for every node $t \in V(T)$ the set $\mathcal{L}(\mathcal{T})[t]$ can be partitioned into a disjoint union of $f(\ell)$ factors of $\mathcal{T}'$. Then for each $a \in \mathsf{App}_T(T_{\text{pref}})$ let $\mathcal{C}_a$ be the partition of $\mathcal{L}(\mathcal{T})[a]$ into $f(\ell)$ parts that are factors of $\mathcal{T}'$, and let $\mathcal{C} = \bigcup_{a \in \mathsf{App}_T(T_{\text{pref}})} \mathcal{C}_a$. It remains to show that $(G[\mathcal{C}], \mathcal{C})$ has rankwidth at most $2k$.

Observe that if all factors in $\mathcal{C}$ would be tree factors, then we would directly get that $(G[\mathcal{C}], \mathcal{C})$ has rankwidth at most k by using the same rank decomposition truncated to the roots of the factors. Therefore, our goal is to change $\mathcal{T}'$ so that all factors in $\mathcal{C}$ become tree factors and the width increases to at most $2k$.

Let us say that an edge $ab \in E(T')$, where b is the parent of a in T', is *processed* if either of the following conditions holds:

- there exists a tree factor $F \in \mathcal{C}$ that intersects both $\mathcal{L}(\mathcal{T}')[\vec{ab}]$ and $\mathcal{L}(\mathcal{T}')[\vec{ba}]$; or
- $\mathcal{L}(\mathcal{T})[a]$ is a tree factor, and there is no context factor in T' of the form $\mathcal{L}(\mathcal{T})[g] \setminus \mathcal{L}(\mathcal{T})[a]$ for a strict ancestor g of b.

Otherwise, ab is *unprocessed*. We will make changes to $\mathcal{T}'$ while maintaining an invariant that every processed edge has width at most $2k$ and every unprocessed edge has width at most k. Suppose there is a node $x \in V(T')$ and a descendant y of x so that $C = \mathcal{L}(\mathcal{T}')[x] \setminus \mathcal{L}(\mathcal{T}')[y]$ is a context factor $C \in \mathcal{C}$. Note that x is not y nor a child of y because otherwise C would be a tree factor. Let p_x be the parent of x (or $p_x = x$ if x is the root of T') and p_y be the parent of y in T'. Note that all edges on the simple path between p_x and y are unprocessed.

We will change $\mathcal{T}'$ into a new rooted rank decomposition $\mathcal{T}''$ so that the number of context factors decreases but the invariant is maintained. In particular, $\mathcal{T}''$ is constructed by cutting off the subtree rooted at y by cutting the edge between y and p_y, and putting it back so that x and y have the same parent in the resulting decomposition. For this, the edge xp_x will be subdivided, or if x is the root a new root will be created so that x and y are its children. Let p' be the new common parent of x and y. Also, the degree-2 node p_y created by cutting the edge yp_y is contracted (Figure 4.1).

We observe that C becomes a tree factor in $\mathcal{T}''$, but no other factors change. This change affects only the widths of edges $ab \in E(T')$ that were on the path from p_y to p_x. Such edges ab were unprocessed, but

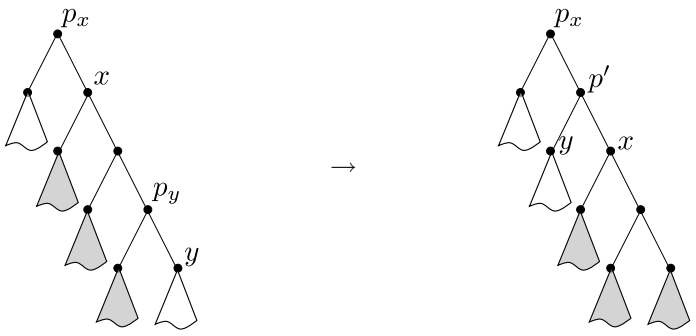


Figure 4.1: A surgery on the rank decomposition for a context factor $C = \mathcal{L}(\mathcal{T}')[x] \setminus \mathcal{L}(\mathcal{T}')[y]$. The subtrees comprising C are marked gray.

the corresponding edges $a'b'$ in T'' become processed as C becomes a tree factor. Suppose b is the parent of a. We have that $\mathcal{L}(\mathcal{T}')[y] \subseteq \mathcal{L}(\mathcal{T}')[\vec{ab}]$, $\mathsf{cutrk}(\mathcal{L}(\mathcal{T}')[\vec{ab}]) \leqslant k$, and $\mathsf{cutrk}(\mathcal{L}(\mathcal{T}')[y]) \leqslant k$. The width of the new edge $a'b'$ corresponding to ab will be $\mathsf{cutrk}(\mathcal{L}(\mathcal{T}'')[\vec{a'b'}]) = \mathsf{cutrk}(\mathcal{L}(\mathcal{T}')[\vec{ab}] \setminus \mathcal{L}(\mathcal{T}')[y])$, which by symmetry and submodularity of the cutrk function is at most $\mathsf{cutrk}(\mathcal{L}(\mathcal{T}')[\vec{ab}]) + \mathsf{cutrk}(\mathcal{L}(\mathcal{T}')[y]) \leqslant 2k$.

Therefore, the process decreases the number of context factors and maintains the invariant, and in the end we obtain a rooted rank decomposition of G of width at most $2k$ so that all parts of $\mathcal{C}$ are tree factors in the decomposition. Such decomposition can be easily turned into a rank decomposition of $(G[\mathcal{C}], \mathcal{C})$ of width at most $2k$. □

Closure linkedness. Let $A \subseteq B \subseteq V(G)$ be two sets of vertices. We say that A is *linked* into B if for all sets S with $A \subseteq S \subseteq B$ it holds that $\mathsf{cutrk}(A) \leqslant \mathsf{cutrk}(S)$. We say that a set $C \subseteq V(G)$ *cuts* a node $t \in V(T)$ if both $\mathcal{L}(\mathcal{T})[t] \cap C$ and $\mathcal{L}(\mathcal{T})[t] \setminus C$ are nonempty. Then we say that k-closure $\mathcal{C}$ of T_{pref} is *linked* if for every $C \in \mathcal{C}$ with $C \subseteq \mathcal{L}(\mathcal{T})[a]$ for $a \in \mathsf{App}_T(T_{\text{pref}})$ it holds that

1. C is linked into $\mathcal{L}(\mathcal{T})[a]$, and
2. if C cuts a descendant t of a, then $\mathsf{cutrk}(C \cup \mathcal{L}(\mathcal{T})[t]) > \mathsf{cutrk}(C)$.

We say that a k-closure $\mathcal{C}$ *cuts* a node $t \in V(T)$ if there is $C \in \mathcal{C}$ so that C cuts t, or equivalently, if more than one part in $\mathcal{C}$ intersects $\mathcal{L}(\mathcal{T})[t]$. Note that any k-closure of T_{pref} cuts all nodes in T_{pref}.

In our algorithm we will use closures that are linked. We will need to guarantee the existence of such closures and to give a method for finding them. For this, the following definition will be useful. We say that a c-small k-closure $\mathcal{C}$ of T_{pref} is *minimal* if among all c-small k-closures it

- primarily minimizes $\sum_{C \in \mathcal{C}} \mathsf{cutrk}(C)$, and
- secondarily minimizes the number of nodes of T that it cuts.

Then, the following lemma guarantees the existence of linked c-small k-closures and provides a method for finding them.

Lemma 4.4.3. *Any minimal c-small k-closure of T_{pref} is linked.*

Proof. Suppose $\mathcal{C}$ is a minimal c-small k-closure of T_{pref} that is not linked. Let $C \in \mathcal{C}$ be a part that violates the linkedness condition, in particular, with $C \subseteq \mathcal{L}(\mathcal{T})[a]$ for some $a \in \mathsf{App}_T(T_{\text{pref}})$ so that there is a set S with $C \subseteq S \subseteq \mathcal{L}(\mathcal{T})[a]$ and either

1. $\mathsf{cutrk}(S) < \mathsf{cutrk}(C)$, or
2. $\mathsf{cutrk}(S) = \mathsf{cutrk}(C)$ and $S = C \cup \mathcal{L}(\mathcal{T})[t]$ for some descendant t of a so that C cuts t.

Let us moreover fix such set S that minimizes $\mathsf{cutrk}(S)$. We will use the set S to construct a new c-small k-closure $\mathcal{C}'$ that will contradict the minimality of $\mathcal{C}$.

We let $\mathcal{C}' = \{S\} \cup \{D \setminus S \mid D \in \mathcal{C} \text{ and } D \not\subseteq S\}$. Let us first show that if $\mathcal{C}'$ is a k-closure then it contradicts the minimality of $\mathcal{C}$, and then show that it indeed is a k-closure. First, the facts that $C \subseteq S$ and this construction changes only parts that are subsets of $\mathcal{L}(\mathcal{T})[a]$ implies that $\mathcal{C}'$ is c-small. In order to bound $\sum_{C' \in \mathcal{C}'} \mathsf{cutrk}(C')$ we show the following.

Claim 4.4.4. *For all $D \in \mathcal{C} \setminus \{C\}$ it holds that $\mathsf{cutrk}(D \setminus S) \leqslant \mathsf{cutrk}(D)$.*

Proof of the claim. Note that if D is a not subset of $\mathcal{L}(\mathcal{T})[a]$, then D is disjoint from S and this holds trivially, so we can assume that $D \subseteq \mathcal{L}(\mathcal{T})[a]$. Recall that for a set X we denote $\overline{X} = V(G) \setminus X$. First we observe that

$$\mathsf{cutrk}(S) \leqslant \mathsf{cutrk}(\overline{D} \cap S) \tag{4.2}$$

because $C \subseteq \overline{D} \cap S \subseteq \mathcal{L}(\mathcal{T})[a]$, but S minimizes $\mathsf{cutrk}(S)$ among such sets. Then,

$$\begin{aligned}
\mathsf{cutrk}(D \setminus S) &= \mathsf{cutrk}(D \cap \overline{S}) = \mathsf{cutrk}(\overline{D} \cup S) && \text{(symmetry of } \mathsf{cutrk}) \\
&\leqslant \mathsf{cutrk}(\overline{D}) + \mathsf{cutrk}(S) - \mathsf{cutrk}(\overline{D} \cap S) && \text{(submodularity of } \mathsf{cutrk}) \\
&\leqslant \mathsf{cutrk}(\overline{D}) = \mathsf{cutrk}(D). && \text{(Eq. (4.2) and symmetry)}
\end{aligned}$$

◁

Claim 4.4.4 and the fact that $\mathsf{cutrk}(S) \leqslant \mathsf{cutrk}(C)$ imply that $\sum_{C' \in \mathcal{C}'} \mathsf{cutrk}(C') \leqslant \sum_{C \in \mathcal{C}} \mathsf{cutrk}(C)$. Moreover, if $\mathsf{cutrk}(S) < \mathsf{cutrk}(C)$ then in fact $\sum_{C' \in \mathcal{C}'} \mathsf{cutrk}(C') < \sum_{C \in \mathcal{C}} \mathsf{cutrk}(C)$, so in the case of Item 1 we have already contradicted the minimality of $\mathcal{C}$ and do not need to consider the secondary minimization.

Then suppose we are in the case of Item 2. First we show that if $\mathcal{C}'$ cuts some node $x \in V(T)$, then also $\mathcal{C}$ cuts x. If x is a descendant of t, then $\mathcal{L}(\mathcal{T})[x] \subseteq S$, so $\mathcal{C}'$ does not cut x. If $\mathcal{L}(\mathcal{T})[x]$ is disjoint from $\mathcal{L}(\mathcal{T})[t]$, then $D \cap \mathcal{L}(\mathcal{T})[x] = (D \setminus S) \cap \mathcal{L}(\mathcal{T})[x]$ for all $D \in \mathcal{C} \setminus \{C\}$ and $C \cap \mathcal{L}(\mathcal{T})[x] = S \cap \mathcal{L}(\mathcal{T})[x]$, so $\mathcal{C}'$ cuts x if and only if $\mathcal{C}$ cuts x. If x is an ancestor of t, then $\mathcal{C}$ cuts x because C cuts t. Then, the fact that $\mathcal{C}$ cuts t but $\mathcal{C}'$ does not cut t implies that $\mathcal{C}'$ cuts fewer nodes of T than $\mathcal{C}$.

Next we show that $\mathcal{C}'$ is a k-closure of T_{pref}. Because $S \subseteq \mathcal{L}(\mathcal{T})[a]$, it holds that for all $C' \in \mathcal{C}'$ there exists $a' \in \mathsf{App}_T(T_{\text{pref}})$ with $C' \subseteq \mathcal{L}(\mathcal{T})[a']$. It remains to bound the rankwidth of $(G[\mathcal{C}'], \mathcal{C}')$.

Claim 4.4.5. *The rankwidth of $(G[\mathcal{C}'], \mathcal{C}')$ is at most the rankwidth of $(G[\mathcal{C}], \mathcal{C})$.*

Proof of the claim. Let $\mathcal{T}^* = (T^*, \lambda^*)$ be an optimum-width rank decomposition of $(G[\mathcal{C}], \mathcal{C})$. We modify $\mathcal{T}^*$ into a rank decomposition $\mathcal{T}' = (T', \lambda')$ of $(G[\mathcal{C}'], \mathcal{C}')$ by simply mapping $S \in \mathcal{C}'$ to the leaf to which C was mapped, and for each $D \setminus S \in \mathcal{C}'$ mapping $D \setminus S$ to the leaf to which D was mapped. This could create some leaves to which no parts of $\mathcal{C}'$ are mapped, so finally we iteratively remove leaves with no mapped parts and contract edges of degree 2.

Consider an edge $x'y' \in E(T')$, and suppose w.l.o.g. that $S \subseteq \mathcal{L}(\mathcal{T}')[\vec{x'y'}]$. Then there exists an oriented edge $\vec{xy} \in \vec{E}(T)$ so that $C \subseteq \mathcal{L}(\mathcal{T}^*)[\vec{xy}]$ and $\mathcal{L}(\mathcal{T}')[\vec{x'y'}] = \mathcal{L}(\mathcal{T}^*)[\vec{xy}] \cup S$. Therefore it suffices to show that $\mathsf{cutrk}(\mathcal{L}(\mathcal{T}^*)[\vec{xy}] \cup S) \leqslant \mathsf{cutrk}(\mathcal{L}(\mathcal{T}^*)[\vec{xy}])$. First, we note that

$$\mathsf{cutrk}(S) \leqslant \mathsf{cutrk}(\mathcal{L}(\mathcal{T}^*)[\vec{xy}] \cap S) \tag{4.3}$$

because $C \subseteq \mathcal{L}(\mathcal{T}^*)[\vec{xy}] \cap S \subseteq \mathcal{L}(\mathcal{T}^*)[a]$, but S minimizes $\mathsf{cutrk}(S)$ among such sets. Then,

$$\begin{aligned}
\mathsf{cutrk}(\mathcal{L}(\mathcal{T}^*)[\vec{xy}] \cup S) &\leqslant \mathsf{cutrk}(\mathcal{L}(\mathcal{T}^*)[\vec{xy}]) + \mathsf{cutrk}(S) - \mathsf{cutrk}(\mathcal{L}(\mathcal{T}^*)[\vec{xy}] \cap S) && \text{(submodularity)} \\
&\leqslant \mathsf{cutrk}(\mathcal{L}(\mathcal{T}^*)[\vec{xy}]). && \text{(Eq. (4.3))}
\end{aligned}$$

◁

This finishes the proof that $\mathcal{C}'$ is a c-small k-closure that contradicts the minimality of $\mathcal{C}$. □

We then observe the main consequence of closure linkedness.

Lemma 4.4.6. *Let $\mathcal{C}$ be a k-closure of T_{pref} that is linked. If $C \in \mathcal{C}$ and C cuts a node $t \in V(T) \setminus T_{\text{pref}}$, then it holds that $\mathsf{cutrk}(C \cap \mathcal{L}(\mathcal{T})[t]) < \mathsf{cutrk}(\mathcal{L}(\mathcal{T})[t])$.*

Proof. Suppose $\mathsf{cutrk}(C \cap \mathcal{L}(\mathcal{T})[t]) \geqslant \mathsf{cutrk}(\mathcal{L}(\mathcal{T})[t])$. Then from submodularity it follows that $\mathsf{cutrk}(C \cup \mathcal{L}(\mathcal{T})[t]) \leqslant \mathsf{cutrk}(C)$, which contradicts that $\mathcal{C}$ is linked. □

Computing closures. For a k-closure $\mathcal{C}$ of T_{pref}, we denote by $\mathsf{cut}_T(\mathcal{C})$ the set of nodes of T that are cut by $\mathcal{C}$. Note that $\mathsf{cut}_T(\mathcal{C})$ is a prefix of T and $T_{\text{pref}} \subseteq \mathsf{cut}_T(\mathcal{C})$. We wish to manipulate k-closures in time proportional to $|\mathsf{cut}_T(\mathcal{C})|$. Let $C \in \mathcal{C}$. The *appendix edge set* $\mathsf{aes}_T(C)$ of C is the set $\mathsf{aes}_T(C) = \{\vec{ap} \in \vec{\mathsf{App}}_T(\mathsf{cut}_T(\mathcal{C})) \mid \mathcal{L}(\mathcal{T})[\vec{ap}] \subseteq C\} \subseteq \vec{\mathsf{App}}_T(\mathsf{cut}_T(\mathcal{C}))$ of appendix edges of $\mathsf{cut}_T(\mathcal{C})$ that correspond to C. Then, we define the *appendix edge partition* $\mathsf{aep}_T(\mathcal{C})$ of $\mathcal{C}$ to be the partition $\mathsf{aep}_T(\mathcal{C}) = \{\mathsf{aes}_T(C) \mid C \in \mathcal{C}\}$ of $\vec{\mathsf{App}}_T(\mathsf{cut}_T(\mathcal{C}))$. Note that $|\vec{\mathsf{App}}_T(\mathsf{cut}_T(\mathcal{C}))| = |\mathsf{cut}_T(\mathcal{C})| + 1$, so the appendix edge partition can be represented in space $\mathcal{O}(|\mathsf{cut}_T(\mathcal{C})|)$.

We will use the following prefix-rebuilding data structure for computing closures. We defer the proof to Section 4.9, but the idea will be to adapt the dynamic programming of [JKO21] for computing optimal rank decompositions to our setting.

Lemma 4.4.7. *There is an ℓ-prefix-rebuilding data structure that takes integer parameters $c \geqslant 1$ and $k \leqslant \ell$ at initialization, has overhead $\mathcal{O}_{c,\ell}(1)$, maintains a rooted annotated rank decomposition $\mathcal{T}$, and additionally supports the following query:*

- $\mathsf{Closure}(T_{\text{pref}})$*: Given a prefix T_{pref} of $\mathcal{T}$, either in time $\mathcal{O}_\ell(|T_{\text{pref}}|)$ returns that no c-small k-closure of T_{pref} exists, or for a minimal c-small k-closure $\mathcal{C}$ of T_{pref} in time $\mathcal{O}_\ell(|\mathsf{cut}_T(\mathcal{C})|)$ returns*
  - *the sets $\mathsf{cut}_T(\mathcal{C})$ and $\mathsf{aep}_T(\mathcal{C})$, and*
  - *a rooted rank decomposition (T^*, λ^*) of $(G[\mathcal{C}], \mathcal{C})$ of width at most $2k$, where λ^* is represented as a function $\lambda\colon \mathsf{aep}_T(\mathcal{C}) \to \vec{L}(T^\star)$.*

4.4.2 Refinement operation

We start by introducing the potential function we use for the amortized analysis of the algorithm.

In a rooted rank decomposition $\mathcal{T} = (T, \lambda)$ of a graph G, let us say that the *width* of a node $t \in V(T)$ is the width of the edge between the node and the parent, and denote it by $\mathsf{width}_{\mathcal{T},G}(t) = \mathsf{cutrk}_G(\mathcal{L}(\mathcal{T})[t])$. The width of the root node is defined to be 0. Let f be the function from Lemma 4.4.2. Then we let the ℓ-potential of t with respect to G be

$$\Phi_{\ell,\mathcal{T},G}(t) = (2 \cdot f(\ell))^{\mathsf{width}_{\mathcal{T},G}(t)} \cdot \mathsf{height}_T(t),$$

and the ℓ-potential of $\mathcal{T}$ with respect to G be

$$\Phi_{\ell,G}(\mathcal{T}) = \sum_{t \in V(T)} \Phi_{\ell,\mathcal{T},G}(t).$$

We will omit the graph G from the subscript in these notations if it is clear from the context.

For a set of nodes $S \subseteq V(T)$ we will denote $\mathsf{height}_T(S) = \sum_{t \in S} \mathsf{height}_T(t)$ and $\Phi_{\ell,\mathcal{T}}(S) = \sum_{t \in S} \Phi_{\ell,\mathcal{T}}(t)$.

Then we give the refinement operation formulated as a prefix-rebuilding data structure.

Lemma 4.4.8. *Let $k \in \mathbb{N}$ and $\ell \geqslant 4k+1$. There exists an ℓ-prefix-rebuilding data structure with overhead $\mathcal{O}_\ell(1)$ that maintains a rooted annotated rank decomposition $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ that encodes a dynamic graph G and supports the following operation:*

- $\mathsf{Refine}(T_{\text{pref}})$*: Given a leafless prefix T_{pref} of T so that T_{pref} contains all nodes of width $> 4k$, returns either that the rankwidth of G is greater than k, or a description $\overline{u}$ of a prefix-rebuilding update so that the rooted rank decomposition $\mathcal{T}'$ to which $\mathcal{T}$ corresponds to after applying $\overline{u}$ has the following properties:*
  1. *$\mathcal{T}'$ encodes G,*
  2. *$\mathcal{T}'$ has width at most $4k$, and*
  3. *the following inequality holds:*

$$\Phi_\ell(\mathcal{T}') \leqslant \Phi_\ell(\mathcal{T}) - \mathsf{height}_T(T_{\text{pref}}) + \log|\mathcal{T}| \cdot \mathcal{O}_\ell(|T_{\text{pref}}| + \mathsf{height}_T(\mathsf{App}_T(T_{\text{pref}}))).$$

  In the former case, the running time of $\mathsf{refine}(T_{\text{pref}})$ is $\mathcal{O}_\ell(|T_{\text{pref}}|)$, and in the latter case the running time and therefore also $|\overline{u}|$ is bounded by

$$\log|\mathcal{T}| \cdot \mathcal{O}_\ell(\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}') + \log|\mathcal{T}| \cdot (|T_{\text{pref}}| + \mathsf{height}_T(\mathsf{App}_T(T_{\text{pref}})))).$$

Proof. We use Lemma 4.3.2 for maintaining a representation of $\mathcal{T}$. Let $c = f(\ell)$, where f is the function from Lemma 4.4.2, in particular, so that if G has rankwidth at most k then there exists a c-small k-closure of T_{pref}. We maintain the ℓ-prefix-rebuilding data structure from Lemma 4.4.7 with these values of c and k and the ℓ-prefix-rebuilding data structure from Lemma 4.3.11, by simply relaying all prefix-rebuilding updates also to these data structures. In particular, they will always store the exactly same rooted annotated rank decomposition $\mathcal{T}$.

Then we describe how the $\mathsf{Refine}(T_{\text{pref}})$ operation is implemented. First we apply the $\mathsf{Closure}(T_{\text{pref}})$ operation of the data structure of Lemma 4.4.7. If it returns that no c-small k-closure of T_{pref} exists, then by Lemma 4.4.2 the rankwidth of G is more than k and we can return immediately. Otherwise, it returns a representation of a minimal c-small k-closure $\mathcal{C}$ of T_{pref}, containing in particular the sets $\mathsf{cut}_T(\mathcal{C})$ and $\mathsf{aep}_T(\mathcal{C})$, and a rooted rank decomposition $\mathcal{T}^{**} = (T^{**}, \lambda^{**})$ of $(G[\mathcal{C}], \mathcal{C})$ of width at most $2k$, where λ^{**} is represented as a function $\lambda^{**} \colon \mathsf{aep}_T(\mathcal{C}) \to \vec{L}(T^{**})$. We immediately use Lemma 4.2.2 to turn $\mathcal{T}^{**}$ into a rooted rank decomposition $\mathcal{T}^{*} = (T^{*}, \lambda^{*})$ of width at most $4k$ and height at most $\mathcal{O}(\log n)$.

Let us describe the construction of the rooted rank decomposition $\mathcal{T}' = (T', \lambda')$. For this, we denote by (T, λ) the rooted rank decomposition that $\mathcal{T}$ corresponds to.

First, for each part $C \in \mathcal{C}$ (represented by $\mathsf{aes}_T(C) \in \mathsf{aep}_T(\mathcal{C})$) we construct a rooted rank decomposition $\mathcal{T}_C = (T_C, \lambda_C)$ as follows. The tree T_C is obtained by first taking the subtree of T induced by nodes $t \in V(T)$ with $\mathcal{L}(\mathcal{T})[t] \cap C \neq \emptyset$ and iteratively contracting all resulting degree-2 nodes. Then we set $\lambda_C := \lambda|_C$. Now $\mathcal{T}_C$ is a rooted rank decomposition of $G[C]$ so that for every node $t \in V(T_C)$ there exists a node $t' \in V(T)$ with $\mathcal{L}(\mathcal{T}_C)[t] = \mathcal{L}(\mathcal{T})[t'] \cap C$. Then, the rooted rank decomposition $\mathcal{T}' = (T', \lambda')$ is constructed by taking $\mathcal{T}^*$ and for each $C \in \mathcal{C}$ attaching $\mathcal{T}_C$ to T^* by identifying the root of T_C with the leaf $\lambda^*(C)$ of T^*. It can be observed that $\mathcal{T}'$ is a rooted rank decomposition of G, for every $t \in V(T') \cap V(T^*)$ it holds that $\mathcal{L}(\mathcal{T}')[t] = \mathcal{L}(\mathcal{T}^*)[t]$, and for every $t \in V(T') \cap V(T_C)$ for $C \in \mathcal{C}$ it holds that $\mathcal{L}(\mathcal{T}')[t] = \mathcal{L}(\mathcal{T}_C)[t]$.

Claim 4.4.9. *A description $\overline{u}$ of a prefix-rebuilding update that turns $\mathcal{T}$ into a rooted annotated rank decomposition that corresponds to $\mathcal{T}'$ can be computed in $\mathcal{O}_\ell(|\mathsf{cut}_T(\mathcal{C})| \log |\mathcal{T}|)$ time.*

Proof of the claim. We will show that such a prefix-rearrangement description can be computed in time $\mathcal{O}_\ell(|\mathsf{cut}_T(\mathcal{C})|)$. This then implies the claim by applying the $\mathsf{Translate}$ query of the prefix-rebuilding data structure of Lemma 4.3.11.

Recall that for every $t \in \mathsf{App}_T(\mathsf{cut}_T(\mathcal{C}))$ we have that $\mathcal{L}(\mathcal{T})[t] \subseteq C$ for some $C \in \mathcal{C}$, so the subtree rooted at t can be copied verbatim from $\mathcal{T}$ to $\mathcal{T}'$. It follows that we can set the prefix of the prefix-rearrangement description to be $\mathsf{cut}_T(\mathcal{C})$. It remains to construct the tree $T^\star$ of the description. We construct it by first taking T^*, and then for every leaf of it that corresponds to a part $C \in \mathcal{C}$ constructing the prefix of T_C that is not copied verbatim. In particular, let $C \subseteq \mathcal{L}(\mathcal{T})[a]$ for $a \in \mathsf{App}_T(T_{\text{pref}})$, and denote by $\mathsf{cut}_{T,a}(\mathcal{C}) \subseteq \mathsf{cut}_T(\mathcal{C})$ the nodes that are cut by $\mathcal{C}$ and are descendants of a. By using the mapping $\lambda^* \colon \mathsf{aep}_T(\mathcal{C}) \to \vec{L}(T^*)$ we can construct the prefix of T_C that is not copied verbatim in $\mathcal{O}(|\mathsf{cut}_{T,a}(\mathcal{C})|)$ time, also finding out how the subtrees that are copied verbatim are attached to the prefix. Because $\mathcal{C}$ is c-small, the total time sums up to $\mathcal{O}(c \cdot |\mathsf{cut}_T(\mathcal{C})|) = \mathcal{O}_\ell(|\mathsf{cut}_T(\mathcal{C})|)$. ◁

For bounding the width of $\mathcal{T}'$ and analyzing the potential, let us relate the nodes in each of the trees T_C to nodes in T. Let us denote by $\pi_C \colon V(T_C) \to V(T)$ the mapping that maps each node $t \in V(T_C)$ to a node $t' \in V(T)$ so that $\mathcal{L}(\mathcal{T}_C)[t] = \mathcal{L}(\mathcal{T})[t'] \cap C$ and t' minimizes $\mathsf{height}_T(t')$ under this condition (this defines $\pi_C(t)$ uniquely). Note that π_C is an injection, and if $a \in \mathsf{App}_T(T_{\text{pref}})$ so that $C \subseteq \mathcal{L}(\mathcal{T})[a]$, then $\pi_C(t)$ is a descendant of a for all $t \in V(T_C)$.

Claim 4.4.10. *For all $t \in V(T_C)$ it holds that $\mathsf{cutrk}_G(\mathcal{L}(\mathcal{T}_C)[t]) \leqslant \mathsf{cutrk}_G(\mathcal{L}(\mathcal{T})[\pi_C(t)])$, and moreover if $\pi_C(t) \in \mathsf{cut}_T(\mathcal{C})$ then $\mathsf{cutrk}_G(\mathcal{L}(\mathcal{T}_C)[t]) < \mathsf{cutrk}_G(\mathcal{L}(\mathcal{T})[\pi_C(t)])$.*

Proof of the claim. First suppose that $\pi_C(t) \notin \mathsf{cut}_T(\mathcal{C})$. In that case, $\mathcal{L}(\mathcal{T}_C)[t] = \mathcal{L}(\mathcal{T})[\pi_C(t)]$ because C intersects $\mathcal{L}(\mathcal{T})[\pi_C(t)]$ but does not cut $\pi_C(t)$. Then, if $\pi_C(t) \in \mathsf{cut}_T(\mathcal{C})$, Lemma 4.4.6 implies that $\mathsf{cutrk}_G(\mathcal{L}(\mathcal{T}_C)[t]) < \mathsf{cutrk}_G(\mathcal{L}(\mathcal{T})[\pi_C(t)])$ because $\mathcal{C}$ is linked because it is minimal. ◁

It follows that $\mathcal{T}'$ has width at most $4k$: All nodes in $V(T^*)$ have width at most $4k$, and for all $C \in \mathcal{C}$ and $t \in V(T_C)$ we have that $\pi_C(t) \notin T_{\text{pref}}$ implying $\mathsf{cutrk}_G(\mathcal{L}(\mathcal{T})[\pi_C(t)]) \leqslant 4k$ and therefore $\mathsf{cutrk}_G(\mathcal{L}(\mathcal{T}')[t]) \leqslant 4k$.

To bound $\Phi_\ell(\mathcal{T}')$, first note that

$$\Phi_\ell(\mathcal{T}') = \Phi_{\ell,\mathcal{T}'}(V(T^*) \setminus L(T^*)) + \sum_{C \in \mathcal{C}} \Phi_\ell(\mathcal{T}_C).$$

Let us first bound the latter term.

Claim 4.4.11.

$$\sum_{C\in\mathcal{C}} \Phi_\ell(\mathcal{T}_C) \leqslant \Phi_\ell(\mathcal{T}) - \Phi_{\ell,\mathcal{T}}(T_{\mathrm{pref}}) - |\mathsf{cut}_T(\mathcal{C}) \setminus T_{\mathrm{pref}}|$$

Proof of the claim. We observe that $\mathsf{height}_{T_C}(t) \leqslant \mathsf{height}_T(\pi_C(t))$ for all $C \in \mathcal{C}$ and $t \in V(T_C)$, which implies $\Phi_{\ell,\mathcal{T}_C}(t) \leqslant \Phi_{\ell,\mathcal{T}}(\pi_C(t))$ for all such t, and moreover when $\pi_C(t) \in \mathsf{cut}_T(\mathcal{C})$ it holds that

$$\begin{aligned}\Phi_{\ell,\mathcal{T}_C}(t) &= (2\cdot f(\ell))^{\mathsf{width}_{\mathcal{T}_C}(t)} \cdot \mathsf{height}_{T_C}(t)\\ &\leqslant (2\cdot f(\ell))^{\mathsf{width}_{\mathcal{T}}(\pi_C(t))-1} \cdot \mathsf{height}_T(\pi_C(t))\\ &\leqslant \Phi_{\ell,\mathcal{T}}(\pi_C(t))/(2\cdot f(\ell)).\end{aligned}$$

Then, for $x \in V(T)$, let us denote by $\pi^{-1}(x)$ the set of nodes in $\bigcup_{C\in\mathcal{C}} V(T_C)$ that are mapped to x by π_C, i.e., $\pi^{-1}(x) = \{t \in V(T_C) \mid C \in \mathcal{C} \text{ and } \pi_C(t) = x\}$. We observe that if $x \in T_{\mathrm{pref}}$ then $\pi^{-1}(x) = \emptyset$, if $x \in \mathsf{cut}_T(\mathcal{C}) \setminus T_{\mathrm{pref}}$ then $|\pi^{-1}(x)| \leqslant f(\ell)$ because $\mathcal{C}$ is c-small for $c = f(\ell)$, and if $x \in V(T) \setminus \mathsf{cut}_T(\mathcal{C})$ then $|\pi^{-1}(x)| = 1$.

By putting these two observations together we obtain

$$\begin{aligned}\sum_{C\in\mathcal{C}} \Phi_\ell(\mathcal{T}_C) &\leqslant \Phi_{\ell,\mathcal{T}}(V(T)\setminus \mathsf{cut}_T(\mathcal{C})) + \sum_{t\in \mathsf{cut}_T(\mathcal{C})\setminus T_{\mathrm{pref}}} f(\ell)\cdot\Phi_{\ell,\mathcal{T}}(t)/(2\cdot f(\ell))\\ &\leqslant \Phi_{\ell,\mathcal{T}}(V(T)\setminus \mathsf{cut}_T(\mathcal{C})) + \Phi_{\ell,\mathcal{T}}(\mathsf{cut}_T(\mathcal{C})\setminus T_{\mathrm{pref}})/2\\ &\leqslant \Phi_\ell(\mathcal{T}) - \Phi_{\ell,\mathcal{T}}(T_{\mathrm{pref}}) - |\mathsf{cut}_T(\mathcal{C})\setminus T_{\mathrm{pref}}|.\end{aligned}$$

◁

Then we bound the former term.

Claim 4.4.12.

$$\Phi_{\ell,\mathcal{T}'}(V(T^*)) \leqslant \log|\mathcal{T}|\cdot\mathcal{O}_\ell(|T_{\mathrm{pref}}| + \mathsf{height}_T(\mathsf{App}_T(T_{\mathrm{pref}})))$$

Proof of the claim. For each node $t \in V(T^*)$, let $\Gamma(t) \in \mathcal{C}$ be a part of $\mathcal{C}$ so that $\lambda^*(\Gamma(t)) \in \vec{L}(T^*)[t]$, and among such parts $\Gamma(t)$ maximizes $\mathsf{height}(T_{\Gamma(t)})$. Such $\Gamma(t)$ is not necessary unique, in which case we assign some such $\Gamma(t)$ arbitrarily. Because the height of T^* is at most $\mathcal{O}(\log|\mathcal{C}|) \leqslant \mathcal{O}(\log|\mathcal{T}|)$, we have that $\mathsf{height}_{T'}(t) \leqslant \mathcal{O}(\log|\mathcal{T}|) + \mathsf{height}(T_{\Gamma(t)})$, implying that

$$\mathsf{height}_{T'}(V(T^*)) \leqslant \mathcal{O}(|V(T^*)|\log|\mathcal{T}|) + \sum_{t\in V(T^*)} \mathsf{height}(T_{\Gamma(t)}).$$

We observe that if $\Gamma(t) \subseteq \mathcal{L}(\mathcal{T})[a]$ for $a \in \mathsf{App}_T(T_{\mathrm{pref}})$, then $\mathsf{height}(T_{\Gamma(t)}) \leqslant \mathsf{height}_T(a)$. Because $\mathcal{C}$ is c-small, for each $a \in \mathsf{App}_T(T_{\mathrm{pref}})$ there are at most c such sets $\Gamma(t)$. Also, because T^* has height at most $\mathcal{O}(\log|\mathcal{T}|)$, each $C \in \mathcal{C}$ can be the set $\Gamma(t)$ for at most $\mathcal{O}(\log|\mathcal{T}|)$ nodes in $V(T^*)$. From these observations it follows that

$$\begin{aligned}\sum_{t\in V(T^*)} \mathsf{height}(T_{\Gamma(t)}) &\leqslant \sum_{a\in\mathsf{App}_T(T_{\mathrm{pref}})} \mathcal{O}(\mathsf{height}_T(a)\cdot c\cdot\log|\mathcal{T}|)\\ &\leqslant \mathcal{O}_\ell(\mathsf{height}_T(\mathsf{App}_T(T_{\mathrm{pref}}))\cdot\log|\mathcal{T}|)\end{aligned}$$

Then the conclusion of the claim follows from $|V(T^*)| \leqslant 2\cdot|\mathcal{C}| \leqslant 4c\cdot|T_{\mathrm{pref}}| \leqslant \mathcal{O}_\ell(|T_{\mathrm{pref}}|)$. ◁

By putting Claims 4.4.11 and 4.4.12 together, we obtain

$$\begin{aligned}\Phi_\ell(\mathcal{T}') \leqslant \Phi_\ell(\mathcal{T}) &- \Phi_{\ell,\mathcal{T}}(T_{\mathrm{pref}}) - |\mathsf{cut}_T(\mathcal{C})\setminus T_{\mathrm{pref}}|\\ &+ \log|\mathcal{T}|\cdot\mathcal{O}_\ell(|T_{\mathrm{pref}}| + \mathsf{height}_T(\mathsf{App}_T(T_{\mathrm{pref}}))),\end{aligned} \tag{4.4}$$

which by $\Phi_{\ell,\mathcal{T}}(T_{\mathrm{pref}}) \geqslant \mathsf{height}_T(T_{\mathrm{pref}})$ implies the desired potential bound of Item 3.

Let us then prove the running time bound of $\mathsf{Refine}(T_{\mathrm{pref}})$ in the lemma statement. The algorithm consists of calling the data structure of Lemma 4.4.7, applying Lemma 4.2.2, and constructing the description $\overline{u}$ of the prefix-rebuilding update, which by Claim 4.4.9 all take at most $\mathcal{O}_\ell(|\mathsf{cut}_T(\mathcal{C})|\log|\mathcal{T}|)$ time. We can rearrange Eq. (4.4) into

$$\Phi_{\ell,\mathcal{T}}(T_{\mathrm{pref}}) + |\mathsf{cut}_T(\mathcal{C})\setminus T_{\mathrm{pref}}| \leqslant \Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}') + \log|\mathcal{T}|\cdot\mathcal{O}_\ell(|T_{\mathrm{pref}}| + \mathsf{height}_T(\mathsf{App}_T(T_{\mathrm{pref}}))),$$

which by $|\mathsf{cut}_T(\mathcal{C})| \leqslant \Phi_{\ell,\mathcal{T}}(T_{\mathrm{pref}}) + |\mathsf{cut}_T(\mathcal{C}) \setminus T_{\mathrm{pref}}|$ implies

$$|\mathsf{cut}_T(\mathcal{C})| \leqslant \Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}') + \log|\mathcal{T}| \cdot \mathcal{O}_\ell(|T_{\mathrm{pref}}| + \mathsf{height}_T(\mathsf{App}_T(T_{\mathrm{pref}}))),$$

which yields the desired running time. □

4.4.3 Height reduction

The main combinatorial ingredient for our height reduction scheme is the following lemma that we proved in Chapter 3.

Lemma 3.5.2. *Let $c \geqslant 2$ and T be a binary tree with at most N nodes. If the height of T is at least $2^{\Omega(\sqrt{\log N \log c})}$, then there exists a nonempty prefix T_{pref} of T so that*

$$c \cdot \left(|T_{\mathrm{pref}}| + \sum_{a \in \mathsf{App}(T_{\mathrm{pref}})} \mathsf{height}_T(a) \right) \leqslant \sum_{x \in T_{\mathrm{pref}}} \mathsf{height}_T(x). \tag{3.13}$$

Moreover, if we can access the height of each node of T in constant time, then such T_{pref} can be computed in time $\mathcal{O}(|T_{\mathrm{pref}}|)$.

Then, our height reduction scheme is formulated as a prefix-rebuilding data structure as follows.

Lemma 4.4.13. *Let $k \in \mathbb{N}$ and $\ell \geqslant 4k+1$. There exists an ℓ-prefix-rebuilding data structure with overhead $\mathcal{O}_\ell(1)$ that maintains a rooted annotated rank decomposition $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ that encodes a dynamic graph G of rankwidth at most k, supports the operation $\mathsf{Refine}(T_{\mathrm{pref}})$ from Lemma 4.4.8, and additionally supports the following operation under the promise that the width of $\mathcal{T}$ is at most $4k$:*

- $\mathsf{ImproveHeight}()$*: Updates $\mathcal{T}$ through a sequence of prefix-rebuilding updates so that the resulting annotated rank decomposition $\mathcal{T}'$ encodes G, has height $2^{\mathcal{O}_\ell(\sqrt{\log n \log\log n})}$ and width at most $4k$, and returns the corresponding sequence of descriptions of prefix-rebuilding updates. All of the intermediate decompositions also have width at most $4k$. It holds that $\Phi_\ell(\mathcal{T}') \leqslant \Phi_\ell(\mathcal{T})$ and the running time of* $\mathsf{ImproveHeight}()$ *is $\mathcal{O}_\ell((\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}'))\log|\mathcal{T}|)$.*

Proof. We maintain a representation of $\mathcal{T}$ by Lemma 4.3.2, and additionally maintain the prefix-rebuilding data structures $\mathbb{D}^{\mathsf{height}}$ given by Lemma 4.3.3 and $\mathbb{D}^{\mathsf{refine}}$ given by Lemma 4.4.8, so that all prefix-rebuilding updates that are applied to $\mathcal{T}$ are also relayed to $\mathbb{D}^{\mathsf{height}}$ and $\mathbb{D}^{\mathsf{refine}}$, in particular, so that they store the exactly same rooted annotated rank decomposition $\mathcal{T}$. The $\mathsf{Refine}(T_{\mathrm{pref}})$ operation is implemented by using $\mathbb{D}^{\mathsf{refine}}$. It remains to implement the $\mathsf{ImproveHeight}()$ operation.

Let us choose $c_0 = \mathcal{O}_\ell(1)$ based on ℓ so that the inequality of Item 3 in Lemma 4.4.8 is true in the form

$$\Phi_\ell(\mathcal{T}') \leqslant \Phi_\ell(\mathcal{T}) - \mathsf{height}_T(T_{\mathrm{pref}}) + \frac{c_0}{2} \cdot \log|\mathcal{T}| \cdot (|T_{\mathrm{pref}}| + \mathsf{height}_T(\mathsf{App}_T(T_{\mathrm{pref}}))). \tag{4.5}$$

Let $c = c_0 \cdot \log|\mathcal{T}|$. First, if $\mathsf{height}(\mathcal{T}) \leqslant 2^{\mathcal{O}(\sqrt{\log n \log c})} \leqslant 2^{\mathcal{O}_\ell(\sqrt{\log n \log\log n})}$, where the constant in the $\mathcal{O}$-notation depends on the constant in the Ω-notation in Lemma 3.5.2, then the height of $\mathcal{T}$ is already small enough and we do not update $\mathcal{T}$ and return an empty sequence of descriptions of prefix-rebuilding updates. Otherwise, we use the algorithm from Lemma 3.5.2 with the $\mathsf{height}_T(t)$ operation supplied from $\mathbb{D}^{\mathsf{height}}$ to find a nonempty prefix T_{pref} of T so that

$$c_0 \cdot \log|\mathcal{T}| \cdot (|T_{\mathrm{pref}}| + \mathsf{height}_T(\mathsf{App}_T(T_{\mathrm{pref}}))) \leqslant \mathsf{height}_T(T_{\mathrm{pref}}). \tag{4.6}$$

Then we apply the Refine operation with this T_{pref} and apply the resulting prefix-rebuilding update to $\mathcal{T}$, relaying it also to $\mathbb{D}^{\mathsf{height}}$ and $\mathbb{D}^{\mathsf{refine}}$. By putting Eqs. (4.5) and (4.6) together, we obtain that the resulting decomposition $\mathcal{T}'$ satisfies

$$\Phi_\ell(\mathcal{T}') \leqslant \Phi_\ell(\mathcal{T}) - \frac{c_0}{2} \cdot \log|\mathcal{T}| \cdot (|T_{\mathrm{pref}}| + \mathsf{height}_T(\mathsf{App}_T(T_{\mathrm{pref}}))).$$

Because T_{pref} is nonempty, we have in particular $\Phi_\ell(\mathcal{T}') < \Phi_\ell(\mathcal{T})$. The time complexity of the application of Lemma 3.5.2 is $\mathcal{O}(|T_{\mathrm{pref}}|) = \mathcal{O}(\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}'))$. The time complexity of the application of the Refine operation and the size of the description of the update is bounded by

$$\begin{aligned}&\log|\mathcal{T}| \cdot \mathcal{O}_\ell(\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}') + \log|\mathcal{T}| \cdot (|T_{\mathrm{pref}}| + \mathsf{height}_T(\mathsf{App}_T(T_{\mathrm{pref}}))))\\ &= \mathcal{O}_\ell((\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}'))\log|\mathcal{T}|),\end{aligned}$$

which is also the time it takes to apply the prefix-rebuilding updates, implying that the total time complexity is $\mathcal{O}_\ell((\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}'))\log|\mathcal{T}|)$. The width of $\mathcal{T}'$ is guaranteed to be at most $4k$ by Lemma 4.4.8.

Applying this update did not necessarily decrease the height of $\mathcal{T}$, but we can run it again repeatedly until it decreases the height to $2^{\mathcal{O}_\ell(\sqrt{\log n \log\log n})}$. Because $\Phi_\ell(\mathcal{T}') < \Phi_\ell(\mathcal{T})$, the number of such iterations is bounded by $\Phi_\ell(\mathcal{T})$, and moreover, as the running time of a single iteration is bounded by $\mathcal{O}_\ell((\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}'))\log|\mathcal{T}|)$, the running time of any sequence of such iterations is bounded by $\mathcal{O}_\ell((\Phi_\ell(\mathcal{T}) - \Phi_\ell(\mathcal{T}''))\log|\mathcal{T}|)$, where $\mathcal{T}''$ is the final decomposition. Because all of the updates were obtained from the Refine operation, all of the rank decompositions in the sequence of updates have width at most $4k$. □

4.5 Automata

In this section we define rank decomposition automata in order to formalize and unify dynamic programming working on rank decompositions. We give a prefix-rebuilding data structure to maintain the runs of rank decomposition automata, give a construction of rank decomposition automata from CMSO_1 sentences (using the construction for cliquewidth by [CMR00] as a black-box), and finally give our framework for performing edge updates using CMSO_1.

4.5.1 Rank decomposition automata

We will define a *rank decomposition automaton*, which is an automaton that processes annotated rank decompositions. Our definitions will be for unrooted annotated rank decompositions, in particular, so that they are suited for computing dynamic programming tables directed in both directions on edges. While these definitions allow annotated rank decompositions that encode partitioned graphs with nontrivial partitions, they are usually used with annotated rank decompositions that encode graphs. Let us start with some auxiliary definitions.

We say that a *transition signature* of width ℓ is a tuple $\tau = (S_\tau, U_\tau, \mathcal{R}_\tau, \mathcal{E}_\tau, \mathcal{F}_\tau)$, where

- S_τ is a tree with three leaf nodes and one nonleaf node,
- U_τ is a set of size at most $6 \cdot 2^\ell$,
- $\mathcal{R}_\tau$ is a function that maps each oriented edge $\vec{xy} \in \vec{E}(S_\tau)$ to a nonempty set $\mathcal{R}_\tau(\vec{xy}) \subseteq U_\tau$,
- $\mathcal{E}_\tau$ is a function that maps each edge $xy \in E(S_\tau)$ to a bipartite graph $\mathcal{E}_\tau(xy)$ with bipartition $(\mathcal{R}_\tau(\vec{xy}), \mathcal{R}_\tau(\vec{yx}))$, with no twins over this bipartition, and with $\mathsf{cutrk}_{\mathcal{E}_\tau(xy)}(\mathcal{R}_\tau(\vec{xy})) \leqslant \ell$, and
- $\mathcal{F}_\tau$ is a function that maps each path of length three $xyz \in \mathcal{P}_3(S_\tau)$ in S_τ to a function $\mathcal{F}_\tau(xyz) : \mathcal{R}_\tau(\vec{xy}) \to \mathcal{R}_\tau(\vec{yz})$.

Let $\mathcal{T} = (T, U, \mathcal{R}, \mathcal{E}, \mathcal{F})$ be an annotated rank decomposition and $\vec{tp} \in \vec{E}(T) \setminus \vec{L}(T)$ a nonleaf oriented edge of T with children $\vec{c_1t}$ and $\vec{c_2t}$. The transition signature of $\mathcal{T}$ at $\vec{tp}$, denoted by $\tau(\mathcal{T}, \vec{tp})$, is the transition signature obtained by setting $S_\tau = T[\{t, p, c_1, c_2\}]$, $\mathcal{R}_\tau = \mathcal{R}|_{\vec{E}(S_\tau)}$, $\mathcal{E}_\tau = \mathcal{E}|_{E(S_\tau)}$, $\mathcal{F}_\tau = \mathcal{F}|_{\mathcal{P}_3(S_\tau)}$, and $U_\tau = \bigcup_{\vec{e} \in \vec{E}(S_\tau)} \mathcal{R}_\tau(\vec{e})$. We observe that the width of $\tau(\mathcal{T}, \vec{tp})$ is at most the width of $\mathcal{T}$.

Then we say that an *edge signature* of width ℓ is a tuple $\sigma = (\mathcal{R}^a_\sigma, \mathcal{R}^b_\sigma, \mathcal{E}_\sigma)$, where

- $\mathcal{R}^a_\sigma$ and $\mathcal{R}^b_\sigma$ are sets of size at most 2^ℓ and
- $\mathcal{E}_\sigma$ is a bipartite graph with bipartition $(\mathcal{R}^a_\sigma, \mathcal{R}^b_\sigma)$, with no twins over this bipartition, and with $\mathsf{cutrk}_{\mathcal{E}_\sigma}(\mathcal{R}^a_\sigma) \leqslant \ell$.

Let $\vec{ab} \in \vec{E}(T)$. The edge signature of $\mathcal{T}$ at $\vec{ab}$ is $\sigma(\mathcal{T}, \vec{ab}) = (\mathcal{R}(\vec{ab}), \mathcal{R}(\vec{ba}), \mathcal{E}(ab))$. Again, the width of $\sigma(\mathcal{T}, \vec{ab})$ is at most the width of $\mathcal{T}$.

A rank decomposition automaton of width ℓ is a tuple $\mathcal{A} = (Q, \Gamma, \iota, \delta, \varepsilon)$ that consists of

- a state set Q,
- a vertex label set Γ,
- an initial mapping ι that maps every pair of form (σ, γ), where $\sigma = (\mathcal{R}^a_\sigma, \mathcal{R}^b_\sigma, \mathcal{E}_\sigma)$ is an edge signature of width ℓ and γ is a function $\gamma\colon \mathcal{R}^a_\sigma \to \Gamma$, to a state $\iota(\sigma, \gamma) \in Q$,

- a transition mapping δ that maps every triple of form (τ, q_1, q_2), where τ is a transition signature of width ℓ and $q_1, q_2 \in Q$, to a state $\delta(\tau, q_1, q_2) \in Q$, and
- a final mapping ε that maps every triple of form (σ, q_1, q_2), where σ is an edge signature of width ℓ and $q_1, q_2 \in Q$, to a state $\varepsilon(\sigma, q_1, q_2) \in Q$.

The state set Q is allowed to be infinite. The *evaluation time* of a rank decomposition automaton is the maximum running time to compute the functions $\iota(\sigma, \gamma)$, $\delta(\tau, q_1, q_2)$, or $\varepsilon(\sigma, q_1, q_2)$ given their arguments.

Let $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ be an annotated rank decomposition of width at most ℓ that encodes a partitioned graph $(G, \mathcal{C})$, $\vec{xy} \in \vec{E}(T)$ an oriented edge of T, and $\alpha \colon V(G) \to \Gamma$ a vertex-labeling of G with Γ. Recall that $\mathsf{pred}_T(\vec{xy})$ denotes the set of predecessor of $\vec{xy}$. The *run* of $\mathcal{A}$ on the triple $(\mathcal{T}, \vec{xy}, \alpha)$ is the unique mapping $\rho \colon \mathsf{pred}_T(\vec{xy}) \to Q$ so that

- for each leaf edge $\vec{lp} \in \mathsf{pred}_T(\vec{xy}) \cap \vec{L}(T)$ it holds that $\rho(\vec{lp}) = \iota(\sigma(\mathcal{T}, \vec{lp}), \alpha|_{\mathcal{R}(\vec{lp})})$, and
- for each nonleaf edge $\vec{tp} \in \mathsf{pred}_T(\vec{xy}) \setminus \vec{L}(T)$ with children $\vec{c_1t}$, $\vec{c_2t}$, where $c_1 < c_2$, it holds that $\rho(\vec{tp}) = \delta(\tau(\mathcal{T}, \vec{tp}), \rho(\vec{c_1t}), \rho(\vec{c_2t}))$.

Then let $a, b \in V(T)$ be two adjacent nodes of T. The run of $\mathcal{A}$ on the 4-tuple $(\mathcal{T}, a, b, \alpha)$ is the unique mapping $\rho \colon \mathsf{pred}_T(\vec{ab}) \cup \mathsf{pred}_T(\vec{ba}) \cup \vartheta \to Q$ so that

- $\rho|_{\mathsf{pred}_T(\vec{ab})}$ is the run of $\mathcal{A}$ on $(\mathcal{T}, \vec{ab}, \alpha)$,
- $\rho|_{\mathsf{pred}_T(\vec{ba})}$ is the run of $\mathcal{A}$ on $(\mathcal{T}, \vec{ba}, \alpha)$, and
- $\rho(\vartheta) = \varepsilon(\sigma(\mathcal{T}, \vec{ab}), \rho(\vec{ab}), \rho(\vec{ba}))$.

The *valuation* of $\mathcal{A}$ on $(\mathcal{T}, a, b, \alpha)$ is $\rho(\vartheta)$ and on $(\mathcal{T}, \vec{xy}, \alpha)$ is $\rho(\vec{xy})$. These definitions are adapted to a rooted annotated rank decompositions with root r whose children are c_1, c_2 by setting $\rho(r\vec{c}_2) \coloneqq \rho(\vec{c_1r})$ and $\rho(r\vec{c}_1) \coloneqq \rho(\vec{c_2r})$. Additionally, the run (resp. valuation) of $\mathcal{A}$ on $(\mathcal{T}, \alpha)$ is defined as the run (resp. valuation) of $\mathcal{A}$ on $(\mathcal{T}, c_1, r, \alpha)$, where $c_1 < c_2$.

If the valuation of $\mathcal{A}$ on $(\mathcal{T}, a, b, \alpha)$ depends only on the partitioned graph $(G, \mathcal{C})$ encoded by $\mathcal{T}$ and the labeling α, then we say that $\mathcal{A}$ is *decomposition-oblivious*, and refer to this valuation as the valuation of $\mathcal{A}$ on $(G, \mathcal{C}, \alpha)$. When $\mathcal{T}$ encodes a graph G, we refer to this as the valuation of $\mathcal{A}$ on (G, α).

Next, if all runs of $\mathcal{A}$ on $(\mathcal{T}, a, b, \alpha)$ are independent on the labeling α (in particular, the value of the initial mapping ι only depends on the edge signature σ and not the function $\gamma \colon \mathcal{R}^a_\sigma \to \Gamma$), then we say that $\mathcal{A}$ is *label-oblivious*. When defining label-oblivious automata, we will for convenience drop the vertex label set Γ from the description of the automaton and consider ι to be a mapping from an edge signature $\sigma = (\mathcal{R}^a_\sigma, \mathcal{R}^b_\sigma, \mathcal{E}_\sigma)$ to a state $\iota(\sigma)$. We also define the runs on $\mathcal{A}$ on pairs $(\mathcal{T}, \vec{xy})$ and on triples $(\mathcal{T}, a, b)$ in a natural way. If $\mathcal{T}$ is rooted, we also define the run of $\mathcal{A}$ on $\mathcal{T}$ naturally.

Then we give a prefix-rebuilding data structure for maintaining runs of rank decomposition automata.

Lemma 4.5.1. *Let $\ell \in \mathbb{N}$ and $\mathcal{A} = (Q, \Gamma, \iota, \delta, \varepsilon)$ a rank decomposition automaton of width ℓ with evaluation time β. There exists an ℓ-prefix-rebuilding data structure with overhead $\mathcal{O}_\ell(1) + \mathcal{O}(\beta)$ that maintains a rooted annotated rank decomposition $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ that encodes a dynamic graph G, and a vertex-labeling $\alpha \colon V(G) \to \Gamma$ whose initial values α_{init} are given at the initialization, and additionally supports the following operations:*

- $\mathsf{Run}(\vec{xy})$*: Given an oriented edge $\vec{xy} \in \vec{E}(T)$ that is directed towards the root, in time $\mathcal{O}(1)$ returns $\rho(\vec{xy})$, where ρ is the run of $\mathcal{A}$ on $(\mathcal{T}, \vec{xy}, \alpha)$.*
- $\mathsf{Valuation}()$*: In time $\mathcal{O}(1)$ returns the valuation of $\mathcal{A}$ on $(\mathcal{T}, \alpha)$.*
- $\mathsf{SetLabel}(v, \gamma)$*: Given a vertex $v \in V(G)$ and a label $\gamma \in \Gamma$, in time $\mathcal{O}(\mathsf{height}(T) \cdot \beta)$ updates $\alpha(v) \coloneqq \gamma$.*

Proof. We maintain a representation of $\mathcal{T}$ with Lemma 4.3.2. We also maintain the vertex labeling α explicitly, and the runs of $\mathcal{A}$ on $(\mathcal{T}, \vec{c_1r}, \alpha)$ and $(\mathcal{T}, \vec{c_2r}, \alpha)$, where r is the root and $c_1 < c_2$ are the children of r. Note that this stores exactly one state $\rho(\vec{xy})$ for each oriented edge $\vec{xy}$ of T directed towards the root. We also maintain the valuation of $\mathcal{A}$ on $(\mathcal{T}, \alpha)$, which is $\varepsilon(\sigma(\mathcal{T}, \vec{c_1r}), \rho(\vec{c_1r}), \rho(\vec{c_2r}))$.

At initialization, we can compute the runs and the valuations in $\mathcal{O}(|\mathcal{T}| \cdot \beta)$ time. Then, consider a prefix-rebuilding update that turns $\mathcal{T}$ into $\mathcal{T}' = (T', V(G), \mathcal{R}', \mathcal{E}', \mathcal{F}')$, where the prefix of $\mathcal{T}$ associated with the update is T_{pref} and the prefix of $\mathcal{T}'$ is T'_{pref}. We observe that all edge signatures and transition

signatures at edges directed towards the root in $\vec{E}(T) \setminus \vec{E}(T[T_{\text{pref}} \cup \mathsf{App}_T(T_{\text{pref}})])$ stay the same in $\mathcal{T}'$. Therefore, to recompute the runs and valuations, it suffices to recompute this information only for edges directed towards the root in $\vec{E}(T'[T'_{\text{pref}} \cup \mathsf{App}_{T'}(T'_{\text{pref}})])$, which takes $\mathcal{O}(|T_{\text{pref}}| \cdot \beta)$ time.

Then consider the SetLabel operation. We observe that it can change the run on $(\mathcal{T}, \vec{xy}, \alpha)$ only if $v \in \mathcal{L}(\mathcal{T})[\vec{xy}]$. There are at most $\mathsf{height}(T)$ such edges $\vec{xy}$ directed towards the root, so we recompute the runs on them in $\mathcal{O}(\mathsf{height}(T) \cdot \beta)$ time.

We explicitly maintain all information required to answer the Run and Valuation queries, so they can be answered in $\mathcal{O}(1)$ time. □

4.5.2 CMSO_1

In this section we will show that rank decomposition automata permit us to verify the satisfaction of formulas of the variant of monadic second-order logic called CMSO_1 and evaluate the formulas of the optimization variant of CMSO_1 called $\mathsf{LinCMSO}_1$. We refer to Preliminaries (Section 2.6) for precise definitions of these logic languages.

For simplicity, we assume in this chapter that all free variables of a CMSO_1 sentence are set variables (note that free single-element variables can be expressed as free set variables). The length of a CMSO_1 sentence φ is the number of symbols appearing in it, and denoted by $|\varphi|$. We note that the length of φ is at least the number of free variables of φ, and use the convention that the free variables are indexed by consecutive integers $1, \ldots, p$.

Let φ be a CMSO_1 sentence with p free variables and G a graph. A tuple $(G, X_1, \ldots, X_p)$, where $X_i \subseteq V(G)$, *satisfies* φ, written as $(G, X_1, \ldots, X_p) \models \varphi$, if G together with the interpretations of the free variables as $X_1, \ldots, X_p$ satisfies φ. Let $\alpha \colon V(G) \to 2^{[p]}$ be a vertex-labeling of G. We define that (G, α) satisfies φ if $(G, X_1, \ldots, X_p)$, where $X_i = \{v \in V(G) \mid i \in \alpha(v)\}$ satisfies φ.

We prove the following lemma in Section 4.10 by translating automata working on a cliquewidth expressions given by Courcelle, Makowsky, and Rotics [CMR00] (see also [CE12, Section 6]) to rank decomposition automata.

Lemma 4.5.2. *There is an algorithm that given a CMSO_1 sentence φ with p free set variables and $\ell \in \mathbb{N}$, in time $\mathcal{O}_{\varphi,\ell}(1)$ constructs a decomposition-oblivious rank decomposition automaton $\mathcal{A} = (Q, \Gamma, \iota, \delta, \varepsilon)$ of width ℓ so that $\Gamma = 2^{[p]}$, the valuation of $\mathcal{A}$ on (G, α) is $\top \in Q$ if and only if $(G, \alpha) \models \varphi$, the number of states is $|Q| \leqslant \mathcal{O}_{\varphi,\ell}(1)$, and the evaluation time is $\mathcal{O}_{\varphi,\ell}(1)$.*

Recall that a $\mathsf{LinCMSO}_1$ sentence with p free variables is a pair (φ, f), where φ is a CMSO_1 sentence with $p + q$ free variables for $q \geqslant 0$, and $f \colon \mathbb{Z}^q \to \mathbb{Z}$ a linear integer function defined by $q + 1$ integers $c_0, \ldots, c_q$ so that $f(x_1, \ldots, x_q) = c_0 + c_1 x_1 + \ldots + c_q x_q$. The value of (φ, f) on a tuple $(G, X_1, \ldots, X_p)$ is the maximum value of $f(|X_{p+1}|, \ldots, |X_{p+q}|)$, where $X_{p+1}, \ldots, X_{p+q} \subseteq V(G)$ and $(G, X_1, \ldots, X_{p+q}) \models \varphi$. If no such sets $X_{p+1}, \ldots, X_{p+q}$ exist, then the value is $\perp$.

Then, Lemma 4.5.2 extends to the following lemma. The proof is also in Section 4.10.

Lemma 4.5.3. *There is an algorithm that given a $\mathsf{LinCMSO}_1$ sentence φ with p free set variables and $\ell \in \mathbb{N}$, in time $\mathcal{O}_{\varphi,\ell}(1)$ constructs a decomposition-oblivious rank decomposition automaton $\mathcal{A} = (Q, \Gamma, \iota, \delta, \varepsilon)$ of width ℓ so that $\Gamma = 2^{[p]}$, the valuation of $\mathcal{A}$ on (G, α) is equal to the value of φ on (G, α), and the evaluation time is $\mathcal{O}_{\varphi,\ell}(1)$.*

We note that the reason for having Lemma 4.5.2 and Lemma 4.5.3 as separate lemmas is that we will use the fact that the number of states in the automaton constructed in Lemma 4.5.2 is $\mathcal{O}_{\varphi,\ell}(1)$. We also note that in both Lemmas 4.5.2 and 4.5.3 the constructed automaton works only on decompositions encoding graphs, not partitioned graphs.

By putting together Lemmas 4.5.1 and 4.5.3, we obtain the following.

Lemma 4.5.4. *Let $w, \ell \in \mathbb{N}$. There exists an ℓ-prefix-rebuilding data structure with overhead $\mathcal{O}_{\ell,w}(1)$ that maintains a rooted annotated rank decomposition $\mathcal{T}$ that encodes a dynamic graph G, and additionally supports the following query:*

- $\mathsf{LinCMSO}_1(\varphi, X_1, \ldots, X_p)$*: Given a $\mathsf{LinCMSO}_1$ sentence φ of length at most w with p free variables and p vertex subsets $X_1, \ldots, X_p \subseteq V(G)$, returns the value of φ on $(G, X_1, \ldots, X_p)$. Runs in time $\mathcal{O}_\varphi(1)$ if the sets are empty, and in $\mathcal{O}_{\ell,\varphi}(\sum_{i=1}^{p} |X_i| \cdot \mathsf{height}(\mathcal{T}))$ time otherwise.*

Proof. We enumerate all $\mathsf{LinCMSO}_1$ sentences φ of length at most w, and for each of them construct an auxiliary ℓ-prefix-rebuilding structure $\mathbb{D}^\varphi$ as follows. Let p be the number of free variables in φ. We apply

Lemma 4.5.3 to obtain a rank decomposition automaton $\mathcal{A} = (Q, \Gamma, \iota, \delta, \varepsilon)$ of width ℓ so that $\Gamma = 2^{[p]}$, the valuation of $\mathcal{A}$ on (G, α) is equal to the value of φ on (G, α), and the evaluation time of $\mathcal{A}$ is $\mathcal{O}_{\varphi,\ell}(1)$. Then we initialize an ℓ-prefix-rebuilding data structure $\mathbb{D}^\varphi$ of Lemma 4.5.1 with $\mathcal{A}$. The overhead of $\mathbb{D}^\varphi$ is $\mathcal{O}_{\varphi,\ell}(1)$. We initialize the labeling α held by $\mathbb{D}^\varphi$ to be $\alpha(v) = \emptyset$ for all $v \in V(G)$.

Note that there are at most $\mathcal{O}_w(1)$ $\mathsf{LinCMSO}_1$ sentences of length at most w, so the initialization works in $\mathcal{O}_{\ell,w}(1)$ time. Then, all prefix-rebuilding updates to our data structures are relayed to all of the auxiliary data structures $\mathbb{D}^\varphi$ so that they also hold the decomposition $\mathcal{T}$ at all times, resulting in the overhead $\mathcal{O}_{\ell,w}(1)$.

The $\mathsf{LinCMSO}_1(\varphi, X_1, \ldots, X_p)$ query is implemented as follows. We maintain that between the queries, the labeling α held by $\mathbb{D}^\varphi$ is $\alpha(v) = \emptyset$ for all $v \in V(G)$. Therefore, if the given sets $X_1, \ldots, X_p$ are empty, we can simply return the value given by the query $\mathsf{Valuation}()$ of $\mathbb{D}^\varphi$. This runs in $\mathcal{O}_\varphi(1)$ time. If some of the sets $X_1, \ldots, X_p$ is nonempty, we compute $X = X_1 \cup \ldots \cup X_p$, use the $\mathsf{SetLabel}$ query of $\mathbb{D}^\varphi$ to set $\alpha(v) = \{i \mid v \in X_i\}$ for all $v \in X$, and return the value given by the query $\mathsf{Valuation}()$ of $\mathbb{D}^\varphi$. Then, we reset the labels $\alpha(v)$ of all $v \in X$ to be $\emptyset$. This takes $\mathcal{O}_{\ell,\varphi}(|X| \cdot \mathsf{height}(\mathcal{T})) = \mathcal{O}_{\ell,\varphi}(\sum_{i=1}^{p} |X_i| \cdot \mathsf{height}(\mathcal{T}))$ time. □

4.5.3 Edge update sentences

Let G be a graph. An *edge update sentence* on G is a tuple $\overline{e} = (\varphi, X, X_1, \ldots, X_p)$, where φ is a CMSO_1 sentence with $p+1$ free set variables, $X \subseteq V(G)$, and $X_i \subseteq X$ for all $i \in [p]$. The graph resulting from applying $\overline{e}$ to G is the graph G' with $V(G') = V(G)$, and with $uv \in E(G')$ for $u \neq v$ if and only if either

- $|\{u, v\} \cap X| \leqslant 1$ and $uv \in E(G)$, or
- $u, v \in X$ and $(G, \{u, v\}, X_1, \ldots, X_p) \models \varphi$.

In other words, the edges inside $G[X]$ are defined by $\overline{e}$, while other edges remain unchanged. We define that *size* of $\overline{e}$ as $|\overline{e}| = |X|$ and that the *length* of $\overline{e}$ is the length of φ, i.e., $|\varphi|$.

Next we give our data structure to turn edge update sentences to edge update descriptions. We note that while it is not immediately obvious that a rank decomposition of G of width ℓ would also be a rank decomposition of G' whose width is bounded by $\mathcal{O}_{\ell,|\varphi|}(1)$, our proof implies this because the resulting edge update description has width $\mathcal{O}_{\ell,|\varphi|}(1)$.

Lemma 4.5.5. *Let $d, \ell \in \mathbb{N}$. There exists an ℓ-prefix-rebuilding data structure with overhead $\mathcal{O}_{\ell,d}(1)$ that maintains a rooted annotated rank decomposition $\mathcal{T}$ that encodes a dynamic graph G and additionally supports the following query:*

- $\mathsf{EdgeUpdate}(\overline{e})$*: Given an edge update sentence $\overline{e}$ on G of length at most d, returns an edge update description of width $\mathcal{O}_{\ell,d}(1)$ that describes the graph G' that results from applying $\overline{e}$ to G. Runs in time $\mathcal{O}_{\ell,d}(\mathsf{height}(\mathcal{T}) \cdot |\overline{e}|)$.*

Proof. In the initialization we construct a set of auxiliary automata and prefix-rebuilding data structures as follows. We enumerate all CMSO_1 sentences of length at most d and at least one free set variable, i.e., all CMSO_1 sentences that could be in the edge update sentence given in $\mathsf{EdgeUpdate}(\overline{e})$. Let φ be such sentence with $p+1$ free variables $Y, X_1, \ldots, X_p$, where Y is the free variable that is supposed to hold the endpoints of the potential edge. We construct a CMSO_1 sentence φ' with $p+2$ free variables $Y, X, X_1, \ldots, X_p$, so that $(G, Y, X, X_1, \ldots, X_p) \models \varphi'$ if and only if either

- $Y \subseteq X$, $|Y| = 2$, and $(G, Y, X_1, \ldots, X_p)$ satisfies φ, or
- $Y \not\subseteq X$ and $Y = \{u, v\}$ with $uv \in E(G)$.

In particular, $(G, Y, X, X_1, \ldots, X_p) \models \varphi'$ if and only if $Y = \{u, v\}$ corresponds to an edge in the graph G' resulting from applying the edge update sentence $(\varphi, X, X_1, \ldots, X_p)$. Such φ' with $|\varphi'| \leqslant \mathcal{O}(|\varphi|)$ can be constructed in time $\mathcal{O}(|\varphi|)$.

Then we use Lemma 4.5.2 to construct a rank decomposition automaton $\mathcal{A}_{\varphi'} = (Q, \Gamma, \iota, \delta, \epsilon)$ of width ℓ so that $\Gamma = 2^{[p+2]}$, the valuation of $\mathcal{A}_{\varphi'}$ on (G, α) is $\top$ if and only if $(G, \alpha) \models \varphi'$, $|Q| \leqslant \mathcal{O}_{\varphi,\ell}(1)$, and the evaluation time is $\mathcal{O}_{\varphi,\ell}(1)$. We say that a labeling $\alpha \colon V(G) \to 2^{[p+2]}$ corresponds to an edge update sentence $(\varphi, X, X_1, \ldots, X_p)$ if $2 \in \alpha(v)$ if and only if $v \in X$, and $2 + i \in \alpha(v)$ if and only if $v \in X_i$.

Let $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ be an annotated rank decomposition that encodes G, and let $\alpha \colon V(G) \to 2^{[p+2]}$ be a labeling of G with $1 \notin \alpha(v)$ for all $v \in V(G)$. Let us also denote by α_v the labeling so that $\alpha_v(u) \setminus \{1\} = \alpha(u)$ for all $u \in V(G)$ and $1 \in \alpha_v(u)$ if and only if $u = v$. With an oriented edge $\vec{xy} \in \vec{E}(T)$ we associate a 4-tuple $(q_{\vec{xy}}, f_{\vec{xy}}, g_{\vec{xy}}, h_{\vec{xy}})$ so that

- $q_{\vec{xy}}$ is the valuation of $\mathcal{A}_{\varphi'}$ on $(\mathcal{T}, \vec{xy}, \alpha)$,
- $f_{\vec{xy}} \colon Q \to V(G) \cup \{\bot\}$ is the function so that for every $q \in Q$ the value $f_{\vec{xy}}(q)$ is the vertex $v \in \mathcal{L}(\mathcal{T})[\vec{xy}]$ with the smallest index so that the valuation of $\mathcal{A}_{\varphi'}$ on $(\mathcal{T}, \vec{xy}, \alpha_v)$ is q, or $\bot$ if no such vertex v exists,
- $g_{\vec{xy}} \colon \mathcal{R}(\vec{xy}) \to V(G)$ is the function so that for every $r \in \mathcal{R}(\vec{xy})$ the value $g_{\vec{xy}}(r)$ is the smallest-index vertex $v \in \mathcal{L}(\mathcal{T})[\vec{xy}]$ so that $N_G(v) \cap \mathcal{R}(\vec{yx}) = N_G(r) \cap \mathcal{R}(\vec{yx})$, and
- $h_{\vec{xy}} \colon \mathcal{R}(\vec{xy}) \to Q$ is the function so that $h_{\vec{xy}}(r)$ is the valuation of $\mathcal{A}_{\varphi'}$ on $(\mathcal{T}, \vec{xy}, \alpha_{g_{\vec{xy}}(r)})$.

We construct a rank decomposition automaton $\mathcal{A}'_{\varphi'}$ of width ℓ so that the valuation of $\mathcal{A}'_{\varphi'}$ on $(\mathcal{T}, \vec{xy}, \alpha)$ is the 4-tuple $(q_{\vec{xy}}, f_{\vec{xy}}, g_{\vec{xy}}, h_{\vec{xy}})$. Such automaton with evaluation time $\mathcal{O}_{\ell,\varphi}(1)$ can be constructed as follows: First, the state $q_{\vec{xy}}$ can be maintained simply by simulating $\mathcal{A}_{\varphi'}$. Then, we observe that $f_{\vec{xy}}$ can be computed from $f_{c_1\vec{x}}$, $f_{c_2\vec{x}}$, $q_{c_1\vec{x}}$, and $q_{c_2\vec{x}}$, where $c_1\vec{x}$ and $c_2\vec{x}$ are the child edges of $\vec{xy}$, in particular

$$f_{\vec{xy}}(q) = \min \left\{ \min_{q_1 \in Q \mid \delta(\tau(\mathcal{T},\vec{xy}),q_1,q_{c_2\vec{x}})=q} f_{c_1\vec{x}}(q_1), \min_{q_2 \in Q \mid \delta(\tau(\mathcal{T},\vec{xy}),q_{c_1\vec{x}},q_2)=q} f_{c_2\vec{x}}(q_2) \right\},$$

where $\bot$ is regarded as larger than any vertex. For $g_{\vec{xy}}$ and $h_{\vec{xy}}$, we first observe that if $g_{\vec{xy}}(r) = v$, then there exists either $r' \in \mathcal{R}(c_1\vec{x})$ with $g_{c_1\vec{x}}(r') = v$ or $r' \in \mathcal{R}(c_2\vec{x})$ with $g_{c_2\vec{x}}(r') = v$. With this observation, $g_{\vec{xy}}$ can be computed from $g_{c_1\vec{x}}$ and $g_{c_2\vec{x}}$ by using $\mathcal{F}(c_1xy)$, $\mathcal{F}(c_2xy)$, and $\mathcal{E}(xy)$, which are stored in $\tau(\mathcal{T}, \vec{xy})$. Then, if $g_{\vec{xy}}(r) = v$ so that there exists $r' \in \mathcal{R}(c_1\vec{x})$ with $g_{c_1\vec{x}}(r') = v$, we have $h_{\vec{xy}}(r) = \delta(\tau(\mathcal{T}, \vec{xy}), h_{c_1\vec{x}}(r'), q_{c_2\vec{x}})$; and the other case is similar. This completes the construction of $\mathcal{A}'_{\varphi'}$.

Then, we construct an ℓ-prefix-rebuilding data structure $\mathbb{D}^{\varphi}$ by invoking Lemma 4.5.1 with $\mathcal{A}'_{\varphi'}$. All prefix-rebuilding updates are relayed to $\mathbb{D}^{\varphi}$ so that it always holds the same annotated rank decomposition as the main prefix-rebuilding data structure of the lemma. The vertex labeling α_{φ} that $\mathbb{D}^{\varphi}$ holds will always be $\alpha_{\varphi}(v) = \emptyset$ for all $v \in V(G)$, except when we are processing the $\mathsf{EdgeUpdate}(\overline{e})$ query. Note that because $|\varphi| \leqslant d$, the number of such prefix-rebuilding data structures $\mathbb{D}^{\varphi}$ we maintain is $\mathcal{O}_d(1)$.

This completes the description of the initialization and the handling of prefix-rebuilding updates. It remains to describe how $\mathsf{EdgeUpdate}(\overline{e})$ is implemented.

Let $\overline{e} = (\varphi, X, X_1, \ldots, X_p)$. We first use the $\mathsf{SetLabel}(v, \gamma)$ query of $\mathbb{D}^{\varphi}$ for all $v \in X$ to set the labeling α to correspond to $\overline{e}$. This takes $\mathcal{O}_{\ell,\varphi}(\mathsf{height}(T) \cdot |\overline{e}|)$ time. Then, let T_{pref} be the unique smallest prefix of T that contains all leaves $l \in L(T)$ with $\mathcal{R}(\vec{lp}) \subseteq X$. We have that $|T_{\mathrm{pref}}| \leqslant \mathsf{height}(T) \cdot |\overline{e}|$. The prefix T_{pref} will be the prefix of the edge update description we output. With the help of $\mathbb{D}^{\varphi}$ we compute the triples $(q_{\vec{xy}}, f_{\vec{xy}}, g_{\vec{xy}})$ for all oriented edges $\vec{xy} \in \vec{E}(T[T_{\mathrm{pref}} \cup \mathsf{App}_T(T_{\mathrm{pref}})])$ in $\mathcal{O}_{\ell,\varphi}(|T_{\mathrm{pref}}|) = \mathcal{O}_{\ell,\varphi}(\mathsf{height}(T) \cdot |\overline{e}|)$ time. In particular, such triples are directly given by $\mathbb{D}^{\varphi}$ for all oriented edges directed towards the root, and for oriented edges directed towards the leaves we can compute them with $\mathcal{A}'_{\varphi'}$ in a top-down manner.

Then, the purpose of the definition of $f_{\vec{xy}}$ is to make the following hold.

Claim 4.5.6. *Let $\vec{xy} \in \vec{E}(T)$ and let G' be the graph resulting from applying $\overline{e}$ to G. The set $R_{\vec{xy}} = \bigcup_{q \in Q} \{f_{\vec{xy}}(q)\} \setminus \{\bot\}$ is a representative of $\mathcal{L}(\mathcal{T})[\vec{xy}]$ in G', and given $f_{\vec{xy}}$ and $f_{\vec{yx}}$ the graph $G'[R_{\vec{xy}}, R_{\vec{yx}}]$ can be determined in $\mathcal{O}_{\varphi,\ell}(1)$ time.*

Proof of the claim. Let $v \in \mathcal{L}(\mathcal{T})[\vec{xy}]$ and $u \in \mathcal{L}(\mathcal{T})[\vec{yx}]$. We observe that $uv \in E(G')$ if and only if the valuation of $\mathcal{A}_{\varphi'}$ on $(\mathcal{T}, \vec{xy}, \alpha_v)$ is q_1, the valuation of $\mathcal{A}_{\varphi'}$ on $(\mathcal{T}, \vec{yx}, \alpha_v)$ is q_2, and $\varepsilon(\sigma(\mathcal{T}, \vec{xy}), q_1, q_2) = \top$. Therefore if $r \in R_{\vec{xy}}$ and the valuations of $\mathcal{A}_{\varphi'}$ on $(\mathcal{T}, \vec{xy}, \alpha_v)$ and $(\mathcal{T}, \vec{xy}, \alpha_r)$ are the same, then $N_{G'}(v) \cap \mathcal{L}(\mathcal{T})[\vec{yx}] = N_{G'}(r) \cap \mathcal{L}(\mathcal{T})[\vec{yx}]$. Because for every $v \in \mathcal{L}(\mathcal{T})[\vec{xy}]$ there exists such $r \in R_{\vec{xy}}$, we have that $R_{\vec{xy}}$ is a representative of $\mathcal{L}(\mathcal{T})[\vec{xy}]$ in G'. Then the graph $G'[R_{\vec{xy}}, R_{\vec{yx}}]$ can be determined by verifying whether $\varepsilon(\sigma(\mathcal{T}, \vec{xy}), q_1, q_2) = \top$ for all $q_1, q_2 \in Q$. ◁

In particular, by Claim 4.5.6 in the edge update description we can set $\mathcal{R}^\star(\vec{xy}) = R_{\vec{xy}}$ for all $\vec{xy} \in \vec{E}(T[T_{\mathrm{pref}}])$. It also gives a way to compute the graphs $\mathcal{E}^\star(xy) = G'[R_{\vec{xy}}, R_{\vec{yx}}]$ for $xy \in E(T[T_{\mathrm{pref}}])$. For $xyz \in \mathcal{P}_3(T[T_{\mathrm{pref}}])$, the graphs $\mathcal{E}^\star(xyz) = G'[R_{\vec{xy}}, R_{\vec{zy}}]$ can be computed as follows. Let $v \in R_{\vec{xy}}$ and $u \in R_{\vec{zy}}$, and let w be the neighbor of y that is not x or z. From $f_{\vec{xy}}$ we know the valuation of $\mathcal{A}_{\varphi'}$ on $(\mathcal{T}, \vec{xy}, \alpha_v)$, from $f_{\vec{zy}}$ we know the valuation of $\mathcal{A}_{\varphi'}$ on $(\mathcal{T}, \vec{zy}, \alpha_u)$, and from $q_{\vec{wy}}$ we know the valuation of $\mathcal{A}_{\varphi'}$ on $(\mathcal{T}, \vec{wy}, \alpha)$. By combining these with $\mathcal{O}(1)$ transitions of $\mathcal{A}_{\varphi'}$ we find whether $uv \in E(G')$. This takes $\mathcal{O}_{\varphi,\ell}(1)$ time for each $xyz \in \mathcal{P}_3(T[T_{\mathrm{pref}}])$, i.e., $\mathcal{O}_{\varphi,\ell}(\mathsf{height}(T) \cdot |\overline{e}|)$ time in total.

It remains to compute for $xyz \in \mathcal{P}_3(T)$ with $x \in \mathsf{App}_T(T_{\mathrm{pref}})$ and $y, z \in T_{\mathrm{pref}}$ the graphs $\mathcal{E}^\star(xyz) = G'[\mathcal{R}(\vec{xy}), \mathcal{R}^\star(\vec{zy})]$. For this, we recall that $g_{\vec{xy}}$ stores for each $r \in \mathcal{R}(\vec{xy})$ the smallest-index vertex $v \in \mathcal{L}(\mathcal{T})[\vec{xy}]$ so that $N_G(v) \cap \mathcal{L}(\mathcal{T})[\vec{yx}] = N_G(r) \cap \mathcal{L}(\mathcal{T})[\vec{yx}]$, and $h_{\vec{xy}}$ stores for each $r \in \mathcal{R}(\vec{xy})$ the valuation

of $\mathcal{A}_{\varphi'}$ on $(\mathcal{T}, \vec{xy}, \alpha_{g_{\vec{xy}}(r)})$. Now, because $\mathcal{L}(\mathcal{T})[\vec{xy}]$ is disjoint from X, we have that $N_{G'}(v) \cap \mathcal{L}(\mathcal{T})[\vec{yx}] = N_{G'}(r) \cap \mathcal{L}(\mathcal{T})[\vec{yx}]$. Therefore, it suffices to find the adjacencies of such vertices v to $\mathcal{R}^\star(\vec{zy})$ in G'. Because we know the valuation of $\mathcal{A}_{\varphi'}$ on $(\mathcal{T}, \vec{xy}, \alpha_v)$, we can do this in a similar manner as in the previous paragraph.

This completes the description of the implementation of $\mathsf{EdgeUpdate}(\overline{e})$. All of the steps took $\mathcal{O}_{\varphi,\ell}(\mathsf{height}(T) \cdot |\overline{e}|) = \mathcal{O}_{d,\ell}(\mathsf{height}(T) \cdot |\overline{e}|)$ time. □

4.6 Dynamic rankwidth

In this section we put together the material from the previous sections to give the final proof of our dynamic data structure for rankwidth.

Let us first bound how much a prefix-rebuilding update resulting from an edge update description can increase the potential of a rank decomposition.

Lemma 4.6.1. *Let $\mathcal{T}$ be a rooted annotated rank decomposition that encodes a graph G, $\overline{u}$ an edge update description that describes a graph G', $\mathcal{T}'$ a rooted annotated rank decomposition that results from applying to $\mathcal{T}$ a prefix-rebuilding update that corresponds to $\overline{u}$, and ℓ an integer so that the widths of both $\mathcal{T}$ and $\mathcal{T}'$ are at most ℓ. Then it holds that*

$$\Phi_{\ell,G'}(\mathcal{T}') \leqslant \Phi_{\ell,G}(\mathcal{T}) + \mathcal{O}_\ell(|\overline{u}| \cdot \mathsf{height}(\mathcal{T}))$$

Proof. Recall that both graphs G and G' share the same set of vertices and for both decompositions $\mathcal{T}$ and $\mathcal{T}'$ the tree T and the sets $\mathcal{R}(\vec{lp})$ on leaf edges $\vec{lp}$ are the same. Let T_{pref} be the prefix of T given in the edge update description. We have that $|T_{\text{pref}}| = |\overline{u}|$ and the width of an edge can change only if it is in $T[T_{\text{pref}}]$. Then, the conclusion follows directly from the definition of Φ. □

Then we state a lemma about computing optimum-width rank decompositions by dynamic programming on annotated rank decompositions, which will be proved in Section 4.9.1.

Lemma 4.6.2. *Let $k, \ell \geqslant 0$ be integers. There exists an algorithm that, given as input an annotated rank decomposition $\mathcal{T}$ of width ℓ that encodes a partitioned graph $(G, \mathcal{C})$, in time $\mathcal{O}_\ell(|\mathcal{T}| \log |\mathcal{T}|)$ either:*

- *correctly determines that $(G, \mathcal{C})$ has rankwidth larger than k; or*
- *outputs an annotated rank decomposition that encodes $(G, \mathcal{C})$ and has width at most k.*

Next we give the main lemma giving the basic version of our data structure. In the statement it is important that the decomposition $\mathcal{T}$ is maintained by prefix-rebuilding updates, as this implies that any feature of $\mathcal{T}$ that can be maintained by a prefix-rebuilding data structure can be plugged in to the data structure.

Lemma 4.6.3. *Let $k, d, n \in \mathbb{N}$. There is a data structure that using prefix-rebuilding updates maintains a rooted annotated rank decomposition $\mathcal{T}$ that encodes a dynamic n-vertex graph G and has width at most $4k$, under the promise that G has rankwidth at most k at all times, under the following operations:*

- $\mathsf{Init}(\tilde{\mathcal{T}})$*: Given a rooted annotated rank decomposition $\tilde{\mathcal{T}}$ that encodes a graph G and has width at most $4k$, initializes the data structure to hold $\mathcal{T} := \tilde{\mathcal{T}}$. Runs in amortized $\mathcal{O}_{k,d}(n \log^2 n)$ time.*
- $\mathsf{Update}(\overline{e})$*: Given an edge update sentence $\overline{e}$ of length at most d, either returns that the graph resulting from applying $\overline{e}$ to G would have rankwidth more than k, or applies $\overline{e}$ to update G. Runs in amortized $|\overline{e}| \cdot 2^{\mathcal{O}_{k,d}(\sqrt{\log n \log\log n})}$ time.*

Moreover, it is guaranteed that after each operation, the height of $\mathcal{T}$ is at most $2^{\mathcal{O}_{k,d}(\sqrt{\log n \log\log n})}$, even though during the implementations of the operations the height of $\mathcal{T}$ can be greater.

Proof. We choose ℓ to be the smallest positive integer so that $\ell \geqslant 4k+1$, $\ell \geqslant d$, and ℓ is at least the largest width of an edge update description that is returned by the $\mathsf{EdgeUpdate}(\overline{e})$ query of the $4k$-prefix-rebuilding data structure of Lemma 4.5.5 with the parameter d. Note that $\ell \leqslant \mathcal{O}_{k,d}(1)$.

Then, the $\mathsf{Init}(\tilde{\mathcal{T}})$ query is implemented as follows. Given the decomposition $\tilde{\mathcal{T}}$ that encodes G, we first use Lemma 4.6.2 to compute a rank decomposition $\tilde{\mathcal{T}}'$ of G of width at most k, then use Lemma 4.2.2 to turn $\tilde{\mathcal{T}}'$ into a rank decomposition $\tilde{\mathcal{T}}''$ of height $\mathcal{O}(\log n)$ and width at most $2k$, and then use Lemma 4.3.8 with $\tilde{\mathcal{T}}$ and $\tilde{\mathcal{T}}''$ to compute an annotated rank decomposition $\mathcal{T}$ that encodes G and corresponds to $\tilde{\mathcal{T}}''$.

This runs in $\mathcal{O}_{k,d}(n \log n) = \mathcal{O}_\ell(n \log n)$ time in total, and because the resulting decomposition $\mathcal{T}$ has width at most $2k$ and height at most $\mathcal{O}(\log n)$, its ℓ-potential is $\Phi_{\ell,G}(\mathcal{T}) \leqslant \mathcal{O}_\ell(n \log n)$. The first prefix-rebuilding update is to update $\tilde{\mathcal{T}}$ into $\mathcal{T}$. Note that we can set its description to fully contain $\mathcal{T}$ in $\mathcal{O}_\ell(n)$ time.

We then initialize the ℓ-prefix-rebuilding data structures $\mathbb{D}^{\mathsf{improve}}$ of Lemma 4.4.13, $\mathbb{D}^{\mathsf{translate}}$ of Lemma 4.3.14, and $\mathbb{D}^{\mathsf{prdsutil}}$ of Lemma 4.3.4 with $\mathcal{T}$, and the $4k$-prefix-rebuilding data structure $\mathbb{D}^{\mathsf{upd}}$ of Lemma 4.5.5 with $\mathcal{T}$. Usually, these four data structures will hold the same current annotated rank decomposition $\mathcal{T}$ of width at most $4k$, but during the Update query the data structure $\mathbb{D}^{\mathsf{improve}}$ may hold an annotated rank decomposition $\mathcal{T}'$ of width up to ℓ. The initialization of these data structures takes $\mathcal{O}_\ell(n)$ time.

Let $h = 2^{\mathcal{O}_\ell(\sqrt{\log n \log\log n})}$ be so that the maximum height of $\mathcal{T}$ after applying the ImproveHeight() operation of $\mathbb{D}^{\mathsf{improve}}$ is at most h. We will maintain the invariant that between the Update queries, the height of $\mathcal{T}$ is at most h. During the Update query the height may grow unboundedly.

Then, the Update($\overline{e}$) query is implemented as follows. Let G' be the graph resulting from applying $\overline{e}$ to G. We first use the data structure $\mathbb{D}^{\mathsf{upd}}$ to compute an edge update description $\overline{u}$ corresponding to $\overline{e}$. This runs in time $\mathcal{O}_{k,d}(\mathsf{height}(\mathcal{T}) \cdot |\overline{e}|) \leqslant \mathcal{O}_\ell(h \cdot |\overline{e}|)$, which is also an upper bound for $|\overline{u}|$. By the choice of ℓ, the width of $\overline{u}$ is at most ℓ, which is also an upper bound for the width of the decomposition resulting from applying $\overline{u}$ to $\mathcal{T}$. Then we use the data structure $\mathbb{D}^{\mathsf{translate}}$ to translate $\overline{u}$ into a description $\overline{u}_1$ of a prefix-rebuilding update. This runs in $\mathcal{O}_\ell(|\overline{u}|) = \mathcal{O}_\ell(h \cdot |\overline{e}|)$ time, which is also an upper bound for $|\overline{u}_1|$. Then, we apply $\overline{u}_1$ to $\mathbb{D}^{\mathsf{improve}}$ (but not the other prefix-rebuilding data structures). Let $\mathcal{T}' = (T, V(G), \mathcal{R}', \mathcal{E}', \mathcal{F}')$ be the decomposition resulting from applying $\overline{u}_1$ to $\mathcal{T}$. We have that $\mathcal{T}'$ encodes G' and by Lemma 4.6.1 the ℓ-potential of $\mathcal{T}'$ is at most

$$\Phi_{\ell,G'}(\mathcal{T}') \leqslant \Phi_{\ell,G}(\mathcal{T}) + \mathcal{O}_\ell(h \cdot |\overline{e}|).$$

Let T_{pref} be the prefix of T' associated with $\overline{u}_1$. We note that all nodes of $\mathcal{T}'$ of width larger than $4k$ are in T_{pref}, and apply the Refine(T_{pref}) operation of $\mathbb{D}^{\mathsf{improve}}$. If it returns that the rankwidth of G' is greater than k, we use the Reverse operation of $\mathbb{D}^{\mathsf{prdsutil}}$ to compute a description of a prefix-rebuilding operation that turns $\mathcal{T}'$ back to $\mathcal{T}$, apply it to $\mathbb{D}^{\mathsf{improve}}$, and then return. In this case the time complexity is $\mathcal{O}_\ell(h \cdot |\overline{e}|)$. The other case is that the Refine(T_{pref}) operation returns a description $\overline{u}_2$ of a prefix-rebuilding update that turns $\mathcal{T}'$ into a decomposition $\mathcal{T}''$ that encodes G', has width at most $4k$, and satisfies

$$\begin{aligned}
\Phi_{\ell,G'}(\mathcal{T}'') &\leqslant \Phi_{\ell,G'}(\mathcal{T}') - \mathsf{height}_{T'}(T_{\mathrm{pref}}) + \log n \cdot \mathcal{O}_\ell(|T_{\mathrm{pref}}| + \mathsf{height}_{T'}(\mathsf{App}_{T'}(T_{\mathrm{pref}}))) \\
&\leqslant \Phi_{\ell,G'}(\mathcal{T}') + \log n \cdot \mathcal{O}_\ell(h \cdot |\overline{e}| + h^2 \cdot |\overline{e}|) \\
&\leqslant \Phi_{\ell,G}(\mathcal{T}) + \mathcal{O}_\ell(h^2 \cdot |\overline{e}| \cdot \log n).
\end{aligned}$$

The running time of the operation and therefore also $|\overline{u}_2|$ is

$$\begin{aligned}
&\log n \cdot \mathcal{O}_\ell(\Phi_{\ell,G'}(\mathcal{T}') - \Phi_{\ell,G'}(\mathcal{T}'') + \log n \cdot (|T_{\mathrm{pref}}| + \mathsf{height}_{T'}(\mathsf{App}_{T'}(T_{\mathrm{pref}})))) \\
\leqslant &\log n \cdot \mathcal{O}_\ell(\Phi_{\ell,G'}(\mathcal{T}') - \Phi_{\ell,G'}(\mathcal{T}'')) + \mathcal{O}_\ell(h^2 \cdot |\overline{e}| \cdot \log^2 n) \\
\leqslant &\log n \cdot \mathcal{O}_\ell(\Phi_{\ell,G'}(\mathcal{T}) - \Phi_{\ell,G'}(\mathcal{T}'')) + \mathcal{O}_\ell(h^2 \cdot |\overline{e}| \cdot \log^2 n).
\end{aligned}$$

Then we use $\mathbb{D}^{\mathsf{compose}}$ to compute from $\overline{u}_1$ and $\overline{u}_2$ a description $\overline{u}_\circ$ of a prefix-rebuilding update that turns $\mathcal{T}$ into $\mathcal{T}''$. We apply $\overline{u}_\circ$ to $\mathbb{D}^{\mathsf{translate}}$, $\mathbb{D}^{\mathsf{compose}}$, and $\mathbb{D}^{\mathsf{upd}}$, and then apply $\overline{u}_2$ to $\mathbb{D}^{\mathsf{improve}}$. Now, all of these data structures hold the same decomposition $\mathcal{T}''$. This takes time $\mathcal{O}_\ell(|\overline{u}_1| + |\overline{u}_2|) \leqslant \log n \cdot \mathcal{O}_\ell(\Phi_{\ell,G'}(\mathcal{T}) - \Phi_{\ell,G'}(\mathcal{T}'')) + \mathcal{O}_\ell(h^2 \cdot |\overline{e}| \cdot \log^2 n)$.

Then, we call the ImproveHeight() operation of $\mathbb{D}^{\mathsf{improve}}$. This updates $\mathcal{T}''$ through a series of prefix-rebuilding updates into a decomposition $\mathcal{T}'''$ that has height at most h and width at most $4k$, and returns the corresponding sequence of descriptions of prefix-rebuilding updates. We also apply the same sequence of prefix-rebuilding updates to $\mathbb{D}^{\mathsf{translate}}$, $\mathbb{D}^{\mathsf{compose}}$, and $\mathbb{D}^{\mathsf{upd}}$, noting that also the intermediate decompositions in this sequence have width at most $4k$. It holds that $\Phi_{\ell,G'}(\mathcal{T}''') \leqslant \Phi_{\ell,G'}(\mathcal{T}'')$ and the running time of this is

$$\begin{aligned}
&\log n \cdot \mathcal{O}_\ell(\Phi_{\ell,G'}(\mathcal{T}'') - \Phi_{\ell,G'}(\mathcal{T}''')) \\
\leqslant &\log n \cdot \mathcal{O}_\ell(\Phi_{\ell,G'}(\mathcal{T}) - \Phi_{\ell,G'}(\mathcal{T}''')) + \mathcal{O}_\ell(h^2 \cdot |\overline{e}| \cdot \log^2 n).
\end{aligned} \tag{4.7}$$

Finally, $\mathcal{T}'''$ is the decomposition that our data structure will hold after the Update operation. Note that we updated $\mathcal{T}$ into $\mathcal{T}'''$ by prefix-rebuilding operations so that all intermediate decompositions had width at most $4k$. As $\Phi_{\ell,G'}(\mathcal{T}''') \leqslant \Phi_{\ell,G'}(\mathcal{T}'')$, the total time complexity of the operation is bounded

by $\log n \cdot \mathcal{O}_\ell(\Phi_{\ell,G'}(\mathcal{T}) - \Phi_{\ell,G'}(\mathcal{T}'''))) + \mathcal{O}_\ell(h^2 \cdot |\overline{e}| \cdot \log^2 n)$. We also have that $\Phi_{\ell,G'}(\mathcal{T}''') \leqslant \Phi_{\ell,G'}(\mathcal{T}'') \leqslant \Phi_{\ell,G}(\mathcal{T}) + \mathcal{O}_\ell(h^2 \cdot |\overline{e}| \cdot \log n)$.

Then we analyze the amortized time complexity. Let us consider the sequence of t first Update operations applied to the data structure, and let us denote by $\overline{e}_1, \ldots, \overline{e}_t$ the edge update sentences given in them and by $\mathcal{T}_1, \ldots, \mathcal{T}_t$ the decompositions after each of the updates, and by $\mathcal{T}_0$ the initial decomposition. By Eq. (4.7), the total time used in the first t Update operations is at most

$$\sum_{i=1}^{t} \left(\mathcal{O}_\ell(h^2 \cdot |\overline{e}_i| \cdot \log^2 n) + \log n \cdot \mathcal{O}_\ell(\Phi_\ell(\mathcal{T}_{i-1}) - \Phi_\ell(\mathcal{T}_i))\right).$$

Now, because $\Phi_\ell(\mathcal{T}_i)$ is always nonnegative, $\Phi_\ell(\mathcal{T}_0) \leqslant \mathcal{O}_\ell(n \log n)$, and $\Phi_\ell(\mathcal{T}_i) \leqslant \Phi_\ell(\mathcal{T}_{i-1}) + \mathcal{O}_\ell(h^2 \cdot |\overline{e}_i| \cdot \log n)$, we have that

$$\sum_{i=1}^{t} \mathcal{O}_\ell(\Phi_\ell(\mathcal{T}_{i-1}) - \Phi_\ell(\mathcal{T}_i)) \leqslant \mathcal{O}_\ell(n \log n) + \sum_{i=1}^{t} \mathcal{O}_\ell(h^2 \cdot |\overline{e}_i| \cdot \log n).$$

This implies that the total running time of the first t operations is bounded by

$$\mathcal{O}_\ell(n \log^2 n) + \sum_{i=1}^{t} \mathcal{O}_\ell(h^2 \cdot |\overline{e}_i| \cdot \log^2 n).$$

We conclude the claimed amortized running time by charging the $\mathcal{O}_\ell(n \log^2 n)$ term from the Init operation and for each $i \in [t]$ the $\mathcal{O}_\ell(h^2 \cdot |\overline{e}_i| \cdot \log^2 n)$ term from the ith Update operation. Note that $\mathcal{O}_\ell(h^2 \cdot |\overline{e}_i| \cdot \log^2 n) \leqslant |\overline{e}_i| \cdot 2^{\mathcal{O}_{k,d}(\sqrt{\log n \log\log n})}$. □

Then we add a couple of more features to the data structure of Lemma 4.6.3.

Lemma 4.6.4. *Let $k, d, n \in \mathbb{N}$. The data structure of Lemma 4.6.3 can furthermore support the following operations:*

- InitEmpty(): *Initializes the data structure to hold the n-vertex edgeless graph G. Runs in amortized $\mathcal{O}_{k,d}(n \log^2 n)$ time.*
- $\mathsf{LinCMSO}_1(\varphi, X_1, \ldots, X_p)$: *Given a $\mathsf{LinCMSO}_1$ sentence φ of length at most d with p free set variables and p vertex subsets $X_1, \ldots, X_p \subseteq V(G)$, returns the value of φ on $(G, X_1, \ldots, X_p)$. Runs in time $\mathcal{O}_d(1)$ if the sets $X_1, \ldots, X_p$ are empty, and in time $\sum_{i=1}^{p} |X_i| \cdot 2^{\mathcal{O}_{k,d}(\sqrt{\log n} \log\log n)}$ otherwise.*

Proof. First, the InitEmpty() operation can implemented by the Init($\tilde{\mathcal{T}}$) operation of the data structure of Lemma 4.6.3, as it is straightforward to construct an annotated rank decomposition of an n-vertex edgeless graph in $\mathcal{O}(n)$ time. Then, to support the $\mathsf{LinCMSO}_1(\varphi, X_1, \ldots, X_p)$ queries, we maintain the $4k$-prefix-rebuilding data structure of Lemma 4.5.4 for $w = d$. □

It is easy to see that Theorem 1.3.5 is a special case of Lemma 4.6.4: The operations to insert and delete edges can be simulated by edge update sentences of constant length and size.

4.7 Almost-linear time algorithm for rankwidth

In this section we prove Theorem 1.3.6 by using Lemma 4.6.3. We prove in fact a bit more general statement, showing that if the $2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$ factor in Lemma 4.6.3 could be improved to $\mathcal{O}_k(\log^{\mathcal{O}(1)} n)$, then the $2^{\sqrt{\log n} \log\log n}$ factor in Theorem 1.3.6 could be improved to $\log^{\mathcal{O}(1)} n$.

4.7.1 The twin flipping problem

When G is a graph and F is a set of unordered pairs of vertices of G, we denote by $G \triangle F$ the graph obtained from G by "flipping" adjacencies between every pair in F. In other words, $V(G \triangle F) = V(G)$ and $E(G \triangle F) = E(G) \triangle F$. Recall that a vertex v is a twin of a vertex u if $N(v) = N(u)$. Our interface between Lemma 4.6.3 and Theorem 1.3.6 will be the following problem.

Problem 4.7.1 (Twin Flipping)**.** *Given an annotated rank decomposition of width at most k that encodes an n-vertex bipartite graph G with bipartition (A, B), two disjoint vertex sets $X, Y \subseteq A$ so that every vertex in X has a twin in Y, and a set $F \subseteq X \times B$ of size $|F| \leqslant \mathcal{O}_k(n)$, either determine that the rankwidth of $G \triangle F$ is more than k, or return an annotated rank decomposition that encodes $G \triangle F$ and has width at most k.*

In this section we will show that algorithms for Problem 4.7.1 can be translated to algorithms for computing rankwidth. Before showing that, let us give an algorithm for Twin Flipping by using Lemma 4.6.3. The following basic observation is useful in this algorithm and later in this section.

Observation 4.7.2. *Let G be a graph that contains twins $u, v \in V(G)$. The rankwidth of G is at most the rankwidth of $G - \{v\}$.*

Proof. Observe that if $A \subseteq V(G) \setminus \{v\}$ and $u \in A$, then $\mathsf{cutrk}_{G-\{v\}}(A) = \mathsf{cutrk}_G(A \cup \{v\})$. Therefore, we can construct a rank decomposition of G of equal width from a rank decomposition of $G - \{v\}$ by adding two children c_1, c_2 to the leaf corresponding to u, and mapping u to c_1 and v to c_2. □

Then we give the algorithm for Twin Flipping.

Lemma 4.7.3. *There is a $n \cdot 2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$ time algorithm for Problem 4.7.1.*

Proof. Denote the vertices in X as $X = \{v_1, \ldots, v_{|X|}\}$. Let $G_0 = G$, and for each $i \in [|X|]$ let G_i be the bipartite graph with bipartition (A, B), so that for $j \leqslant i$ it holds that $N_{G_i}(v_j) = N_{G \triangle F}(v_j)$, for $j > i$ it holds that $N_{G_i}(v_j) = N_G(v_j)$, and for $u \in A \setminus X$ it holds that $N_{G_i}(u) = N_G(u) = N_{G \triangle F}(u)$. We have that $G_{|X|} = G \triangle F$ and because for each $v \in X$ there exists $u \in Y$ so that $N_G(v) = N_G(u) = N_{G \triangle F}(u)$, each G_i can be obtained from $G \triangle F$ by adding twins and deleting vertices, which by Observation 4.7.2 implies that if $G \triangle F$ has rankwidth at most k then also G_i for each $i \in [|X|]$ has rankwidth at most k.

Now, for each vertex $v_i \in X$, let F_i be the set of vertices $F_i = \{u \in B \mid v_i u \in F\}$. We can write an edge update sentence $\overline{e}_i$ of size $|\overline{e}_i| = |F_i| + 1$ and constant length that turns G_{i-1} into G_i. Let $\mathcal{T}$ be the given annotated rank decomposition that encodes the graph G. We initialize the data structure of Lemma 4.6.3 with $\mathcal{T}$ and k, and the length bound $d = \mathcal{O}(1)$ of these edge update sentences, which takes $\mathcal{O}_k(n \log^2 n)$ amortized time. We then apply the edge update sentences $\overline{e}_i$ one by one to $\mathcal{T}$. If the data structure at any point returns that the rankwidth would become larger than k, we can return that the rankwidth of $G \triangle F$ is more than k. This takes $|F| \cdot 2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$ amortized time in total.

Finally, we obtain an annotated rank decomposition $\mathcal{T}'$ that encodes $G \triangle F$ and has width at most $4k$. We then use Lemma 4.6.2 to obtain in time $\mathcal{O}_k(n \log n)$ an annotated rank decomposition $\mathcal{T}''$ that encodes $G \triangle F$ and has width at most k or determine that $G \triangle F$ has rankwidth more than k, and then return $\mathcal{T}''$.

The running time is $\mathcal{O}_k(n \log^2 n) + |F| \cdot 2^{\mathcal{O}_k(\sqrt{\log n \log\log n})} = n \cdot 2^{\mathcal{O}_k(\sqrt{\log n \log\log n})}$. □

Then, the rest of this section will be devoted to showing that algorithms for Twin Flipping imply algorithms for computing rankwidth, in particular, to proving the following lemma.

Lemma 4.7.4. *Let $T \colon \mathbb{N} \to \mathbb{N}$ be a function so that there is a $\mathcal{O}_k(T(n))$ time algorithm for Problem 4.7.1. Then there is an algorithm that given an n-vertex m-edge graph G and an integer k, in time $\mathcal{O}_k(T(n) \log^2 n) + \mathcal{O}(m)$ either returns that the rankwidth of G is more than k, or returns an annotated rank decomposition that encodes G and has width at most k.*

Putting Lemmas 4.7.3 and 4.7.4 together implies the first part of Theorem 1.3.6. In particular, as $n \cdot 2^{\mathcal{O}_k(\sqrt{\log n \log\log n})} \leqslant \mathcal{O}_k(n \cdot 2^{\sqrt{\log n} \log\log n} / \log^2 n)$, we can set $T(n) = n \cdot 2^{\sqrt{\log n} \log\log n} / \log^2 n$ to obtain an algorithm with a running time of $\mathcal{O}_k(n \cdot 2^{\sqrt{\log n} \log\log n}) + \mathcal{O}(m)$. Then, we prove in Section 4.10 (Lemma 4.10.4) that given an annotated rank decomposition of width k that encodes G, we can in $\mathcal{O}_k(n)$ time output a $(2^{k+1} - 1)$-expression for cliquewidth of G. This gives the second part of Theorem 1.3.6.

We remark that in the proof of Lemma 4.7.4 we make the natural assumptions that $T(n) \geqslant \Omega(n)$ and $T(n)$ is increasing and convex.

4.7.2 Reduction to bipartite graphs

We will work on bipartite graphs in our algorithm, so the first step is to reduce the task of computing the rankwidth of a graph to bipartite graphs. For this, we will use a reduction given by Courcelle [Cou06] and further analyzed by Oum [Oum08a, Section 4.1].

Let G be a graph. We define $B(G)$ to be the bipartite graph whose vertex set is $V(B(G)) = V(G) \times [4]$, and edge set is defined so that

1. if $v \in V(G)$ and $i \in [3]$, then (v,i) is adjacent to $(v,i+1)$ in $B(G)$ and
2. if $uv \in E(G)$, then $(u,1)$ is adjacent to $(v,4)$ in $B(G)$.

We observe that given an n-vertex m-edge graph G, we can compute $B(G)$ in $\mathcal{O}(n+m)$ time. Oum showed that the rankwidths of G and $B(G)$ are tied to each other.

Lemma 4.7.5 ([Oum08a])**.** *If the rankwidth of G is k, then the rankwidth of $B(G)$ is at least $k/4$ and at most* $\max(2k,1)$.

Even though Oum gives an explicit construction of a rank decomposition of G given a rank decomposition of G, it seems complicated to adapt to work in linear time with annotated rank decompositions. We use an alternative approach by using edge update sentences.

Lemma 4.7.6. *Let G be an n-vertex graph. There is an algorithm that given an annotated rank decomposition $\mathcal{T}$ that encodes $B(G)$ and has width k, in time $\mathcal{O}_k(n \log n)$ returns an annotated rank decomposition that encodes G and has optimum width.*

Proof. Consider an edge update sentence $\overline{e} = (\varphi, X, X_1, X_2, X_3, X_4)$ that has $X = V(B(G))$, $X_i = V(G) \times \{i\}$, and $\varphi(Y, X_1, X_2, X_3, X_4) =$

$$\exists u \in Y, v \in Y.(u \neq v \wedge \forall w \in Y.(u = w \vee v = w)) \wedge u \in X_1 \wedge v \in X_1$$
$$\wedge(\exists u_2 \in X_2, u_3 \in X_3, u_4 \in X_4.(E(u,u_2) \wedge E(u_2,u_3) \wedge E(u_3,u_4) \wedge E(u_4,v)).$$

Let G' be the graph resulting from applying $\overline{e}$ to $B(G)$. We observe that the subgraph of G' induced by $V(G) \times \{1\}$ is equal to G, after renaming every vertex of form $(v,1)$ to v.

Therefore we use our machinery built in previous sections as follows. First, we use Lemma 4.2.2 with $\mathcal{T}$ to compute a rank decomposition $\mathcal{T}^1$ of $B(G)$ of width at most $2k$ and height $\mathcal{O}(\log n)$. Then we use Lemma 4.3.8 with $\mathcal{T}$ and $\mathcal{T}^1$ to obtain an annotated rank decomposition $\mathcal{T}^2$ that encodes $B(G)$, has width at most $2k$, and height $\mathcal{O}(\log n)$. These steps take $\mathcal{O}_k(n \log n)$ time. Then we initialize the $2k$-prefix-rebuilding data structure of Lemma 4.5.5 with $\mathcal{T}^2$ and the parameter d (the bound on the length of an edge update sentence) equal to the length of φ (which is constant), and then apply the $\mathsf{EdgeUpdate}(\overline{e})$ query to obtain an edge update description $\overline{u}$ of width $\ell = \mathcal{O}_k(1)$ that describes G'. This takes $\mathcal{O}_k(n \log n)$ time as the height of $\mathcal{T}^2$ is $\mathcal{O}(\log n)$. Then, we initialize the $2k$-prefix-rebuilding data structure of Lemma 4.3.14 with $\mathcal{T}^2$, and translate $\overline{u}$ to a description $\overline{u}'$ of a prefix-rebuilding update. This takes $\mathcal{O}_{k,\ell}(n) = \mathcal{O}_k(n)$ time. Then, we use Lemma 4.3.2 to apply $\overline{u}'$ to $\mathcal{T}^2$, turning $\mathcal{T}^2$ into an annotated rank decomposition $\mathcal{T}^3$ that encodes G' and has width at most $\max(k,\ell) = \mathcal{O}_k(1)$. Then we use Lemma 4.3.5 to turn $\mathcal{T}^3$ into an annotated rank decomposition of the subgraph of G' induced by $V(G) \times \{1\}$, and then by renaming vertices turn it into an annotated rank decomposition $\mathcal{T}^4$ of G. These steps take $\mathcal{O}_k(n)$ time. Finally we use Lemma 4.6.2 with $\mathcal{T}^4$ to compute an optimum-width rank decomposition that encodes G, and return it. This runs in time $\mathcal{O}_k(n \log n)$. □

Lemmas 4.7.5 and 4.7.6 and the fact that $B(G)$ can be computed from G in $\mathcal{O}(n+m)$ time imply that we can now focus on bipartite graphs.

4.7.3 Twins and near-twins

In this subsection we prove lemmas about finding twins and near-twins in graphs of small rankwidth. The following lemma will be our main tool. Recall here from Section 4.4 that for a rooted rank decomposition $\mathcal{T} = (T, \lambda)$ of a graph G, a set $F \subseteq V(G)$ is a *tree factor* whenever $F = \mathcal{L}(\mathcal{T})[x]$ for some $x \in V(T)$, and a *context factor* whenever F is not a tree factor but $F = F_1 \setminus F_2$ for tree factors F_1, F_2. F is a factor if F is a tree factor or a context factor.

Lemma 4.7.7. *There is an algorithm that given a rooted rank decomposition $\mathcal{T}$ of an n-vertex graph G, an integer $\ell \geqslant 1$, and a set $W \subseteq V(G)$ with $|W| \geqslant 16\ell$, in time $\mathcal{O}(n)$ outputs a set of at least $|W|/(16\ell)$ disjoint factors of $\mathcal{T}$ so that each of them contains at least ℓ vertices in W. The outputted tree factors are represented by single nodes of $\mathcal{T}$ and context factors by pairs of nodes of $\mathcal{T}$.*

Proof. Let $\mathcal{T} = (T, \lambda)$. We say that a node x of T is *important* if $|\mathcal{L}(\mathcal{T})[x] \cap W| \geqslant \ell$. Let us denote the set of important nodes of T by $I \subseteq V(T)$. If a node is important, then also its parent is, so I is a prefix of T. Note that the root of T is important. Let us furthermore say that a node is a *junction* if it is important,

and also either has degree 1 or 3 in $T[I]$ or is the root of T. We denote the set of junctions by $J \subseteq I$. Note that if $x, y \in J$, then the lowest common ancestor of x and y is also in J.

Then we define a rooted tree T' so that $V(T') = J$, there is an edge between $x, y \in J$ if there is a path between x and y in T that avoids other nodes in J, and the root of T' is the root of T. Observe that T' is a rooted tree where each node except the root has either 0 or 2 children, and the root has 1 or 2 children. Now, $V(G)$ can be partitioned into a disjoint union of factors of $\mathcal{T}$ as follows:

- for each leaf l of T' there is a tree factor $\mathcal{L}(\mathcal{T})[l]$,
- for each edge xp of T', where p is the parent of x in T' and c is the child of p on the path from p to x in T there is a context factor $\mathcal{L}(\mathcal{T})[c] \setminus \mathcal{L}(\mathcal{T})[x]$, and
- if c is a child of the root and is not in I, then there is a tree factor $\mathcal{L}(\mathcal{T})[c]$.

We consider cases based on $|V(T')|$. First, suppose that $|V(T')| \geqslant |W|/(8\ell)$. This implies that T' has at least $|W|/(16\ell)$ leaves, so by outputting the leaves of T' we output at least $|W|/(16\ell)$ tree factors that each contains at least ℓ vertices in W.

Then, suppose $|V(T')| \leqslant |W|/(8\ell)$. We note that each tree factor corresponding to a leaf of T' contains at most $2\ell - 1$ vertices in W, and the possible single tree factor corresponding to a child of the root not in I contains at most $\ell - 1$ vertices in W, so therefore the context factors corresponding to the edges of T' contain at least

$$|W| - |V(T')| \cdot (2\ell - 1) - (\ell - 1) \geqslant |W| - \frac{|W| \cdot (2\ell - 1)}{8\ell} - \frac{|W|}{16} \geqslant |W|/2$$

vertices in W. Now, consider an edge xp of T', where p is the parent of x in T'. This corresponds to a path $x, y_1, \ldots, y_t, p$ in T. Then, for each $i \in [t]$ let z_i be the child of y_i that is not on this path. We observe that the context factor associated with xp is equal to $\bigcup_{i=1}^{t} \mathcal{L}(\mathcal{T})[z_i]$, and that for each i it holds that $|\mathcal{L}(\mathcal{T})[z_i] \cap W| < \ell$. This implies that if this context factor contains w vertices in W, then it can be further partitioned into at least $\lfloor \frac{w}{2\ell} \rfloor \geqslant \frac{w}{2\ell} - 1$ context factors that each contain at least ℓ vertices in W, plus at most one context factor that contains less than ℓ vertices in W. By performing this partitioning to all $|E(T')| \leqslant |W|/(8\ell)$ such context factors that in total contain at least $|W|/2$ vertices in W, we obtain at least

$$\frac{|W|/2}{2\ell} - |E(T')| \geqslant |W|/(8\ell)$$

context factors that each contain at least ℓ vertices in W. This procedure clearly can be implemented in $\mathcal{O}(n)$ time given $\mathcal{T}$. □

Then we apply Lemma 4.7.7 to prove that bipartite graphs with small rankwidth and unbalanced bipartition contain a lot of twins.

Lemma 4.7.8. *There is a function $f(k) \in 2^{\mathcal{O}(k)}$, so that if G is a bipartite graph with bipartition (A, B) and rankwidth k, and $|A| \geqslant f(k) \cdot |B|$, then there exist at least $|A|/f(k)$ disjoint pairs of twins in A.*

Proof. We will prove the lemma for $f(k) = 32 \cdot (2^{2k} + 1)$, so assume that $|A| \geqslant f(k) \cdot |B|$. Let $\mathcal{T}$ be a rank decomposition of G of width at most k, and let us apply Lemma 4.7.7 with $\ell = f(k)/32$ and $W = A$. This outputs at least $\frac{2|A|}{f(k)}$ disjoint factors of $\mathcal{T}$ so that each of them contains at least $f(k)/32 = 2^{2k} + 1$ vertices in A. Among them, there are at least $\frac{2|A|}{f(k)} - |B| \geqslant |A|/f(k)$ factors that contain no vertices in B. It suffices to prove that each of these contains a pair of twins in A.

Consider a factor F of $\mathcal{T}$ with $|F| \geqslant 2^{2k} + 1$ and $F \subseteq A$. If F is a tree factor, then $\mathsf{cutrk}_G(F) \leqslant k$ by definition, and if F is a context factor, we can prove by symmetry and submodularity of cutrk_G that $\mathsf{cutrk}_G(F) \leqslant 2k$. Now, Lemma 4.2.4 implies that F has a representative R of size $|R| \leqslant 2^{2k}$. Because $|F| > 2^{2k}$, there exists a vertex $v \in F \setminus R$, and because R is a representative of F, there exists $u \in R$ so that $N(v) \setminus F = N(u) \setminus F$. Because $F \subseteq A$, the vertices u and v are twins. □

We say that two vertices u and v of a graph G are *q-near-twins* if $|N(u) \triangle N(v)| \leqslant q$. Next we use Lemma 4.7.7 to give an algorithm for finding many near-twins in graphs of small rankwidth.

Lemma 4.7.9. *There exists a function $f \in 2^{\mathcal{O}(k)}$ so that there is an algorithm that given an annotated rank decomposition $\mathcal{T}$ of width k that encodes an n-vertex graph G and a set $W \subseteq V(G)$ such that $|W| \geqslant f(k)$, in time $\mathcal{O}_k(n)$ returns $|W|/f(k)$ disjoint pairs of vertices $(u_1, v_1), \ldots, (u_t, v_t)$ in W, so that u_i and v_i are $(f(k) \cdot n/|W|)$-near-twins. The algorithm furthermore returns the sets $N(u_i) \triangle N(v_i)$ for all $i \in [t]$.*

Proof. The proof will use similar ideas to the proof of Lemma 4.7.8. We will prove the the lemma for $f(k) = 32 \cdot (2^{2k} + 1)$. Let us root $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ arbitrarily and apply Lemma 4.7.7 with $\mathcal{T}$, $\ell = f(k)/32$, and the set W. This outputs at least $\frac{2|W|}{f(k)}$ disjoint factors of $\mathcal{T}$ so that each of them contains at least $f(k)/32 = 2^{2k} + 1$ vertices in W. Let us say that a factor $F \subseteq V(G)$ is *big* if $|F| > \frac{f(k)n}{|W|}$ and *small* otherwise. Because the factors are disjoint, there are at most $\frac{|W|}{f(k)}$ big factors, implying that there are at least $\frac{|W|}{f(k)}$ small factors.

Now it suffices to output a single such pair (u_i, v_i) from each small factor. We observe that if F is a small factor and $u, v \in F \cap W$ are two vertices with $N(u) \setminus F = N(v) \setminus F$, then $|N(u) \triangle N(v)| \leqslant |F|$, implying that they are $(f(k) \cdot n/|W|)$-near-twins. It remains to argue that we can find such u and v in $\mathcal{O}_k(|F|)$ time for each small factor F.

First suppose that F is a tree factor, given as $F = \mathcal{L}(\mathcal{T})[x]$ for some $x \in V(T)$, and let p be the parent of x in T. In this case, the subtree below x in T has $\mathcal{O}(|F|)$ nodes. For each vertex $v \in F$ there exists a vertex $w \in \mathcal{R}(\vec{xp})$ so that $N(v) \setminus F = N(w) \setminus F$, and given v we can find such vertex w in time $\mathcal{O}(|F|)$ by following the mapping $\mathcal{F}$ of $\mathcal{T}$. We iterate through vertices in $F \cap W$ until we find two vertices $u, v \in F \cap W$ with the same such vertex w. This implies that $N(u) \setminus F = N(v) \setminus F$, so we can return the pair (u, v). As $|\mathcal{R}(\vec{xp})| \leqslant 2^k$, finding such u and v takes at most $2^k + 1$ iterations of finding such w, resulting in $\mathcal{O}_k(|F|)$ time, and we are guaranteed to find such u and v because $|F \cap W| \geqslant 2^{2k} + 1$. To compute $N(u) \triangle N(v)$, we first compute $N(u) \cap F$ and $N(v) \cap F$ in $\mathcal{O}_k(|F|)$ time by modifying the method of Lemma 4.3.7 so that we follow the mapping $\mathcal{F}$ only inside the subtree below x. Then, we can output $(N(u) \cap F) \triangle (N(v) \cap F) = N(u) \triangle N(v)$.

Then suppose F is a context factor, given as $F = \mathcal{L}(\mathcal{T})[x] \setminus \mathcal{L}(\mathcal{T})[y]$ for some nodes $x, y \in V(T)$, so that y is a descendant of x. Let p^x be the parent of x and p^y the parent of y. We have that the subtree of T consisting of the descendants of x minus the descendants of y has $\mathcal{O}(|F|)$ nodes. Again, for each vertex $v \in F$ there exists a vertex $w^x \in \mathcal{R}(x\vec{p^x})$ so that $N(v) \setminus \mathcal{L}(\mathcal{T})[x] = N(w^x) \setminus \mathcal{L}(\mathcal{T})[x]$ and a vertex $w^y \in \mathcal{R}(p^{\vec{y}}y)$ so that $N(v) \cap \mathcal{L}(\mathcal{T})[y] = N(w^y) \cap \mathcal{L}(\mathcal{T})[y]$, and we can find such w^x and w^y in $\mathcal{O}(|F|)$ time given v by following the mapping $\mathcal{F}$ of $\mathcal{T}$. Now, if we find two vertices $u, v \in F \cap W$ with the same such pair (w^x, w^y), then $N(u) \setminus F = N(v) \setminus F$. Because $|\mathcal{R}(x\vec{p^x})|, |\mathcal{R}(p^{\vec{y}}y)| \leqslant 2^k$, there are at most 2^{2k} such pairs, so we find such u, v within the first $2^{2k}+1$ iterations, resulting in $\mathcal{O}_k(|F|)$ time. The set $N(u) \triangle N(v)$ can be computed in $\mathcal{O}_k(|F|)$ time by similar arguments as in the previous case. □

Then we give a data structure for finding twins guaranteed by Lemma 4.7.8 efficiently in a certain setting where we consider induced subgraphs defined by an interval. For a graph G and a vertex set $X \subseteq V(G)$, the *twin-equivalence classes* of X in G are the maximal sets $X' \subseteq X$ so that any two vertices in X' are twins in G.

Lemma 4.7.10. *There is a data structure that is initialized with an n-vertex m-edge bipartite graph G given with a bipartition (A, B), where B is indexed as $B = \{v_1, \ldots, v_{|B|}\}$, and supports the following query:*

- $\mathsf{Twins}(X, \ell, r)$*: Given a set $X \subseteq A$ and two integers ℓ, r with $1 \leqslant \ell \leqslant r \leqslant |B|$, in time $\mathcal{O}(|X| \log n)$ returns the twin-equivalence classes of X in the graph $G[X, \{v_\ell, \ldots, v_r\}]$.*

The initialization time of the data structure is $\mathcal{O}(n + m)$.

Proof. We will use tools from the theory of string algorithms: the *suffix array* and the *LCP array*. For a string $S = s_1, \ldots, s_t$ of length t, the suffix array of S is the array SA of length t that at position $i \in [t]$ stores the index $\mathsf{SA}[i] \in [t]$ so that the ith lexicographically smallest suffix of S starts at index $\mathsf{SA}[i]$ of S. The LCP array associated with S and SA is the array LCP of length $t - 1$ that at position $i \in [t - 1]$ stores the length $\mathsf{LCP}[i]$ of the longest common prefix of the suffix of S starting at $\mathsf{SA}[i]$ and the suffix of S starting at $\mathsf{SA}[i + 1]$. It is known that both the suffix array and the LCP array of a given string can be computed in linear time [KSB06].

The initialization of our data structure works as follows. We consider the total order of the vertices $B = \{v_1, \ldots, v_{|B|}\}$ so that $v_i < v_j$ whenever $i < j$. First we use bucket sort to sort the neighborhoods $N(a)$ of each vertex $a \in A$ into an ordered list, in total time $\mathcal{O}(n + m)$. Then we concatenate these lists into a string S of length m, so that for each vertex $a \in A$, the neighborhood of a corresponds to a substring $S[L_a, R_a] = s_{L_a}, \ldots, s_{R_a}$ of S, in which the neighbors of a occur in the sorted order. We store the indices L_a and R_a of each $a \in A$. We then compute the suffix array SA and the LCP array LCP of S by using the algorithm of [KSB06] in $\mathcal{O}(m)$ time. We also compute the inverse array of SA, in particular, the array InvSA so that for each $i \in [m]$ it holds that $\mathsf{SA}[\mathsf{InvSA}[i]] = i$. Finally, we compute a range minimum query

data structure on the LCP array, in particular, a data structure that can answer queries that given indices $\ell, r \in [m]$, report $\min_{i\in[\ell,r]} \mathsf{LCP}[i]$. Such data structure that answers queries in $\mathcal{O}(\log m) = \mathcal{O}(\log n)$ time can be computed by folklore techniques with binary trees in $\mathcal{O}(m)$ time. All together, the initialization works in $\mathcal{O}(n+m)$ time.

Then the $\mathsf{Twins}(X, \ell, r)$ query is implemented as follows. Let us denote $Y = \{v_\ell, \ldots, v_r\}$. First, for each $a \in X$, we use binary search to compute the indices L'_a, R'_a so that the neighborhood $N(a) \cap Y$ of a into Y corresponds to the substring $S[L'_a, R'_a]$, or decide that the neighborhood of a into Y is empty. This takes $\mathcal{O}(|X| \log n)$ time. The first equivalence class is the vertices in X whose neighborhood into Y is empty. Then, based on the computed indices L'_a and R'_a, we know for each $a \in X$ the size $|N(a) \cap Y|$. We group the remaining vertices in X based on $|N(a) \cap Y|$, which can be done in $\mathcal{O}(|X| \log n)$ time. It remains to consider the problem where given $X' \subseteq X$ so that each $a \in X'$ has exactly $p \geqslant 1$ neighbors in Y, we have to compute the twin-equivalence classes of X' in $G[X', Y]$.

Consider two vertices $a, b \in X'$ and assume $\mathsf{InvSA}[L'_a] < \mathsf{InvSA}[L'_b]$; so the suffix of S starting at index L'_a is lexicographically smaller than the suffix starting at index L'_b. Then $N(a) \cap Y = N(b) \cap Y$ holds if and only if these suffixes share a common prefix of length p, or equivalently $p \leqslant \min_{i\in[\mathsf{InvSA}[L'_a],\mathsf{InvSA}[L'_b]-1]} \mathsf{LCP}[i]$. Therefore, to compute the twin-equivalence classes of X', we first sort X' based on the integers $\mathsf{InvSA}[L'_a]$ in time $\mathcal{O}(|X'| \log n)$, then assuming this sorted order of X' is $a_1, \ldots, a_{|X'|}$, we compute for each $i \in [|X'|-1]$ the integer $z_i = \min_{j\in[\mathsf{InvSA}[L'_{a_i}],\mathsf{InvSA}[L'_{a_{i+1}}]-1]} \mathsf{LCP}[j]$ by using the range minimum query data structure in $\mathcal{O}(|X'| \log n)$ time. Now we have that a_i and a_j with $i < j$ have $N(a_i) \cap Y = N(a_j) \cap Y$ if and only if $p \leqslant \min_{k\in[i,j-1]} z_k$, so with this information we can output the twin-equivalence classes of X' in $\mathcal{O}(|X'|)$ time. Therefore, the total time to answer the query is $\mathcal{O}(|X| \log n)$. □

Then we show that the method of adding twins to a rank decomposition discussed in Observation 4.7.2 can be efficiently implemented on annotated rank decompositions.

Lemma 4.7.11. *Let G be a graph with twins $u, v \in V(G)$. Suppose a representation of an annotated rank decomposition $\mathcal{T}'$ that encodes $G - \{v\}$ and has width k is already stored. Then, given u and v, the representation of $\mathcal{T}'$ can in time $\mathcal{O}(1)$ be turned into a representation of an annotated rank decomposition $\mathcal{T}$ that encodes G and has width k.*

Proof. We implement the construction discussed in the proof of Observation 4.7.2. Denote the stored decomposition by $\mathcal{T}' = (T', V(G) \setminus \{v\}, \mathcal{R}', \mathcal{E}', \mathcal{F}')$ and let $\vec{lp} \in \vec{L}(T')$ so that $\mathcal{R}'(\vec{lp}) = \{u\}$. We construct $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ as follows. The tree T is created by adding two children c_1 and c_2 for the leaf l of T'. The annotations for edges of T that exist in T' are directly copied from $\mathcal{T}'$ to $\mathcal{T}$. Then we set $\mathcal{R}(\vec{c_1 l}) := \{u\}$, $\mathcal{R}(\vec{c_2 l}) := \{v\}$, and $\mathcal{R}(\vec{lc_1}) := \mathcal{R}(\vec{lc_2}) := \mathcal{R}'(\vec{pl})$. We also set $\mathcal{E}(c_1 l) := \mathcal{E}'(lp)$ and obtain $\mathcal{E}(c_2 l)$ by replacing u by v in $\mathcal{E}'(lp)$. The functions $\mathcal{F}(c_1 lp)$ and $\mathcal{F}(c_2 lp)$ both map to the single vertex $u \in \mathcal{R}(\vec{lp})$.

We can verify that $\mathcal{T}$ is indeed an annotated rank decomposition that encodes G, and whose width is at most the width of $\mathcal{T}'$. The construction can be implemented in $\mathcal{O}(1)$ time because $|\mathcal{R}'(\vec{lp})| = 1$ and $|\mathcal{R}'(\vec{pl})| \leqslant 2$. □

4.7.4 Proof of Lemma 4.7.4

Before finally proving Lemma 4.7.4, let us give the crucial subroutine for which the algorithm for Problem 4.7.1 is used.

Lemma 4.7.12. *Let $T\colon \mathbb{N} \to \mathbb{N}$ be a function so that there is a $\mathcal{O}_k(T(n))$ time algorithm for Problem 4.7.1. Let also G be a bipartite graph with bipartition $(X, Y_1 \cup Y_2)$, where Y_1 and Y_2 are disjoint. There is an algorithm that given an annotated rank decomposition $\mathcal{T}_1$ of width at most k that encodes $G[X, Y_1]$ and an annotated rank decomposition $\mathcal{T}_2$ of width at most k that encodes $G[X, Y_2]$, either returns that the rankwidth of G is more than k, or returns an annotated rank decomposition that encodes G and has width at most k. The algorithm runs in time $\mathcal{O}_k(T(n) \log n)$, where $n = |X| + |Y_1| + |Y_2|$.*

Proof. The algorithm is recursive. Let f be the function from Lemma 4.7.9.

We first consider the base case that $|Y_2| \leqslant f(k)$. If Y_1 is empty, we can simply return $\mathcal{T}_2$. Otherwise, let v be an arbitrary vertex in Y_1. We use Lemma 4.7.11 to add to $\mathcal{T}_1$ for each vertex $u \in Y_2$ two new vertices u' and u'' as twins of v, and denote by Y'_2 the set of such vertices u' and by Y''_2 such vertices u''. Let G' denote the resulting graph. The rankwidth of G' is at most k because it is created from $G[X, Y_1]$ by adding twins.

We use Lemma 4.3.7 with $\mathcal{T}_2$ to compute for each $u \in Y_2$ the neighborhood $N(u)$, and with $\mathcal{T}_1$ to compute $N(v)$. Then, we compute $F = \{u'w \mid u' \in Y_2', w \in N(u) \triangle N(v)\}$. As $|Y_2| \leqslant f(k)$, this takes $\mathcal{O}_k(n)$ time, which is also an upper bound for $|F|$. We observe that the graph $G' \triangle F$ is isomorphic to a graph created from $G[X, Y_1 \cup Y_2]$ by adding a twin for each vertex in Y_2. Then we apply the algorithm for Twin Flipping (Problem 4.7.1) with the sets Y_2', Y_2'', and F, and the decomposition $\mathcal{T}_1$ to obtain either that the rankwidth of $G' \triangle F$ is more than k, in which case we can return that the rankwidth of $G[X, Y_1 \cup Y_2]$ is more than k, or an annotated rank decomposition $\mathcal{T}'$ of $G' \triangle F$ of width at most k. This takes $\mathcal{O}_k(T(n))$ time. Then, $\mathcal{T}'$ can be turned into an annotated rank decomposition $\mathcal{T}$ of $G[X, Y_1 \cup Y_2]$ by using Lemma 4.3.5 to delete Y_2'' and renaming all vertices $u' \in Y_2'$ to $u \in Y_2$. This takes $\mathcal{O}_k(n)$ time. This finishes the description of the base case. The total running time in this case is $\mathcal{O}_k(n) + \mathcal{O}_k(T(n)) = \mathcal{O}_k(T(n))$ (we assume $T(n) \geqslant \Omega(n)$).

Then consider the case that $|Y_2| > f(k)$. We first apply Lemma 4.7.9 with $\mathcal{T}_2$ and Y_2 to find $|Y_2|/f(k)$ disjoint pairs of vertices $(u_1, v_1), \ldots, (u_t, v_t)$ so that u_i and v_i are $(f(k) \cdot n/|Y_2|)$-near-twins, and the sets $N(u_i) \triangle N(v_i)$. We let $F = \bigcup_{i=1}^{t} \{v_i w \mid w \in N(u_i) \triangle N(v_i)\}$. This runs in $\mathcal{O}_k(n)$ time, which is also an upper bound for $|F|$. Let $Y_2' = \{u_1, \ldots, u_t\}$ and $Y_2'' = \{v_1, \ldots, v_t\}$. We use Lemma 4.3.5 to obtain an annotated rank decomposition $\mathcal{T}_2'$ that encodes $G[X, Y_2 \setminus Y_2'']$, and call the algorithm recursively with $\mathcal{T}_1$ and $\mathcal{T}_2'$. If it returns that the rankwidth of $G[X, Y_1 \cup Y_2 \setminus Y_2'']$ is more than k, then we can return that the rankwidth of $G[X, Y_1 \cup Y_2]$ is more than k. Otherwise, let $\mathcal{T}$ be the returned annotated rank decomposition that encodes $G[X, Y_1 \cup Y_2 \setminus Y_2'']$ and has width at most k. We insert the vertices $Y_2'' = \{v_1, \ldots, v_t\}$ into $\mathcal{T}$ with Lemma 4.7.11 so that v_i is inserted as a twin of u_i. Let G' be the graph that the resulting decomposition encodes. We have that $G' \triangle F = G[X, Y_1 \cup Y_2]$, and we apply the algorithm for Problem 4.7.1 with this decomposition and the sets Y_2', Y_2'', and F. This either returns that the rankwidth of $G[X, Y_1 \cup Y_2]$ is more than k or an annotated rank decomposition of $G[X, Y_1 \cup Y_2]$ of width at most k. This finishes the description of the recursive case. The total running time of also this case, not counting the time spent in the recursive call, is also $\mathcal{O}_k(T(n))$.

At each level of recursion the size of Y_2 decreases by at least $|Y_2|/f(k)$, so the depth of the recursion is $\mathcal{O}_k(\log |Y_2|)$. At each level the running time is $\mathcal{O}_k(T(n))$, so the total running time is $\mathcal{O}_k(T(n) \log |Y_2|) = \mathcal{O}_k(T(n) \log n)$. □

Then we prove Lemma 4.7.4, which we restate here.

Lemma 4.7.4. *Let $T \colon \mathbb{N} \to \mathbb{N}$ be a function so that there is a $\mathcal{O}_k(T(n))$ time algorithm for Problem 4.7.1. Then there is an algorithm that given an n-vertex m-edge graph G and an integer k, in time $\mathcal{O}_k(T(n) \log^2 n) + \mathcal{O}(m)$ either returns that the rankwidth of G is more than k, or returns an annotated rank decomposition that encodes G and has width at most k.*

Proof. By Lemmas 4.7.5 and 4.7.6, proving the lemma under the assumption that G is bipartite implies the lemma for general G. Therefore, we then assume that G is bipartite. Let us fix a bipartition (A, B) of G and an indexing $B = \{v_1, \ldots, v_{|B|}\}$ of B, and initialize the data structure of Lemma 4.7.10 with these. This takes $\mathcal{O}(n + m)$ time.

We will describe a recursive algorithm that takes as input

- a subset $X \subseteq A$ and two integers ℓ, r with $1 \leqslant \ell \leqslant r \leqslant |B|$,

and outputs

- either an annotated rank decomposition of $G[X, \{v_\ell, \ldots, v_r\}]$ of width at most k, or that the graph $G[X, \{v_\ell, \ldots, v_r\}]$ has rankwidth more than k.

We denote $Y = \{v_\ell, \ldots, v_r\}$. If $|Y| = 1$, we compute an annotated rank decomposition of $G[X, Y]$ of width at most 1 in time $\mathcal{O}(|X| + |Y|)$ and return it. Then assume $|Y| \geqslant 2$.

We first use the data structure of Lemma 4.7.10 to find the twin-equivalence classes of X in $G[X, Y]$, and then compute a set $X' \subseteq X$ that contains exactly one vertex from each of the equivalence classes. We also store for each vertex $u \in X \setminus X'$ a vertex $u_x \in X'$ so that $N_{G[X,Y]}(u) = N_{G[X,Y]}(u_x)$. This step takes in total $\mathcal{O}(|X| \log n)$ time. Let $f(k)$ be the function from Lemma 4.7.8. The lemma implies that if $|X'| \geqslant f(k) \cdot |Y|$, then the rankwidth of $G[X', Y]$ (and thus also of $G[X, Y]$) is more than k. In this case we can return immediately, Then assume $|X'| \leqslant \mathcal{O}_k(|Y|)$.

We select $t \in [\ell, r-1]$ so that both $Y_1 = \{v_\ell, \ldots, v_t\}$ and $Y_2 = \{v_{t+1}, \ldots, v_r\}$ have size either $\lfloor |Y|/2 \rfloor$ or $\lceil |Y|/2 \rceil$. Then we make two recursive calls of the algorithm, one with X' and ℓ, t, and another with X' and $t+1, r$. If either of the calls returns that the graph has rankwidth more than k, we can return that the rankwidth of $G[X, Y]$ is more than k. Otherwise, let $\mathcal{T}_1$ be the decomposition returned by the first call

and $\mathcal{T}_2$ the decomposition returned by the second call. We apply the algorithm of Lemma 4.7.12 with these decompositions to either conclude that the rankwidth of $G[X,Y]$ is more than k, or to obtain an annotated rank decomposition $\mathcal{T}$ of $G[X',Y]$ of width at most k. This runs in $\mathcal{O}_k(T(|X'|+|Y|)\log(|X'|+|Y|))$ time. Finally, we insert the vertices $X \setminus X'$ to the decomposition in $\mathcal{O}(|X \setminus X'|)$ time by using Lemma 4.7.11, and return the resulting decomposition. This completes the description of the algorithm.

We observe that the running time of each recursive call, not counting the time spent in the subcalls, is $\mathcal{O}_k(T(|X|+|Y|)\log n)$. The sum of the sizes of the sets Y over all such calls is $\mathcal{O}(n \log n)$. On all calls except the first, it is guaranteed that $|X| \leqslant \mathcal{O}_k(|Y|)$, so the sum of sizes of the sets X over all such calls is $\mathcal{O}_k(n \log n)$. Then, the facts that $|X|+|Y| \leqslant n$ in each call and the function T is convex imply that the total running time past the initialization of the data structure of Lemma 4.7.10 is $\mathcal{O}_k(T(n)\log^2 n)$. This concludes the proof since the data structure is initialized in $\mathcal{O}(n+m)$ time. □

4.8 Dealternation Lemma

In this section, we prove the Dealternation Lemma announced in Lemma 4.4.1:

Lemma 4.4.1. *There exists a function $f(\ell)$ so that if G is a graph of rankwidth k and $\mathcal{T}$ a rooted rank decomposition of G of width ℓ, then there exists a rooted rank decomposition $\mathcal{T}'$ of G of width k so that for every node $t \in V(T)$, the set $\mathcal{L}(\mathcal{T})[t]$ can be partitioned into a disjoint union of $f(\ell)$ factors of $\mathcal{T}'$.*

We will actually prove a slightly more general result, showing an analog of the Dealternation Lemma for *subspace arrangements* – structures described by families of linear spaces that generalize the notions of graphs, hypergraphs and linear matroids.

We begin by introducing the concepts and notation used throughout the proof.

4.8.1 Section-specific preliminaries

Linear spaces. Let $\mathbb{F}$ be a fixed finite field; in this chapter we assume $\mathbb{F} = \mathrm{GF}(2)$. The linear space over $\mathbb{F}$ of dimension d is denoted by $\mathbb{F}^d$. Given two linear subspaces V_1, V_2 of $\mathbb{F}^d$, we denote by $V_1 + V_2$ their sum and by $V_1 \cap V_2$ their intersection. By $\dim(V)$ we denote the dimension of the subspace V of $\mathbb{F}^d$.

The following facts are standard.

Lemma 4.8.1. *For any two linear subspaces V_1, V_2 of $\mathbb{F}^d$, we have that*

$$\dim(V_1) + \dim(V_2) = \dim(V_1 + V_2) + \dim(V_1 \cap V_2).$$

Lemma 4.8.2 ([JKO17, Lemma 25])**.** *For any four linear subspaces U_1, U_2, V_1, V_2 of $\mathbb{F}^d$, we have that*

$$\begin{aligned}&\dim((U_1+U_2)\cap(V_1+V_2)) + \dim(U_1\cap U_2) + \dim(V_1\cap V_2)\\ &= \dim((U_1+V_1)\cap(U_2+V_2)) + \dim(U_1\cap V_1) + \dim(U_2\cap V_2).\end{aligned}$$

For any set of vectors $A \subseteq \mathbb{F}^d$, we denote by $\langle A\rangle$ the subspace of $\mathbb{F}^d$ spanned by the vectors of A. If $\dim(\langle A\rangle) = |A|$, then we say that A is a *basis* of $\langle A\rangle$. Then any permutation $\mathfrak{B}$ of elements of A is called an *ordered basis* of $\langle A\rangle$; for convenience, we define that $\langle\mathfrak{B}\rangle = \langle A\rangle$. Letting $\mathfrak{B} = (\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_c)$, we have that every vector $\mathbf{u} \in \langle\mathfrak{B}\rangle$ can be uniquely represented as a linear combination $\mathbf{u} = \sum_{i=1}^{c} \alpha_i \mathbf{v}_i$. In this chapter, whenever the ordered basis $\mathfrak{B}$ of a vector space V is known from context, all vectors $\mathbf{u} \in V$ will be implicitly represented as such a linear combination. Similarly, subspaces of V are then implicitly represented as $\langle\{\mathbf{u}_1, \ldots, \mathbf{u}_d\}\rangle$, where $\mathbf{u}_1, \ldots, \mathbf{u}_d \in V$ are implicitly represented as linear combinations of vectors of $\mathfrak{B}$. Such a representation can be then stored using $\mathcal{O}(cd)$ elements of $\mathbb{F}$.

Subspace arrangements and rank decompositions. Let $d \in \mathbb{N}$ and consider the linear space $\mathbb{F}^d$. Any family $\mathcal{V} = \{V_1, V_2, \ldots, V_n\}$ of linear subspaces of $\mathbb{F}^d$ is called a *subspace arrangement*. For visual clarity, let $\langle\mathcal{V}\rangle = \sum_{i=1}^{n} V_i$ be the sum of all subspaces in the arrangement.

A *rank decomposition* of a subspace arrangement $\mathcal{V}$ is a pair $\mathcal{T} = (T, \lambda)$, where T is a cubic tree and λ is a bijection $\lambda : \mathcal{V} \to \vec{L}(T)$. For an oriented edge $\vec{uv} \in \vec{E}(T)$, we denote by $\mathcal{L}(\mathcal{T})[\vec{uv}] = \{\lambda^{-1}(\vec{lp}) \mid \vec{lp} \in \vec{L}(T)[\vec{uv}]\}$ the subfamily of $\mathcal{V}$ comprising all linear subspaces that are mapped to leaf edges that are closer to u than v. The *boundary space* of an edge uv is defined as $B_{uv} = \langle\mathcal{L}(\mathcal{T})[\vec{uv}]\rangle \cap \langle\mathcal{L}(\mathcal{T})[\vec{vu}]\rangle$.

A *rooted rank decomposition* is defined analogously to a rank decomposition, only that T is a binary tree. Recall that a rank decomposition can be rooted by subdividing a single edge uv once – replacing it

with a path urv – and rooting the tree at r. The boundary space of a nonroot node v with parent p is $B_v = B_{vp}$ and the boundary space of the root r is $B_r = \{\mathbf{0}\}$. Also, we set $\mathcal{L}(\mathcal{T})[v] = \mathcal{L}(\mathcal{T})[\vec{vp}]$ for $v \neq r$ and $\mathcal{L}(\mathcal{T})[r] = \mathcal{V}$.

The width of an edge $uv \in E(T)$ is defined as $\dim(B_{uv})$. The width of a rank decomposition is the maximum width of any edge of the decomposition. Thus, the width of a rooted rank decomposition is equivalently the maximum value of $\dim(B_v)$ ranging over nonroot nodes v.

Rank decompositions of (partitioned) graphs can be transformed to equivalent rank decompositions of subspace arrangements; the reduction is shown below, but it is also present in [JKO21].

Suppose G is a graph; for simplicity, assume $V(G) = \{1, \ldots, |V(G)|\}$. Consider the vector space $\mathrm{GF}(2)^{|V(G)|}$ and its canonical basis $\{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_{|V(G)|}\}$. To each vertex $v \in V(G)$ assign the vector space A_v spanned by the vectors $\mathbf{e}_v$ and $\sum_{u \in N(v)} \mathbf{e}_u$, which we will call the *canonical subspace* of v. Similarly, for a set $S \subseteq V(G)$, we assign to it the canonical subspace $A_S := \sum_{v \in S} A_v$. It is then straightforward to verify that:

Lemma 4.8.3 ([JKO17, Lemma 52])**.** *For any set $S \subseteq V(G)$, we have*

$$\dim\left(A_S \cap A_{V(G)\setminus S}\right) = 2 \cdot \mathsf{cutrk}(S).$$

We then immediately have that:

Lemma 4.8.4. *Let $(G, \mathcal{C})$ be a partitioned graph with $\mathcal{C} = \{S_1, S_2, \ldots, S_n\}$. Let $\mathcal{V} = \{V_1, V_2, \ldots, V_n\}$ be a subspace arrangement over $GF(2)^{|V(G)|}$, where $V_i = A_{S_i}$ for each $i \in [n]$. Then $\mathcal{V}$ satisfies the following property.*

Let T be a cubic tree with leaves $\ell_1, \ldots, \ell_n$. Define bijections $\lambda_1 : \mathcal{C} \to \vec{L}(T)$ and $\lambda_2 : \mathcal{V} \to \vec{L}(T)$ so that for every $i \in [n]$, both $\lambda_1(S_i)$ and $\lambda_2(V_i)$ are assigned to the oriented edge incident to ℓ_i. Note that $\mathcal{T} = (T, \lambda_1)$ is a rank decomposition of $(G, \mathcal{C})$ and $\mathcal{T}' = (T, \lambda_2)$ is an (isomorphic) rank decomposition of $\mathcal{V}$. Then the width of $\mathcal{T}'$ is equal to twice the width of $\mathcal{T}$.

We encourage the reader to check [JKO21] to find how rankwidth of subspace arrangement also generalizes the notions of *branchwidth of representable matroids*, *branchwidth of graphs* and *carving-width of graphs*.

We then generalize the statement of the Dealternation Lemma (Lemma 4.4.1) to the rank decompositions of subspace arrangements. Mimicking the concepts defined for graphs, we say that a set $F \subseteq \mathcal{V}$ is a tree factor of $\mathcal{T} = (T, \lambda)$ if $F = \mathcal{L}(\mathcal{T})[t]$ for some $t \in V(T)$; and a context factor if it is not a tree factor, but a set of the form $F = F_1 \setminus F_2$, where F_1 and F_2 are tree factors of $\mathcal{T}$. F is a factor of $\mathcal{T}$ if it is either a tree factor or a context factor of $\mathcal{T}$. Then:

Lemma 4.8.5 (Dealternation Lemma for subspace arrangements)**.** *There exists a function $f_{4.8.5} : \mathbb{N} \to \mathbb{N}$ so that if $\mathcal{V}$ is a subspace arrangement and $\mathcal{T}^b = (T^b, \lambda^b)$ is a rooted rank decomposition of $\mathcal{V}$ of width $\ell \geqslant 0$, then there exists a rooted rank decomposition $\mathcal{T}$ of $\mathcal{V}$ of optimum width so that for every node $t \in V(T^b)$, the set $\mathcal{L}(\mathcal{T}^b)[t]$ can be partitioned into a disjoint union of at most $f_{4.8.5}(\ell)$ factors of $\mathcal{T}$.*

Note that Lemma 4.8.5 directly implies the Dealternation Lemma through Lemma 4.8.4. Hence, the rest of this section will be devoted to the proof of Lemma 4.8.5.

Fullness, emptiness and mixedness of edges and nodes. Let $\mathcal{V} = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_n\}$ be a subspace arrangement and $\mathcal{T}^b = (T^b, \lambda^b)$ be a rooted rank decomposition of $\mathcal{V}$ (possibly of unoptimal width). We introduce the ancestor-descendant relationship on the nodes of T^b: we say $x \leqslant y$ whenever $x = y$ or x is a descendant of y, and by $x < y$ we mean $x \leqslant y$ and $x \neq y$. Moreover, define $\mathcal{V}_x = \mathcal{L}(\mathcal{T}^b)[x]$ as the subfamily of $\mathcal{V}$ comprising those subspaces $\mathcal{V}_i$ that are mapped to the leaf edges $\vec{lp}$ with $x \geqslant l$. Note that $\mathcal{V}_r = \mathcal{V}$ if and only if r is the root of T^b, and $|\mathcal{V}_l| = 1$ if and only if l is a leaf of T^b. We will then say that each $V \in \mathcal{V}_x$ is *in the subtree of $\mathcal{T}^b$ rooted at x*. We remark that if $x, y \in V(T^b)$ with $x \leqslant y$, then $\mathcal{V}_x \subseteq \mathcal{V}_y$; and whenever x, y are not in the ancestor-descendant relationship in T^b, then $\mathcal{V}_x \cap \mathcal{V}_y = \emptyset$.

For the following description, consider a node x of T^b. Let $\mathcal{T} = (T, \lambda)$ be a rank decomposition of $\mathcal{V}$ (rooted or unrooted). Define $\mathcal{L}_x(\mathcal{T})[\vec{uv}] = \mathcal{L}(\mathcal{T})[\vec{uv}] \cap \mathcal{V}_x$ as the family of linear spaces containing exactly those linear spaces $V \in \mathcal{V}$ that:

- are in the subtree of (T^b, λ^b) rooted at x; and
- in (T, λ), are mapped to a leaf edge closer to u than v.

Similarly, we set $\mathcal{L}_{\bar{x}}(\mathcal{T})[\vec{uv}] = \mathcal{L}(\mathcal{T})[\vec{uv}] \setminus \mathcal{V}_x = \mathcal{L}(\mathcal{T})[\vec{uv}] \setminus \mathcal{L}_x(\mathcal{T})[\vec{uv}]$. Note that if an edge $v_1\vec{v}_2$ is a predecessor of an edge $v_3\vec{v}_4$ in T, then $\mathcal{L}_x(\mathcal{T})[v_1\vec{v}_2] \subseteq \mathcal{L}_x(\mathcal{T})[v_3\vec{v}_4]$ and $\mathcal{L}_{\bar{x}}(\mathcal{T})[v_1\vec{v}_2] \subseteq \mathcal{L}_{\bar{x}}(\mathcal{T})[v_3\vec{v}_4]$.

We also say that a directed edge $\vec{uv}$ of T is:

- *x-full* if $\mathcal{L}(\mathcal{T})[\vec{uv}] \subseteq \mathcal{V}_x$; that is, for every leaf edge e of (T, λ) closer to u than v, e is mapped to a space $V \in \mathcal{V}$ in the subtree of $\mathcal{T}^b$ rooted at x;
- *x-empty* if $\mathcal{L}(\mathcal{T})[\vec{uv}] \cap \mathcal{V}_x = \emptyset$, or equivalently, $\mathcal{L}_x(\mathcal{T})[\vec{uv}] = \emptyset$;
- *x-mixed* otherwise.

Similarly, if $\mathcal{T}$ is rooted, then we additionally say that a node $v \in V(T)$ is x-full (resp. x-empty or x-mixed) if $\mathcal{L}(\mathcal{T})[v] \subseteq \mathcal{V}_x$ (resp. $\mathcal{L}(\mathcal{T})[v] \cap \mathcal{V}_x = \emptyset$ or $\mathcal{L}(\mathcal{T})[v] \cap \mathcal{V}_x \notin \{\emptyset, \mathcal{L}(\mathcal{T})[v]\}$). Equivalently for nonroot nodes v, v is x-full (resp. x-empty, x-mixed) if and only if the directed edge $\vec{vp}$ is x-full (resp. x-empty, x-mixed), where p is the parent of v in T.

The following observation shows how the notions of fullness, emptiness and mixedness of edges of T are related for pairs of nodes of T^b:

Observation 4.8.6. *Let $x, y \in V(T^b)$ and $\vec{uv} \in \vec{E}(T)$.*

- *If $\vec{uv}$ is x-empty and $y \leqslant x$, then $\vec{uv}$ is y-empty.*
- *If $\vec{uv}$ is x-mixed and $y \not> x$, then $\vec{uv}$ is y-empty or y-mixed.*
- *If $\vec{uv}$ is x-mixed and $y \geqslant x$, then $\vec{uv}$ is y-mixed or y-full.*
- *If $\vec{uv}$ is x-full and $y \geqslant x$, then $\vec{uv}$ is y-full.*
- *If $\vec{uv}$ is x-full and x, y are not in the ancestor-descendant relationship, then $\vec{uv}$ is y-empty.*

Naturally, Observation 4.8.6 directly translates to the fullness, emptiness and mixedness of nodes of T whenever T is rooted.

Well-structured rank decompositions. In the proof we will use the result of Jeong, Kim and Oum [JKO21] asserting the existence of well-structured rank decompositions of subspace arrangements of optimum width, called *totally pure rank decompositions*. Intuitively, a rank decomposition $\mathcal{T}$ of a subspace arrangement $\mathcal{V}$ is totally pure with respect to another rank decomposition $\mathcal{T}^b$ if, for every $x \in V(\mathcal{T}^b)$, $\mathcal{T}$ excludes some small local patterns defined in terms of subspaces $\mathcal{L}_x(\mathcal{T})[\vec{uv}]$ for $\vec{uv} \in \vec{E}(\mathcal{T})$. The formal definition follows below; we mostly follow the notation of [JKO21].

Let $\mathcal{T} = (T, \lambda)$ be an unrooted rank decomposition and $\mathcal{T}^b = (T^b, \lambda^b)$ be rooted. Let also $x \in V(T^b)$. We say that $\mathcal{T}$ is *x-disjoint* if either x is the root of T^b, or T contains an edge uv such that $\mathcal{L}(\mathcal{T})[\vec{uv}] = \mathcal{V}_x$ (equivalently, $\vec{uv}$ is x-full and $\vec{vu}$ is x-empty).

Let B_x be the boundary space of x in $\mathcal{T}^b$, defined as $B_x = \langle \mathcal{L}(\mathcal{T}^b)[\vec{xp}] \rangle \cap \langle \mathcal{L}(\mathcal{T}^b)[\vec{px}] \rangle$, where p is the parent of x in T^b; observe that equivalently, $B_x = \langle \mathcal{V}_x \rangle \cap \langle \mathcal{V} \setminus \mathcal{V}_x \rangle$. With this in mind, we say that an edge uv of T is *x-degenerate* if the following linear space equality holds:

$$\langle \mathcal{L}_x(\mathcal{T})[\vec{uv}] \rangle \cap B_x = \langle \mathcal{L}_x(\mathcal{T})[\vec{vu}] \rangle \cap B_x.$$

Such an edge is *proper* x-degenerate if at least one of the following conditions holds:

- either $\vec{uv}$ or $\vec{vu}$ is x-empty; or
- there exists $y \in V(T^b)$ with $y < x$ such that: (a) there exists a y-degenerate edge in T (possibly different than uv) that is not proper, and (b) neither $\vec{uv}$ nor $\vec{vu}$ is y-empty.

Even though the definition above is recursive, it is defined correctly and uniquely – the notion of proper x-degeneracy only depends on the proper y-degeneracy of edges for $y < x$.

An x-degenerate edge that is not proper is called *improper* x-degenerate. If $\mathcal{T}$ contains an improper x-degenerate edge, we say that $\mathcal{T}$ is *x-degenerate*.

Next, an edge $\vec{uv}$ of (T, λ) is *x-guarding* (or: uv x-guards its end u) if the following strict inclusion holds:

$$\langle \mathcal{L}_x(\mathcal{T})[\vec{uv}] \rangle \cap B_x \subsetneq \langle \mathcal{L}_x(\mathcal{T})[\vec{vu}] \rangle \cap B_x.$$

In this case, $\vec{uv}$ is *improper* x-guarding if all of the following conditions hold: $\deg(u) = 3$; $\vec{uv}$ is x-mixed; and if u_1, u_2 are the two neighbors of u other than v, then neither $\vec{u_1u}$ nor $\vec{u_2u}$ is x-empty. Otherwise, $\vec{uv}$ is *proper* x-guarding.

Finally, a two-edge path uvw of (T, λ) is an *x-blocking path* if the following two equalities hold:

$$\langle \mathcal{L}_x(\mathcal{T})[\vec{uv}] \rangle \cap B_x = \langle \mathcal{L}_x(\mathcal{T})[\vec{vw}] \rangle \cap B_x =: A_1,$$
$$\langle \mathcal{L}_x(\mathcal{T})[\vec{wv}] \rangle \cap B_x = \langle \mathcal{L}_x(\mathcal{T})[\vec{vu}] \rangle \cap B_x =: A_2;$$

and moreover, neither $A_1 \subseteq A_2$ nor $A_2 \subseteq A_1$. (Note that this implies that $\dim(A_1), \dim(A_2) > 0$, so in particular, neither $\vec{uv}$ nor $\vec{wv}$ is x-empty.) In this case, uvw is an *improper* x-blocking path if $\deg(v) = 3$ and $\vec{v'v}$ is x-mixed, for the unique neighbor v' of v other than u and w. Otherwise, uvw is *proper* x-blocking.

With this bag of definitions at hand, we say that $\mathcal{T}$ is *x-pure* if one of the following holds:

- $\mathcal{T}$ is x-degenerate and x-disjoint; or
- $\mathcal{T}$ is not x-degenerate, and every x-guarding edge $\vec{uv}$ and every x-guarding path uvw is proper.

Finally, $\mathcal{T}$ is *totally pure* with respect to $\mathcal{T}^b$ if it is x-pure for all $x \in V(T^b)$.

Now, the structure theorem proven by Jeong, Kim and Oum reads as follows:

Lemma 4.8.7 ([JKO21, Proposition 4.6])**.** *Let $\mathcal{T}^b$ be a rooted rank decomposition of a subspace arrangement $\mathcal{V}$. Then there exists a rooted rank decomposition $\mathcal{T}$ of the same subspace arrangement $\mathcal{V}$ of optimum width that is totally pure with respect to $\mathcal{T}^b$.*

4.8.2 Mixed skeletons

Suppose again that $\mathcal{V}$ is a subspace arrangement, $\mathcal{T}^b = (T^b, \lambda^b)$ is a rooted rank decomposition of $\mathcal{V}$, and $\mathcal{T} = (T, \lambda)$ is a rooted rank decomposition of $\mathcal{V}$. Let $x \in V(T^b)$ be a node of T^b. We define the *x-mixed skeleton* of $\mathcal{T}$ as a (possibly empty) rooted tree T^{M} with $V(T^{\mathsf{M}}) \subseteq V(T)$ constructed as follows. For $v \in V(T)$, we put v in $V(T^{\mathsf{M}})$ if v has two children and one of the following cases holds:

- one child is x-empty and the other is x-full; or
- both children are x-mixed.

In the first case we will say that v is an *x-leaf point*, and in the second – that v is an *x-branch point*. Then two vertices $u, v \in V(T^{\mathsf{M}})$ are connected by an edge in T^{M} if the path between u and v in T is internally disjoint from $V(T^{\mathsf{M}})$ (Figure 4.2).

We will now show the correctness and the properties of this construction in a series of claims.

Lemma 4.8.8. *Suppose $v \in V(T^{\mathsf{M}})$. Then every ancestor of v (including v) is x-mixed.*

Proof. Follows from the straightforward verification with the definitions. □

It is also easily verified that a "converse" statement also holds:

Lemma 4.8.9. *Suppose $v \in V(T)$ is x-mixed. Then some descendant of v in T is an x-leaf point in $\mathcal{T}$.*

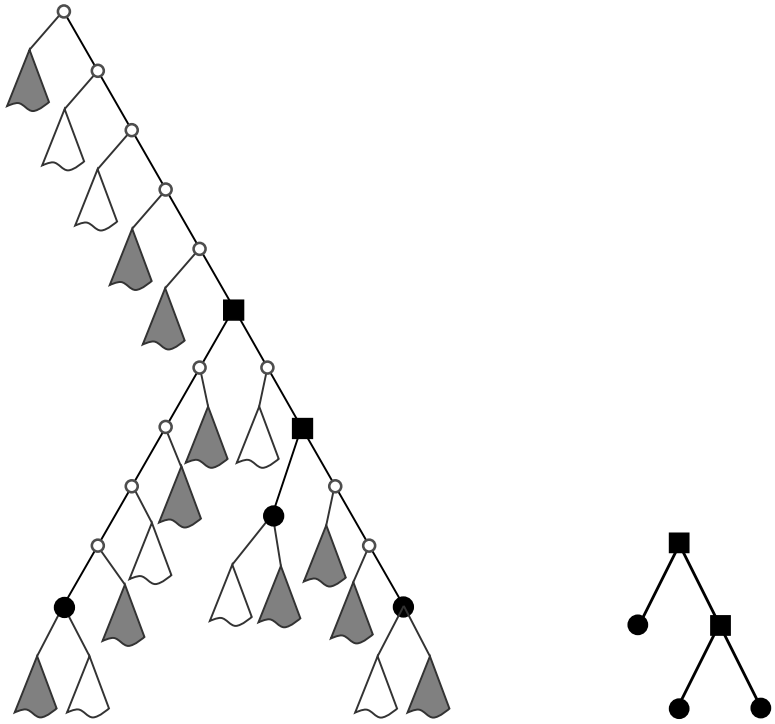
From the following lemma it follows directly that T^{M} indeed forms a rooted tree; in particular, $uv \in E(T^{\mathsf{M}})$ implies that u and v are in the ancestor-descendant relationship in T:

Lemma 4.8.10. *Suppose $u, v \in V(T^{\mathsf{M}})$. Then the lowest common ancestor of u and v belongs to T^{M}.*

Proof. Let w be the lowest common ancestor of u and v. if $w \in \{u, v\}$, then the lemma is trivial. Otherwise, let w_u and w_v be the two children of w that are ancestors of u and w, respectively. By Lemma 4.8.8, both w_u and w_v are x-mixed. Thus w is an x-branch point. □

We continue with several properties of mixed skeletons:

Lemma 4.8.11. *Suppose $p, q \in V(T^{\mathsf{M}})$ and let $uv \in E(T)$ be an edge on the path between p and q in T. Then both $\vec{uv}$ and $\vec{vu}$ are x-mixed.*



(a) (b)

Figure 4.2: (a) An example rooted rank decomposition $\mathcal{T}$. All nodes of $\mathcal{T}$ that are not x-mixed are contracted to rooted subtrees; white subtrees have x-empty roots, while the dark subtrees have x-full roots. The x-leaf points of $\mathcal{T}$ are marked by $\bullet$, while the x-branch points of $\mathcal{T}$ are marked by $\blacksquare$. The nodes that are x-mixed in $\mathcal{T}$, but neither x-leaf points nor x-branch points, are marked by $\circ$. (b) The x-mixed skeleton of $\mathcal{T}$.

Proof. Suppose not. Without loss of generality assume that: $pq \in E(T^{\mathsf{M}})$, and in particular that p is an ancestor of q in T; and that in T, q is closer to u than v. Let q_1, q_2 be the two children of q in T and p_1, p_2 be the two children of p in T; without loss of generality, assume p_1 is an ancestor of q. Note that by Lemma 4.8.8, p_1 is x-mixed; therefore, since $p \in V(T^{\mathsf{M}})$, we have that p_2 is x-mixed as well.

First suppose that $\vec{uv}$ is x-full. Then it follows immediately that both q_1 and q_2 are x-full as well (since both $\vec{q_1q}$ and $\vec{q_2q}$ are predecessors of $\vec{uv}$), contradicting that $q \in V(T^{\mathsf{M}})$. A similar contradiction follows when $\vec{uv}$ is x-empty. In the same way, observe that if $\vec{vu}$ is x-full (resp. x-empty), then p_2 is x-full (resp. x-empty) as well since $\vec{p_2p}$ is a predecessor of $\vec{vu}$. Therefore, both $\vec{uv}$ and $\vec{vu}$ must be x-mixed. □

The following lemma implies that the x-mixed skeleton is a full binary tree.

Lemma 4.8.12. *Every x-leaf point is a leaf of T^{M}, and every x-branch point is an internal node of T^{M} with two children.*

Proof. If v is an x-leaf point, then naturally every strict descendant of v in T is either x-full or x-empty. Thus by Lemma 4.8.8, no strict descendant of v is in T^{M} and therefore v is a leaf in T^{M}.

Then let v be a x-branch point. Let v_1, v_2 be the children of v in T; by definition, both v_1 and v_2 are x-mixed. By Lemma 4.8.9, there exist x-leaf points $u_1, u_2 \in V(T^{\mathsf{M}})$ that are descendants of v_1 and v_2 in T, respectively, which implies that v has at least two children in T^{M}. The lemma follows by observing from Lemma 4.8.10 that for each $i \in \{1, 2\}$, at most one vertex of $V(T^{\mathsf{M}})$ in the subtree of T rooted at v_i can be connected to v by a path internally disjoint from $V(T^{\mathsf{M}})$. □

The main product of this subsection is the following statement asserting that there exists an optimum-width rooted decomposition of $\mathcal{V}$ admitting small x-mixed skeletons for *all* $x \in V(T^b)$. In fact, any decomposition that is totally pure with respect to $\mathcal{T}^b$ fulfills the requirements of Lemma 4.8.13; we will show that through a straightforward (though careful) analysis of the definition of a totally pure decomposition.

Lemma 4.8.13. *There exists a function $f_{4.8.13} : \mathbb{N} \to \mathbb{N}$ such that the following holds. Let $\mathcal{T}^b = (T^b, \lambda^b)$ be a rooted rank decomposition of $\mathcal{V}$ of width $\ell \geqslant 0$. Then there exists a rooted rank decomposition $\mathcal{T}$ of $\mathcal{V}$ of optimum width such that, for every $x \in V(T^b)$, the x-mixed skeleton of $\mathcal{T}$ contains at most $f_{4.8.13}(\ell)$ nodes.*

Proof. Let $\mathcal{T} = (T, \lambda)$ be a rooted optimum-width rank decomposition of $\mathcal{V}$ that is totally pure with respect to $\mathcal{T}^b$; such a decomposition exists by Lemma 4.8.7. We claim that, for every $x \in V(T^b)$, the height of the x-mixed skeleton of $\mathcal{T}$ is at most $2\ell + 2$. Since mixed skeletons are rooted binary trees, the statement of the lemma will follow immediately.

Fix $x \in V(T^b)$ and let T^{M} be the x-mixed skeleton of $\mathcal{T}$. Assume for contradiction that there exists a vertical path $P = v_0v_1 \dots v_{p+1}$ in T^{M} for some $p \geqslant 2\ell + 1$, where for each $i \in [p+1]$, the node v_{i-1} is an ancestor of v_i in T. For each $i \in [p+1]$, define v_i^L as the parent of v_i in T, and for each $i \in [0, p]$, define v_i^R as the unique child of v_i in T on the simple path between v_i and v_{i+1}. For each $i \in [p]$, let v_i' be the remaining child of v_i in T. Note that for each $i \in [p]$, the node v_i is an x-branch point (Lemma 4.8.12), so the edges $v_i^{\vec{R}}v_i$ and $v_i'\vec{v_i}$ are x-mixed; moreover, the edge $v_i^{\vec{L}}v_i$ is x-mixed by Lemma 4.8.11.

Recall that $B_x = \langle \mathcal{L}_x(\mathcal{T}^b)[\vec{xp}] \rangle \cap \langle \mathcal{L}_x(\mathcal{T}^b)[\vec{px}] \rangle$, where p is the parent of x in T^b. Since $\mathcal{T}^b$ has width ℓ, by definition we necessarily have that $\dim(B_x) \leqslant \ell$. Consider the following vector spaces for each $i \in [p+1]$:

$$A_i = \langle \mathcal{L}_x(\mathcal{T})[v_i\vec{v_i^L}] \rangle \cap B_x,$$
$$B_i = \langle \mathcal{L}_x(\mathcal{T})[v_i^{\vec{L}}v_i] \rangle \cap B_x.$$

Note that $v_i^{\vec{L}}v_i$ is a predecessor of $v_{i+1}^L\vec{v_{i+1}}$ for each $i \in [p]$. Therefore we have the following chains of inclusions of vector spaces:

$$A_1 \supseteq A_2 \supseteq \dots \supseteq A_{p+1},$$
$$B_1 \subseteq B_2 \subseteq \dots \subseteq B_{p+1}.$$

Each A_i and each B_i is a vector space of dimension at most ℓ since each is a subspace of B_x. Since $p \geqslant 2\ell+1$, we find that there exists an index $t \in [p]$ such that $A_t = A_{t+1}$ and $B_t = B_{t+1}$. Because $\langle \mathcal{L}_x(\mathcal{T})[v_{t+1}\vec{v_{t+1}^L}] \rangle \subseteq$

$\langle \mathcal{L}_x(\mathcal{T})[v_t^{\vec{R}} v_t]\rangle \subseteq \langle \mathcal{L}_x(\mathcal{T})[v_t \vec{v_t^L}]\rangle$ and $\langle \mathcal{L}_x(\mathcal{T})[v_t^{\vec{L}} v_t]\rangle \subseteq \langle \mathcal{L}_x(\mathcal{T})[v_t \vec{v_t^R}]\rangle \subseteq \langle \mathcal{L}_x(\mathcal{T})[v_{t+1}^L \vec{v}_{t+1}]\rangle$, we have

$$\langle \mathcal{L}_x(\mathcal{T})[v_t \vec{v_t^L}]\rangle \cap B_x = \langle \mathcal{L}_x(\mathcal{T})[v_t^{\vec{R}} v_t]\rangle \cap B_x = A_t \quad \text{and}$$
$$\langle \mathcal{L}_x(\mathcal{T})[v_t^{\vec{L}} v_t]\rangle \cap B_x = \langle \mathcal{L}_x(\mathcal{T})[v_t \vec{v_t^R}]\rangle \cap B_x = B_t.$$

We now consider several cases with regard to the containment relation between A_t and B_t.

- If $A_t = B_t$, then the edge $e := v_t^L v_t$ is by definition x-degenerate.

  Suppose first e is improper. Then $\mathcal{T}$ is x-degenerate and so by the total purity of $\mathcal{T}$, $\mathcal{T}$ is x-pure and thus x-disjoint (i.e., either x is the root of T^b and then $\mathcal{V}_x = \mathcal{V}$, or there exists an edge $pq \in E(T)$ such that $\mathcal{L}(\mathcal{T})[\vec{pq}] = \mathcal{V}_x$). However, by Lemma 4.8.11, the edges $v_t^{\vec{L}} v_t$ and $v_t \vec{v_t^L}$ are both x-mixed. This is a contradiction as in an x-disjoint decomposition, there cannot exist an edge $uv \in E(T)$ such that both $\vec{uv}$ and $\vec{vu}$ are x-mixed.

  Now assume that e is proper. Again by Lemma 4.8.11, the edges $v_t^{\vec{L}} v_t$ and $v_t \vec{v_t^L}$ are both x-mixed. By the fact that e is proper, it must be the case that for some $y \in V(T^b)$ with $y < x$, the decomposition $\mathcal{T}$ is y-degenerate and neither $v_t^{\vec{L}} v_t$ nor $v_t \vec{v_t^L}$ is y-empty. By the total purity of $\mathcal{T}$, we have that $\mathcal{T}$ is y-disjoint. As previously, it cannot be that both $v_t^{\vec{L}} v_t$ and $v_t \vec{v_t^L}$ are y-mixed. Therefore, one of the edges $v_t^{\vec{L}} v_t, v_t \vec{v_t^L}$ is y-full. So by Observation 4.8.6, that edge is x-full, too – a contradiction.

- If $A_t \subsetneq B_t$, then the edge $e := v_t \vec{v_t^L}$ is x-guarding by definition. But recall that the three edges $v_t \vec{v_t^L}$, $v_t^{\vec{R}} v_t$ and $v_t' \vec{v}_t$ are x-mixed. Hence e is improper x-guarding by definition, which contradicts the assumption that $\mathcal{T}$ is totally pure with respect to $\mathcal{T}^b$.

- If $B_t \subsetneq A_t$, the analogous argument follows, using the x-guarding edge $v_t \vec{v_t^R}$ instead.

- If $A_t \not\subseteq B_t$ and $B_t \not\subseteq A_t$, then the path $v_t^L v_t v_t^R$ is x-blocking by definition. But since $v_t' \vec{v}_t$ is x-mixed, we get that $v_t^L v_t v_t^R$ is improperly x-blocking and thus $\mathcal{T}$ is not x-pure – a contradiction.

Since we reached a contradiction in each possible case, the proof of the lemma is complete. □

4.8.3 Statement of the Local Dealternation Lemma

The strategy of the proof of the Dealternation Lemma for subspace arrangements (Lemma 4.8.5) will be similar to that in the work of Bojańczyk and Pilipczuk [BP22]: Given as input a decomposition $\mathcal{T}^b$ of width $\ell \geqslant 0$, we first create a decomposition $\mathcal{T}$ satisfying some strong structural properties and then update $\mathcal{T}$ in a sequence of local improvement steps so as to produce the decomposition satisfying the Dealternation Lemma, preserving the structural properties throughout the process. In our case of rank decompositions of subspace arrangements, the property maintained throughout the process is precisely admitting small x-mixed skeletons for all $x \in V(T^b)$. Now we define the local improvement step in the form of the Local Dealternation Lemma.

Reusing the notation from the previous sections, assume that $x \in V(T^b)$. We say that a set $F \subseteq \mathcal{V}$ is an x-factor (resp. x-tree factor, x-context factor) in $\mathcal{T}$ if it is a factor (resp. tree factor, context factor) in $\mathcal{T}$ and moreover $F \subseteq \mathcal{V}_x$.

Lemma 4.8.14 (Local Dealternation Lemma)**.** *There exists a function $f_{4.8.14} : \mathbb{N} \to \mathbb{N}$ so that the following holds. Suppose $\mathcal{T}^b$ is a rooted rank decomposition of $\mathcal{V}$ of width $\ell \geqslant 0$, and $\mathcal{T}$ is a rooted rank decomposition of $\mathcal{V}$ of optimum width. Moreover, let $x \in V(T^b)$ be such that the x-mixed skeleton of $\mathcal{T}$ has at most $f_{4.8.13}(\ell)$ nodes. Then there exists a rooted rank decomposition $\mathcal{T}'$ of $\mathcal{V}$ of optimum width such that:*

- *the set $\mathcal{L}(\mathcal{T}^b)[x]$ is a disjoint union of at most $f_{4.8.14}(\ell)$ x-factors of $\mathcal{T}'$;*
- *for every $y \in V(T^b)$, the y-mixed skeletons of $\mathcal{T}$ and $\mathcal{T}'$ are equal; and*
- *for every $y \in V(T^b)$ with $y \not> x$, every y-factor of $\mathcal{T}$ is also a y-factor of $\mathcal{T}'$.*

We proceed to show how the "global variant" of the Dealternation Lemma for subspace arrangements (Lemma 4.8.5) follows from Lemma 4.8.14.

Proof of Lemma 4.8.5 from the Local Dealternation Lemma. Create an ordering $x_1, x_2, x_3, \ldots, x_n$ of the nodes of T^b consistent with the descendant-ancestor relationship $<$; that is, choose any ordering of the nodes in which for every pair of nodes x, y such that x is a descendant of y, x precedes y in the ordering. Throughout the proof, we will inductively create a sequence of rooted rank decompositions of $\mathcal{V}$ of optimum width: $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_n$, such that for each $t \in [0, n]$, the decomposition $\mathcal{T}_t$ satisfies the following properties:

- for every $i \in [t]$, the set $\mathcal{L}(\mathcal{T}^b)[x_i]$ is a disjoint union of at most $f_{4.8.14}(\ell)$ x_i-factors of $\mathcal{T}_t$; and
- for every $i \in [n]$, the x_i-mixed skeleton of $\mathcal{T}_t$ contains at most $f_{4.8.13}(\ell)$ nodes.

Then the decomposition $\mathcal{T}_n$ will witness the Dealternation Lemma for the subspace arrangement $\mathcal{V}$, with $f_{4.8.5} = f_{4.8.14}$.

By Lemma 4.8.13, there exists a rank decomposition $\mathcal{T}_0$ of $\mathcal{V}$ of optimum width such that for every $x \in V(T^b)$, the x-mixed skeleton of $\mathcal{T}_0$ contains at most $f_{4.8.13}(\ell)$ nodes. This verifies the inductive assumption about $\mathcal{T}_0$.

Now assume that $t \in [n]$, we are given a rank decomposition $\mathcal{T}_{t-1}$ of optimum width satisfying the inductive assumption, and we want to produce a rank decomposition $\mathcal{T}_t$. Let us apply Lemma 4.8.14 with the decomposition $\mathcal{T}_{t-1}$ and $x = x_t$, yielding the decomposition $\mathcal{T}_t$. We are left to verify that $\mathcal{T}_t$ satisfies the inductive assumptions.

First, for every $i \in [t-1]$, the set $\mathcal{L}(\mathcal{T}^b)[x_i]$ is a disjoint union of at most $f_{4.8.14}(\ell)$ x_i-factors of $\mathcal{T}_{t-1}$. Observe that $x_i \not> x_t$ by the construction of the order $x_1, \ldots, x_n$. Thus by Lemma 4.8.14, each such factor is also an x_i-factor of $\mathcal{T}_t$. Also, directly by Lemma 4.8.14 we have that $\mathcal{L}(\mathcal{T}^b)[x_t]$ is a disjoint union of at most $f_{4.8.14}(\ell)$ x_t-factors of $\mathcal{T}_t$.

Finally, let $i \in [n]$ and recall that the x_i-mixed skeleton of $\mathcal{T}_{t-1}$ contains at most $f_{4.8.13}(\ell)$ nodes. By Lemma 4.8.14, the x_i-mixed skeletons of $\mathcal{T}_t$ and $\mathcal{T}_{t-1}$ are equal, so the bound on the number of nodes applies also to the x_i-mixed skeleton of $\mathcal{T}_t$. Thus the inductive step is correct and thus the sought decomposition $\mathcal{T}_n$ exists. □

The following sections will introduce operations implementing "local rearrangements" of rank decompositions that will be used in the proof of the Local Dealternation Lemma: tree swaps and block shuffles.

4.8.4 Tree swaps

Again assume that $\mathcal{T}^b = (T^b, \lambda^b)$ and $\mathcal{T} = (T, \lambda)$ are rooted rank decompositions of $\mathcal{V}$. Let T contain a vertical path $v_0v_1v_2v_3$. We define a *swap* of $\mathcal{T}$ along the vertical path $v_0v_1v_2v_3$ as an update of the decomposition replacing the path $v_0v_1v_2v_3$ with the (vertical) path $v_0v_2v_1v_3$ (Figure 4.3). It is easy to see that after the swap, the resulting tree remains binary. Note also that swaps are invertible: Whenever the swap of $\mathcal{T}$ along $v_0v_1v_2v_3$ produces a tree $\mathcal{T}_{\mathrm{swap}}$, the original decomposition $\mathcal{T}$ is a result of a swap of $\mathcal{T}_{\mathrm{swap}}$ along $v_0v_2v_1v_3$. Finally, we say that a swap of $\mathcal{T}$ along the vertical path $v_0v_1v_2v_3$ is an *x-swap* for some $x \in V(T^b)$ if the following preconditions are met:

- v_3 is x-mixed; and
- if v_1' and v_2' are the (unique) children of v_1 and v_2, respectively, outside of the path, then exactly one of the nodes v_1', v_2' is x-empty and the other is x-full.

Observe that whenever $\mathcal{T}'$ is an x-swap of $\mathcal{T}$ along $v_0v_1v_2v_3$, then also $\mathcal{T}$ is an x-swap of $\mathcal{T}'$ along $v_0v_2v_1v_3$.

The main product of this subsection is the following lemma, asserting that for any $x \in V(T^b)$, any x-swap of $\mathcal{T}$ preserves the y-mixed skeletons of $\mathcal{T}$ for all $y \in V(T^b)$:

Lemma 4.8.15. *Let $x, y \in V(T^b)$. Suppose $\mathcal{T}_{\mathrm{swap}}$ is created from $\mathcal{T}$ by performing an x-swap along the path $v_0v_1v_2v_3$. Then the y-mixed skeletons of $\mathcal{T}$ and $\mathcal{T}_{\mathrm{swap}}$ are equal.*

The rest of this section is dedicated to the proof of Lemma 4.8.15. The proof proceeds in two steps. First, we phrase, in terms of y-emptiness, y-mixedness and y-fullness of nodes only, the structural properties of a vertical path $v_0v_1v_2v_3$ in $\mathcal{T}$ which, when fulfilled by the path, implies the perseverance of the y-mixed skeleton of $\mathcal{T}$ after the swap along $v_0v_1v_2v_3$. Then we show that whenever a swap of $\mathcal{T}$ along a path P happens to be an x-swap for any $x \in V(T^b)$, then P fulfills this structural property for every $y \in V(T^b)$; hence, such a swap will preserve all y-mixed skeletons for all $y \in V(T^b)$.

Let $v_0v_1v_2v_3$ be a vertical path in T, and v_1', v_2' be the neighbors of v_1 and v_2, respectively, outside of the path. Let also $y \in V(T^b)$. We then say that the path satisfies:

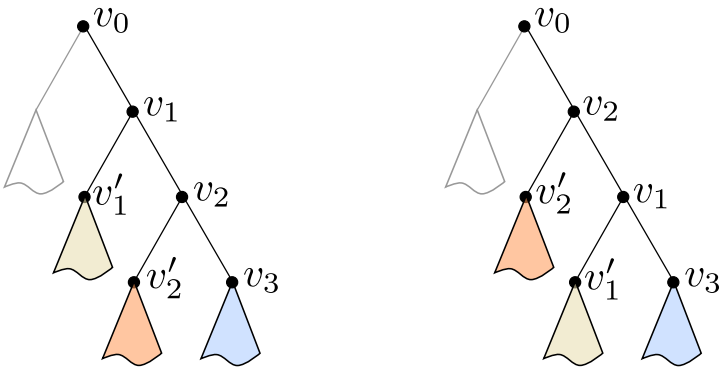


Figure 4.3: An example swap. The right decomposition is a swap of the left decomposition along $v_0v_1v_2v_3$.

- the *y-empty property* if at least one of v_1' and v_2' is y-empty, and v_3 is either y-empty or y-mixed; and
- the *y-full property* if at least one of v_1' and v_2' is y-full, and v_3 is either y-full or y-mixed.

Lemma 4.8.16. *Let $y \in V(T^b)$ and $v_0v_1v_2v_3$ be a vertical path in T satisfying either the y-empty property or the y-full property. Suppose $\mathcal{T}_{\text{swap}}$ is created from $\mathcal{T}$ by performing a swap along $v_0v_1v_2v_3$. Then the y-mixed skeletons of $\mathcal{T}$ and $\mathcal{T}_{\text{swap}}$ are equal.*

Proof. In the proof, we assume the y-empty property; the proof for the y-full property is analogous (with the roles of the y-emptiness and the y-fullness of nodes exchanged). For the course of the proof, let $\mathcal{T}_{\text{swap}} = (T_{\text{swap}}, \lambda_{\text{swap}})$, let T^{M} be a y-mixed skeleton of $\mathcal{T}$, and let $T^{\mathsf{M}}_{\text{swap}}$ be a y-mixed skeleton of $\mathcal{T}_{\text{swap}}$. Let also v_1', v_2' be the children of v_1 and v_2, respectively, outside of the path $v_0v_1v_2v_3$ in T.

Our proof crucially relies on the following helper claim:

Claim 4.8.17. *Suppose that $V(T^{\mathsf{M}}_{\text{swap}}) = V(T^{\mathsf{M}})$ and $\{v_1, v_2\} \not\subseteq V(T^{\mathsf{M}})$. Then $T^{\mathsf{M}}_{\text{swap}} = T^{\mathsf{M}}$.*

Proof of the claim. By Lemma 4.8.10, we find that $T^{\mathsf{M}}_{\text{swap}} = T^{\mathsf{M}}$ if and only if $V(T^{\mathsf{M}}_{\text{swap}}) = V(T^{\mathsf{M}})$ and the ancestor-descendant relationship is preserved on the pairs of vertices of $V(T^{\mathsf{M}})$ (i.e., $u_1 \leqslant u_2$ holds in T for some $u_1, u_2 \in V(T^{\mathsf{M}})$ if and only if $u_1 \leqslant u_2$ holds in T_{swap}).

So suppose there exist $u_1, u_2 \in V(T^{\mathsf{M}})$ such that the relation $u_1 \leqslant u_2$ holds in exactly one of the trees T, T_{swap}. By the construction of T_{swap}, one of these two vertices (say, u_1) either is equal to v_1 or is a descendant of v_1'; and the other (say, u_2) either is equal to v_2 or is a descendant of v_2'. The lowest common ancestor of u_1 and u_2 is then v_1 in T and v_2 in T_{swap}. By Lemma 4.8.10 and $V(T^{\mathsf{M}}_{\text{swap}}) = V(T^{\mathsf{M}})$, we have $\{v_1, v_2\} \subseteq V(T^{\mathsf{M}})$ – a contradiction. ◁

It is immediate that for every nonleaf node $w \notin \{v_1, v_2\}$, both subtrees rooted at the children of w in T contain the same set of nodes before and after the x-swap. Hence,

$$\{\mathcal{L}(T_{\text{swap}})[w'] \mid w' \text{ is a child of } w \text{ in } T_{\text{swap}}\} = \{\mathcal{L}(T)[w'] \mid w' \text{ is a child of } w \text{ in } T\}.$$

Thus, each $w \notin \{v_1, v_2\}$ is a y-branch point (resp. a y-leaf point) in T_{swap} if and only if w is a y-branch point (resp. a y-leaf point) in T. Moreover, it is easy to see that for each $w \notin \{v_1, v_2\}$, w is y-empty (resp. y-mixed, y-full) in T if and only if w is y-empty (resp. y-mixed, y-full) in T_{swap}.

Therefore, by Claim 4.8.17, for the equality of the y-mixed skeletons of $\mathcal{T}$ and $\mathcal{T}_{\text{swap}}$ it is enough to prove that:

- for each $w \in \{v_1, v_2\}$, $w \in V(T^{\mathsf{M}}_{\text{swap}})$ if and only if $w \in V(T^{\mathsf{M}})$; and
- v_1, v_2 do not both belong to the y-mixed skeleton of $\mathcal{T}$.

These conditions will follow immediately from the following series of claims.

Claim 4.8.18. *Suppose $v_1 \in V(T^{\mathsf{M}})$. Then $v_1 \in V(T^{\mathsf{M}}_{\text{swap}})$.*

Proof of the claim. If v_1' is y-empty in $\mathcal{T}$, then v_2 must be y-full in $\mathcal{T}$ (otherwise we would have $v_1 \notin V(T^{\mathsf{M}})$); but this contradicts the assumption that v_3 is y-empty or y-mixed. Therefore, it is v_2' that is y-empty in $\mathcal{T}$. We now consider cases depending on the type of v_3 in $\mathcal{T}$:

- If v_3 is y-empty in $\mathcal{T}$, then it follows that v_2 is y-empty in $\mathcal{T}$. Since $v_1 \in V(T^{\mathsf{M}})$, we infer that v_1' is y-full in $\mathcal{T}$ and v_1 is a y-leaf point in $\mathcal{T}$. Then, in $\mathcal{T}_{\text{swap}}$, the two children of v_1 (that is, v_1' and v_3) are y-full and y-empty, respectively. Thus v_1 is also a y-leaf point in $\mathcal{T}_{\text{swap}}$.
- If v_3 is y-mixed in $\mathcal{T}$, then so is v_2. Since $v_1 \in V(T^{\mathsf{M}})$, it must be the case that v_1' is also y-mixed in $\mathcal{T}$ and v_1 is a y-branch point in $\mathcal{T}$. Hence in $\mathcal{T}_{\text{swap}}$, both children of v_1 (again, v_1' and v_3) are y-mixed, witnessing that v_1 is a y-branch point also in $\mathcal{T}_{\text{swap}}$. ◁

Claim 4.8.19. *Suppose $v_2 \in V(T^{\mathsf{M}})$. Then $v_2 \in V(T^{\mathsf{M}}_{\text{swap}})$.*

Proof of the claim. If v_2' is y-empty in $\mathcal{T}$, then v_3 must be y-full in $\mathcal{T}$ (otherwise $v_2 \notin V(T^{\mathsf{M}})$) – a contradiction with the y-empty property of $v_0v_1v_2v_3$ in $\mathcal{T}$. So it is v_1' that is y-empty in $\mathcal{T}$. Again, consider cases depending on the type of v_3 in T:

- If v_3 is y-empty in $\mathcal{T}$, then $v_2 \in V(T^{\mathsf{M}})$ implies that v_2' is y-full in $\mathcal{T}$. Then, in $\mathcal{T}_{\text{swap}}$, v_1 is y-empty (since both children v_1', v_3 are y-empty) and so $v_2 \in V(T^{\mathsf{M}}_{\text{swap}})$ (since one child v_2' is y-full and the other child v_1 is y-empty).
- If v_3 is y-mixed in $\mathcal{T}$, then $v_2 \in V(T^{\mathsf{M}})$ implies that v_2' is also y-mixed in $\mathcal{T}$. Hence, in $\mathcal{T}_{\text{swap}}$, v_1 is y-mixed (since a child v_3 is y-mixed), and so $v_2 \in V(T^{\mathsf{M}}_{\text{swap}})$ (since both children v_2', v_1 are y-mixed). ◁

Claim 4.8.20. *If $v_1 \in V(T^{\mathsf{M}}_{\text{swap}})$, then $v_1 \in V(T^{\mathsf{M}})$. Similarly, if $v_2 \in V(T^{\mathsf{M}}_{\text{swap}})$, then $v_2 \in V(T^{\mathsf{M}})$.*

Proof of the claim. Observe that the vertical path $v_0v_2v_1v_3$ satisfies the y-empty property in $\mathcal{T}_{\text{swap}}$; moreover, the swap of $\mathcal{T}_{\text{swap}}$ along this path produces the original decomposition $\mathcal{T}$. Thus, by Claim 4.8.18, $v_2 \in V(T^{\mathsf{M}}_{\text{swap}})$ implies that $v_2 \in V(T^{\mathsf{M}})$. Similarly, by Claim 4.8.19, $v_1 \in V(T^{\mathsf{M}}_{\text{swap}})$ implies $v_1 \in V(T^{\mathsf{M}})$. ◁

Claim 4.8.21. *It cannot happen that $v_1, v_2 \in V(T^{\mathsf{M}})$.*

Proof of the claim. If v_3 is y-empty in $\mathcal{T}$, then v_2' must be y-full (otherwise $v_2 \notin V(T^{\mathsf{M}})$), and so v_2 must be y-mixed. But then from the y-empty property of $v_0v_1v_2v_3$, the node v_1' must be y-empty and thus $v_1 \notin V(T^{\mathsf{M}})$ – a contradiction.

If v_3 is y-mixed in $\mathcal{T}$, then so is v_2' (or else $v_2 \notin V(T^{\mathsf{M}})$), and v_2 is y-mixed, too. But then again, v_1' must be y-empty from the y-empty property of $v_0v_1v_2v_3$, which contradicts that $v_1 \in V(T^{\mathsf{M}})$. ◁

Claims 4.8.18 to 4.8.21 conclude the proof of the lemma. □

We are now ready to give a proof of Lemma 4.8.15.

Proof of Lemma 4.8.15. We only show the proof in the case where v_1' is x-empty and v_2' is x-full in $\mathcal{T}$; the proof for the symmetric case is analogous. Recall that v_3 is x-mixed in $\mathcal{T}$. We consider three cases, depending on how x and y are related with respect to the ancestor-descendant relationship in T^b.

Case 1: $y \geqslant x$ (i.e., y is an ancestor of x in T^b). Then by Observation 4.8.6, we have that v_2' is y-full in $\mathcal{T}$; and v_3 is y-mixed or y-full. So $v_0v_1v_2v_3$ satisfies the y-full property, hence Lemma 4.8.16 applies.

Case 2: $y \leqslant x$ (i.e., y is a descendant of x in T^b). Then by Observation 4.8.6, we have that in $\mathcal{T}$, v_1' is y-empty and v_3 is y-empty or y-mixed. Therefore, $v_0v_1v_2v_3$ satisfies the y-empty property and Lemma 4.8.16 applies.

Case 3: y is not in the ancestor-descendant relationship with x in T^b. Again by Observation 4.8.6, we have that in $\mathcal{T}$, v_2' is y-empty and v_3 is y-empty or y-mixed. Hence we can apply Lemma 4.8.16 as the path $v_0v_1v_2v_3$ satisfies the y-empty property. □

4.8.5 Block shuffles

While the operation of swaps is quite strong in the sense that any x-swap preserves the y-mixed skeleton for any $x, y \in V(T^b)$, this unfortunately is not the case for y-factors: It could happen that a y-factor of $\mathcal{T}$ could cease to exist after performing an x-swap. We will resolve this issue by introducing a more structured counterpart of a swap: a (boundary-preserving) block-shuffle.

Suppose that T contains a long vertical path $v_0v_1 \dots v_pv_{p+1}$, $p \geqslant 0$. For each $i \in [p]$, let v_i' be the (unique) child of v_i not on the path. Let also $x \in V(T^b)$ and consider the case that for each $i \in [p]$, the vertex v_i' is either x-empty or x-full in $\mathcal{T}$; and that v_{p+1} is x-mixed in $\mathcal{T}$. (This is equivalently the case

where the x-mixed skeleton of $\mathcal{T}$ contains a vertex in the subtree rooted at v_{p+1}, but none of the vertices $v_1, \ldots, v_p$ are vertices of this skeleton.) Any such path will be called *x-shuffleable* from now on.

Now we say that an integer interval $I = [\ell, r] \subseteq [1, p]$ is an *x-empty block* if all the vertices v'_i for $i \in I$ are x-empty, and the interval cannot be extended from either side so as to preserve this property. We similarly define *x-full blocks*. Then an *x-block* is either an x-empty block or an x-full block. Naturally, x-blocks form a partition of $[1, p]$ into intervals, and in this partition, x-empty blocks and x-full blocks alternate. In the following description, we will sometimes identify x-blocks $[\ell, r]$ with the sequences of vertices $(v'_\ell, \ldots, v'_r)$ and $(v_\ell, \ldots, v_r)$.

For a permutation σ of $\{1, 2, \ldots, p\}$, we say that the replacement of the vertical path $v_0 v_1 \ldots v_p v_{p+1}$ with the path $v_0 v_{\sigma(1)} v_{\sigma(2)} \ldots v_{\sigma(p)} v_{p+1}$ is an *x-block shuffle along $v_0 \ldots v_{p+1}$ using σ* if all the following conditions hold:

- If i and $i+1$ belong to the same x-block, then $\sigma^{-1}(i+1) = \sigma^{-1}(i) + 1$ (i.e., the value $i+1$ appears in the permutation immediately after i); and
- If $1 \leqslant i < j \leqslant p$ and both v'_i, v'_j are x-empty (or both are x-full), then $\sigma^{-1}(i) < \sigma^{-1}(j)$ (i.e., the value j appears in the permutation later than i).

For convenience, we say that the permutation σ is the *recipe* of the block shuffle.

Intuitively, an x-block shuffle can be pictured as an arbitrary shuffle of vertices along the vertical path that preserves the x-blocks of vertices along the path and never swaps two x-blocks of the same kind. For our convenience, we extend σ to be a permutation of $\{0, \ldots, p+1\}$ by setting $\sigma(0) = 0$ and $\sigma(p+1) = p+1$. If additionally it holds that $\sigma(1) = 1$ and $\sigma(p) = p$, then we say that an x-block shuffle is *boundary-preserving*; equivalently, the first and the last x-blocks are preserved intact by the shuffle (Figure 4.4).

The following fact is straightforward.

Lemma 4.8.22. *An x-block shuffle of a rank decomposition is equivalent to a composition of x-swaps. In other words, if $\mathcal{T}'$ is a result of an x-block shuffle along a vertical path of $\mathcal{T}$, then $\mathcal{T}'$ can also be produced from $\mathcal{T}$ by applying a sequence of x-swaps.*

Together with Lemma 4.8.15, this immediately implies the following:

Lemma 4.8.23. *Let $x, y \in V(T^b)$. Suppose $\mathcal{T}'$ is created from $\mathcal{T}$ by performing an x-block shuffle along a vertical path. Then the y-mixed skeletons of $\mathcal{T}$ and $\mathcal{T}'$ are equal.*

However, the structure introduced to x-block shuffles atop the x-swaps now allows us to reason about the perseverance of y-factors in the modified rank decomposition:

Lemma 4.8.24. *Let $x, y \in V(T^b)$ with $y \not> x$. Suppose $\mathcal{T}'$ is created from $\mathcal{T}$ by performing a boundary-preserving x-block shuffle along $v_0 \ldots v_{p+1}$. Then every y-factor of $\mathcal{T}$ is also a y-factor of $\mathcal{T}'$.*

Proof. Assume that $\mathcal{T}' \neq \mathcal{T}$, i.e., the performed block shuffle was nontrivial. Then $v_0 v_1 \ldots v_{p+1}$ contains at least four x-blocks; let $[\ell_1, r_1], [\ell_2, r_2], \ldots, [\ell_t, r_t]$ be the partition of $[1, p]$ into x-blocks, with $1 = \ell_1 \leqslant r_1 < \ell_2 \leqslant r_2 < \cdots < \ell_t \leqslant r_t = p$ and $\ell_{i+1} = r_i + 1$ for all $i \in [p-1]$. Let σ be the recipe of the block shuffle. Since the block shuffle is boundary-preserving, we have $\sigma(i) = i$ for $i \leqslant r_1$ and $i \geqslant \ell_t$. Note that by the construction, $\mathcal{L}(\mathcal{T})[v] = \mathcal{L}(\mathcal{T}')[v]$ for every $v \in V(\mathcal{T}) \setminus \{v_{\ell_2}, v_{\ell_2+1}, \ldots, v_{r_{t-1}}\}$. Moreover, $\mathcal{L}(\mathcal{T})[v_{\ell_2}] = \mathcal{L}(\mathcal{T}')[v_{\sigma(\ell_2)}]$.

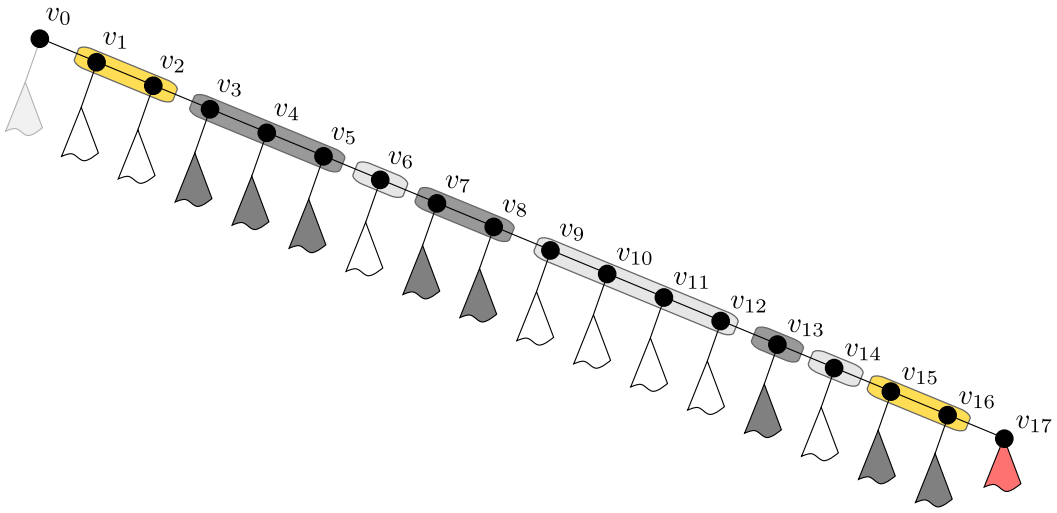
Let $F \subseteq \mathcal{V}_y$ be a y-factor of $\mathcal{T}$. The following claim captures the essential property of y-factors for $y \not> x$ that will be used in the current proof.

Claim 4.8.25. *$F \subseteq \mathcal{V}_x$ or F is disjoint from $\mathcal{V}_x$.*

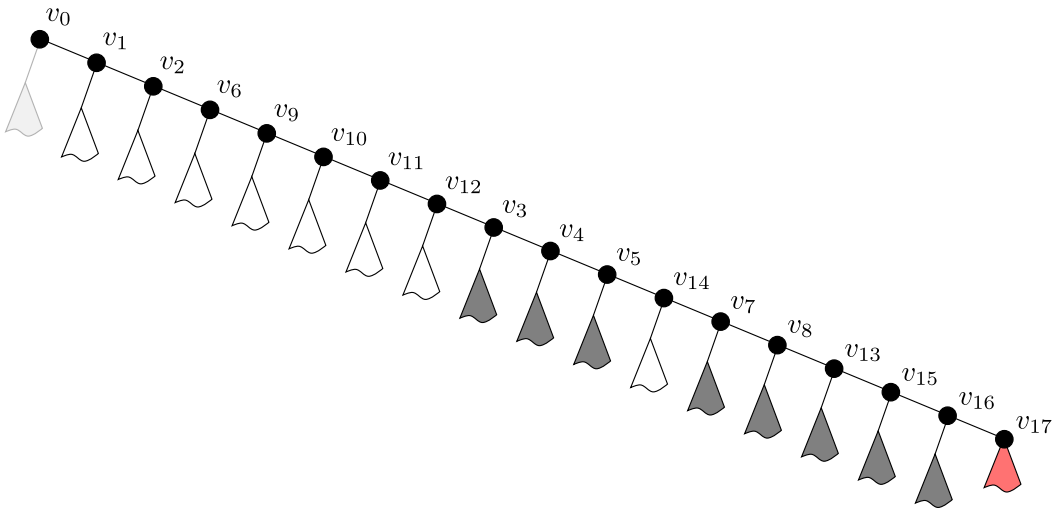
Proof of the claim. If $y \leqslant x$, then $\mathcal{V}_y \subseteq \mathcal{V}_x$ and thus $F \subseteq \mathcal{V}_x$. On the other hand, if y is incomparable with x with respect to the ancestor-descendant relationship in T^b, then $\mathcal{V}_y$ is disjoint from $\mathcal{V}_x$, so also F is disjoint from $\mathcal{V}_x$. ◁

First suppose that F is a y-tree factor, i.e., $F = \mathcal{L}(\mathcal{T})[w]$ for some $w \in V(T)$. Note that if w is an ancestor of $v_{r_{t-1}}$, then w is also an ancestor of both $v'_{r_{t-1}}$ and v'_{ℓ_t}. But exactly one of the vertices $v'_{r_{t-1}}, v'_{\ell_t}$ is x-empty and the other is x-full. In other words, we have $\mathcal{L}(\mathcal{T})[v'_{r_{t-1}}] \cup \mathcal{L}(\mathcal{T})[v'_{\ell_t}] \subseteq F$, but exactly one of the sets $\mathcal{L}(\mathcal{T})[v'_{r_{t-1}}]$, $\mathcal{L}(\mathcal{T})[v'_{\ell_t}]$ is a subset of $\mathcal{V}_x$ and the other is disjoint from $\mathcal{V}_x$. This, however, contradicts Claim 4.8.25. Hence, w is not an ancestor of $v_{r_{t-1}}$. But then $w \notin \{v_{\ell_2}, v_{\ell_2+1}, \ldots, v_{r_{t-1}}\}$, so $\mathcal{L}(\mathcal{T})[w] = \mathcal{L}(\mathcal{T}')[w]$ and thus F is also a y-factor of $\mathcal{T}'$.

Now consider the case where F is a y-context factor in $\mathcal{T}$, that is, $F = \mathcal{L}(\mathcal{T})[w_1] \setminus \mathcal{L}(\mathcal{T})[w_2]$ and w_1 is a strict ancestor of w_2 in T.



(a)



(b)

Figure 4.4: (a) A sample x-shuffleable path. White subtrees have x-empty roots and dark subtrees have x-full roots; the root v_{17} of the red subtree is x-mixed. The blocks of the path are indicated by boxes; the two boundary blocks are colored yellow.
(b) An example boundary-preserving x-block shuffle of the path. The recipe of the block shuffle is $\sigma = (1, 2, 6, 9, 10, 11, 12, 3, 4, 5, 14, 7, 8, 13, 15, 16)$.

Claim 4.8.26. *It cannot happen that, for some $i \in [t-1]$, w_1 is an ancestor of v_{r_i} and w_2 is not an ancestor of $v_{\ell_{i+1}} = v_{r_i+1}$.*

Proof of the claim. Proof by contradiction. First suppose that w_2 is not in the ancestor-descendant relationship with v_{r_i+2} in T. Since $r_i < \ell_{i+1} \leqslant p$, we get that $\mathcal{L}(\mathcal{T})[v_{p+1}]$ is disjoint from $\mathcal{L}(\mathcal{T})[w_2]$ and thus $\mathcal{L}(\mathcal{T})[v_{p+1}] \subseteq F$. But v_{p+1} is x-mixed in $\mathcal{T}$, so $\mathcal{L}(\mathcal{T})[v_{p+1}]$ is neither a subset of $\mathcal{V}_x$ nor disjoint from $\mathcal{V}_x$. Hence contradiction with Claim 4.8.25.

Since w_2 is not an ancestor of v_{r_i+1}, it means that w_2 is a descendant of v_{r_i+2} and so $\mathcal{L}(\mathcal{T})[v'_{r_i}] \cup \mathcal{L}(\mathcal{T})[v'_{r_i+1}] \subseteq F$. However exactly one of $\mathcal{L}(\mathcal{T})[v'_{r_i}]$ and $\mathcal{L}(\mathcal{T})[v'_{r_i+1}] = \mathcal{L}(\mathcal{T})[v'_{\ell_{i+1}}]$ is x-empty in T and the other is x-full in T. So again $\mathcal{L}(\mathcal{T})[v'_{r_i}] \cup \mathcal{L}(\mathcal{T})[v'_{r_i+1}]$ is neither a subset of $\mathcal{V}_x$ nor disjoint from $\mathcal{V}_x$ – a contradiction. ◁

If $w_1, w_2 \notin \{v_{\ell_2}, v_{\ell_2+1}, \ldots, v_{r_t-1}\}$, then $\mathcal{L}(\mathcal{T})[w_1] = \mathcal{L}(\mathcal{T}')[w_1]$ and $\mathcal{L}(\mathcal{T})[w_2] = \mathcal{L}(\mathcal{T}')[w_2]$, so F is also a y-context factor in $\mathcal{T}'$. Now suppose that at least one of w_1, w_2 is in $\{v_{\ell_2}, v_{\ell_2+1}, \ldots, v_{r_t-1}\}$. Since w_1 is a (strict) ancestor of w_2, we must have that w_1 is also an ancestor of $v_{r_{t-1}}$ and w_2 is a descendant of v_{ℓ_2}. Let then $j \in [t-1]$ be the smallest positive integer such that w_1 is an ancestor of v_{r_j}. So by Claim 4.8.26, w_2 is an ancestor of $v_{\ell_{j+1}}$. If $j = 1$, then $w_2 = v_{\ell_2}$ and $w_1 \notin \{v_{\ell_2}, v_{\ell_2+1}, \ldots, v_{r_t-1}\}$. Hence $F = \mathcal{L}(\mathcal{T})[w_1] \setminus \mathcal{L}(\mathcal{T})[v_{\ell_2}] = \mathcal{L}(\mathcal{T}')[w_1] \setminus \mathcal{L}(\mathcal{T}')[v_{\sigma(\ell_2)}]$ and F is a y-context factor in $\mathcal{T}'$. On the other hand, assume $j \geqslant 2$. In this case, w_2 is an ancestor of $v_{\ell_{j+1}}$ and w_1 is an ancestor of w_2, but a descendant of v_{ℓ_j} (by the definition of j). Let i_1, i_2 (with $\ell_j \leqslant i_1 < i_2 \leqslant \ell_{j+1}$) be such that $w_1 = v_{i_1}$ and $w_2 = v_{i_2}$. Then, $F = \bigcup_{i=i_1}^{i_2-1} \mathcal{L}(\mathcal{T})[v'_i]$. Since $[i_1, i_2 - 1]$ is a part of an x-block of the path $v_0v_1 \ldots v_{p+1}$, there exists some $q \in \mathbb{N}$ such that $\sigma(q+i) = i_1 + i$ for all $i \in [0, i_2 - i_1 - 1]$. We conclude that $F = \bigcup_{i=0}^{i_2-i_1-1} \mathcal{L}(\mathcal{T}')[v'_{\sigma(q+i)}] = \mathcal{L}(\mathcal{T}')[v_{\sigma(q)}] \setminus \mathcal{L}(\mathcal{T}')[v_{\sigma(q+i_2-i_1)}]$. Hence also in this case, F is a y-context factor of T'. As all cases have been exhausted, this finishes the proof. □

Observe that an x-block shuffle will never increase the number of x-blocks along the shuffled path; on the other hand, the number of such x-blocks might decrease significantly if many x-blocks of the same kind are placed one after another. We will now prove that it is indeed possible to perform such a shuffle so as to decrease the number of x-blocks to a constant (depending only on the width of $\mathcal{T}^b$) *without* increasing the width of $\mathcal{T}$:

Lemma 4.8.27. *There exists a function $f_{4.8.27} : \mathbb{N} \to \mathbb{N}$ such that the following holds. Assume that the width of $\mathcal{T}$ and $\mathcal{T}^b$ is bounded by $\ell \geqslant 0$ and let $x \in V(T^b)$. Suppose $v_0v_1 \ldots v_{p+1}$ is an x-shuffleable path in $\mathcal{T}$. Then there exists a boundary-preserving x-block shuffle of the path using a permutation σ such that:*

- *the decomposition $\mathcal{T}'$ after the shuffle has width not greater than the width of $\mathcal{T}$; and*
- *in $\mathcal{T}'$, the vertical path $v_{\sigma(1)} \ldots v_{\sigma(p)}$ contains at most $f_{4.8.27}(\ell)$ x-blocks.*

In the remaining part of this section we will cover the proof of Lemma 4.8.27. We will call an x-shuffleable vertical path $v_0v_1 \ldots v_{p+1}$:

- *x-static* if all of the following subspace equalities hold:

$$\begin{aligned}
\langle \mathcal{L}_x(T)[v_1\vec{v}_0] \rangle \cap B_x &= \langle \mathcal{L}_x(T)[v_{p+1}\vec{v}_p] \rangle \cap B_x, \\
\langle \mathcal{L}_{\bar{x}}(T)[v_1\vec{v}_0] \rangle \cap B_x &= \langle \mathcal{L}_{\bar{x}}(T)[v_{p+1}\vec{v}_p] \rangle \cap B_x, \\
\langle \mathcal{L}_x(T)[v_0\vec{v}_1] \rangle \cap B_x &= \langle \mathcal{L}_x(T)[v_p\vec{v}_{p+1}] \rangle \cap B_x, \\
\langle \mathcal{L}_{\bar{x}}(T)[v_0\vec{v}_1] \rangle \cap B_x &= \langle \mathcal{L}_{\bar{x}}(T)[v_p\vec{v}_{p+1}] \rangle \cap B_x;
\end{aligned}$$

- *x-separable* if there exist integers $c_0, c_1, \ldots, c_p \in \mathbb{Z}$ such that the following holds. Suppose $\mathcal{T}'$ is formed from $\mathcal{T}$ by performing a boundary-preserving x-block shuffle along $v_0v_1 \ldots v_{p+1}$ using σ. Then, for every $i \in [0, p]$, the width of the edge $v_{\sigma(i)}v_{\sigma(i+1)}$ in $\mathcal{T}'$ is equal to $c_{\sigma(0)} + c_{\sigma(1)} + \ldots + c_{\sigma(i)}$.

The following lemma relates these notions:

Lemma 4.8.28. *Every x-static path is x-separable.*

Proof. Let $v_0v_1 \ldots v_{p+1}$ be an x-static path, and for $i \in [p]$, let v'_i be the unique child of v_i outside of the path. Let us partition the sequence of nodes $v'_1, v'_2, \ldots, v'_p$ into those that are x-full and those that are x-empty. Formally, let q be the number of x-full nodes among $\{v'_1, \ldots, v'_p\}$ and let $1 \leqslant a_1^+ < a_2^+ < \cdots < a_q^+ \leqslant p$ denote the sequence of indices of x-full nodes $v'_{a_1^+}, \ldots, v'_{a_q^+}$. Similarly define r as the number of x-empty

nodes among $\{v'_1, \dots, v'_r\}$ and let $1 \leqslant a_1^- < a_2^- < \dots < a_r^- \leqslant p$ denote the complementary sequence of indices of x-empty nodes $v'_{a_1^-}, \dots, v'_{a_r^-}$.

Recall that x-block shuffles do not exchange the order of x-full nodes or the order of x-empty nodes; that is, in every decomposition formed by an x-block shuffle, the order of the nodes $v_{a_1^+}, \dots, v_{a_q^+}$ along the shuffled path is preserved, and so is the order of the nodes $v_{a_1^-}, \dots, v_{a_r^-}$. Therefore, if we assume that a rank decomposition $\mathcal{T}'$ is formed by performing an x-block shuffle using a permutation σ on $\mathcal{T}$, then for any $i \in [0, p]$, the sets $\mathcal{L}(\mathcal{T}')[v_{\sigma(i)}\vec{v}_{\sigma(i+1)}], \mathcal{L}(\mathcal{T}')[v_{\sigma(i+1)}\vec{v}_{\sigma(i)}]$ of vector spaces on either side of the edge of the edge $v_{\sigma(i)}v_{\sigma(i+1)}$ only depend on:

- the number $i^+ \in [0, q]$ of x-full nodes in the prefix $v'_{\sigma(1)}, \dots, v'_{\sigma(i)}$; and
- the number $i^- = i - i^+ \in [0, r]$ of x-empty nodes in the prefix $v'_{\sigma(1)}, \dots, v'_{\sigma(i)}$.

Note that $\{v'_{\sigma(1)}, v'_{\sigma(2)}, \dots, v'_{\sigma(i)}\} = \{v'_{a_1^+}, \dots, v_{a_{i^+}^+}, v'_{a_1^-}, \dots, v_{a_{i^-}^-}\}$. Next, define the following vector spaces:

$$\begin{aligned}
X_L &= \langle \mathcal{L}_x(\mathcal{T})[v_0\vec{v}_1]\rangle, & Y_L &= \langle \mathcal{L}_{\bar{x}}(\mathcal{T})[v_0\vec{v}_1]\rangle,\\
X_R &= \langle \mathcal{L}_x(\mathcal{T})[v_{p+1}\vec{v}_p]\rangle, & Y_R &= \langle \mathcal{L}_{\bar{x}}(\mathcal{T})[v_{p+1}\vec{v}_p]\rangle,\\
X_i &= \langle \mathcal{L}_x(\mathcal{T})[v'_{a_i^+}\vec{v}_{a_i^+}]\rangle = \langle \mathcal{L}(\mathcal{T})[v'_{a_i^+}\vec{v}_{a_i^+}]\rangle, & Y_j &= \langle \mathcal{L}_{\bar{x}}(\mathcal{T})[v'_{a_j^-}\vec{v}_{a_j^-}]\rangle = \langle \mathcal{L}(\mathcal{T})[v'_{a_j^-}\vec{v}_{a_j^-}]\rangle,
\end{aligned}$$

where $i \in [q]$ and $j \in [r]$, and

$$\begin{aligned}
X_{\leqslant i} &= X_L + X_1 + \ldots + X_i, & Y_{\leqslant j} &= Y_L + Y_1 + \ldots + Y_j,\\
X_{>i} &= X_{i+1} + \ldots + X_q + X_R, & Y_{>j} &= Y_{j+1} + \ldots + Y_r + Y_R,
\end{aligned}$$

where $i \in [0, q]$ and $j \in [0, r]$. Then

$$\begin{aligned}
\langle \mathcal{L}(T')[v_{\sigma(i)}\vec{v}_{\sigma(i+1)}]\rangle &= X_{\leqslant i^+} + Y_{\leqslant i^-},\\
\langle \mathcal{L}(T')[v_{\sigma(i+1)}\vec{v}_{\sigma(i)}]\rangle &= X_{>i^+} + Y_{>i^-}.
\end{aligned}$$

Moreover, the property of the path being x-static can be equivalently restated as follows:

$$\begin{aligned}
X_L \cap B_x &= X_{\leqslant q} \cap B_x, & Y_L \cap B_x &= Y_{\leqslant r} \cap B_x,\\
X_R \cap B_x &= X_{>0} \cap B_x, & Y_R \cap B_x &= Y_{>0} \cap B_x.
\end{aligned}$$

Note also that $B_x = \langle \mathcal{V}_x\rangle \cap \langle \mathcal{V} \setminus \mathcal{V}_x\rangle = (X_L + X_1 + \ldots + X_q + X_R) \cap (Y_L + Y_1 + \ldots + Y_r + Y_R)$.

We are interested in the width of the edge $v_{\sigma(i)}v_{\sigma(i+1)}$, that is, the dimension d_i of the subspace $\mathcal{L}(\mathcal{T}')[v_{\sigma(i)}\vec{v}_{\sigma(i+1)}] \cap \mathcal{L}(\mathcal{T}')[v_{\sigma(i+1)}\vec{v}_{\sigma(i)}] = (X_{\leqslant i^+} + Y_{\leqslant i^-}) \cap (X_{>i^+} + Y_{>i^-})$. Applying Lemma 4.8.2 with $U_1 = X_{\leqslant i^+}$, $U_2 = Y_{\leqslant i^-}$, $V_1 = X_{>i^+}$, $V_2 = Y_{>i^-}$, we find that

$$\begin{aligned}
&\dim((X_{\leqslant i^+} + Y_{\leqslant i^-}) \cap (X_{>i^+} + Y_{>i^-})) + \dim(X_{\leqslant i^+} \cap Y_{\leqslant i^-}) + \dim(X_{>i^+} \cap Y_{>i^-}) =\\
&\quad = \dim((X_{\leqslant i^+} + X_{>i^+}) \cap (Y_{\leqslant i^-} + Y_{>i^-})) + \dim(X_{\leqslant i^+} \cap X_{>i^+}) + \dim(Y_{\leqslant i^-} \cap Y_{>i^-}).
\end{aligned} \tag{4.8}$$

Since $X_{\leqslant i^+} + X_{>i^+} = \langle \mathcal{V}_x\rangle$ and $Y_{\leqslant i^-} + Y_{>i^-} = \langle \mathcal{V} \setminus \mathcal{V}_x\rangle$, we have by definition

$$(X_{\leqslant i^+} + X_{>i^+}) \cap (Y_{\leqslant i^-} + Y_{>i^-}) = B_x. \tag{4.9}$$

Now, $X_{\leqslant i^+} \subseteq \langle \mathcal{V}_x\rangle$ and $Y_{\leqslant i^-} \subseteq \langle \mathcal{V} \setminus \mathcal{V}_x\rangle$; since $\langle \mathcal{V}_x\rangle \cap \langle \mathcal{V} \setminus \mathcal{V}_x\rangle = B_x$, we see that $X_{\leqslant i^+} \cap Y_{\leqslant i^-} \subseteq B_x$. Therefore,

$$X_{\leqslant i^+} \cap Y_{\leqslant i^-} = (X_{\leqslant i^+} \cap B_x) \cap (Y_{\leqslant i^-} \cap B_x).$$

But now, using the fact that the path $v_0v_1 \dots v_{p+1}$ is x-static, we have

$$X_L \cap B_x \subseteq X_{\leqslant i^+} \cap B_x \subseteq X_{\leqslant q} \cap B_x = X_L \cap B_x,$$

so $X_{\leqslant i^+} \cap B_x = X_L \cap B_x$; similarly, we compute that $Y_{\leqslant i^+} \cap B_x = Y_L \cap B_x$. Hence,

$$X_{\leqslant i^+} \cap Y_{\leqslant i^-} = (X_L \cap B_x) \cap (Y_L \cap B_x) = X_L \cap Y_L \cap B_x = X_L \cap Y_L, \tag{4.10}$$

since once again, $X_L \cap Y_L \subseteq B_x$. By an analogous argument, we also deduce that

$$X_{>i^+} \cap Y_{>i^-} = X_R \cap Y_R. \tag{4.11}$$

Plugging in Eqs. (4.9) to (4.11) into Eq. (4.8), we conclude that

$$\begin{aligned} d_i &= \dim((X_{\leqslant i^+} + Y_{\leqslant i^-}) \cap (X_{>i^+} + Y_{>i^-})) = \\ &= [\dim(B_x) - \dim(X_L \cap Y_L) - \dim(X_R \cap Y_R)] + \dim(X_{\leqslant i^+} \cap X_{>i^+}) + \dim(Y_{\leqslant i^-} \cap Y_{>i^-}). \end{aligned}$$

That is, setting $\alpha = \dim(B_x) - \dim(X_L \cap Y_L) - \dim(X_R \cap Y_R)$ (a constant independent on i and σ), $\beta_{i^+} = \dim(X_{\leqslant i^+} \cap X_{>i^+})$ (a constant dependent only on i^+, but not on i or σ), and $\gamma_{i^-} = \dim(Y_{\leqslant i^-} \cap Y_{>i^-})$ (a constant dependent only on i^- and not on i or σ), we have that

$$d_i = \alpha + \beta_{i^+} + \gamma_{i^-}.$$

Now, set

$$\begin{aligned} c_0 &= \alpha + \beta_0 + \gamma_0, \\ c_{a_i^+} &= \beta_i - \beta_{i-1} && \text{for } i \in [q], \\ c_{a_j^-} &= \gamma_j - \gamma_{j-1} && \text{for } j \in [r]. \end{aligned}$$

It is now easy to verify that for every $i \in [0, p]$, the width of the edge $v_{\sigma(i)}v_{\sigma(i+1)}$ in $\mathcal{T}'$ is

$$\begin{aligned} \dim(\mathcal{L}(\mathcal{T}')[v_{\sigma(i)}\vec{v_{\sigma(i+1)}}] \cap \mathcal{L}(\mathcal{T}')[v_{\sigma(i+1)}\vec{v_{\sigma(i)}}]) &= \\ &= \dim((X_{\leqslant i^+} + Y_{\leqslant i^-}) \cap (X_{>i^+} + Y_{>i^-})) = \\ &= \alpha + \beta_{i^+} + \gamma_{i^-} = \\ &= c_0 + (c_{a_1^+} + c_{a_2^+} + \ldots + c_{a_{i^+}^+}) + (c_{a_1^-} + c_{a_2^-} + \ldots + c_{a_{i^-}^-}) = \\ &= c_{\sigma(0)} + c_{\sigma(1)} + \ldots + c_{\sigma(i)}, \end{aligned}$$

since $\sigma(0) = 0$ and $\{\sigma(1), \ldots, \sigma(i)\} = \{a_1^+, \ldots, a_{i^+}^+, a_1^-, \ldots, a_{i^-}^-\}$. □

We now show that Lemma 4.8.27 holds for x-separable paths (so, in turn, also for x-static paths).

Lemma 4.8.29. *There exists a function $f_{4.8.29} : \mathbb{N} \to \mathbb{N}$ such that the following holds. Let $x \in V(T^b)$ and assume that the width of $\mathcal{T}$ and $\mathcal{T}^b$ is bounded by $\ell \geqslant 0$. Suppose $v_0v_1 \ldots v_{p+1}$ is an x-separable path in $\mathcal{T}$. Then there exists a boundary-preserving x-block shuffle of the path using a permutation σ such that:*

- *the decomposition $\mathcal{T}'$ after the shuffle has width not greater than the width of $\mathcal{T}$; and*
- *in $\mathcal{T}'$, the vertical path $v_{\sigma(1)} \ldots v_{\sigma(p)}$ contains at most $f_{4.8.29}(\ell)$ x-blocks.*

Proof. The lemma is a consequence of a similar statement from the work of Bojańczyk and Pilipczuk [BP22], formulated for bichromatic words, which in turn captures the understanding of *typical sequences* from the work of Bodlaender and Kloks [BK96]. Before we provide the statement of their lemma, we need to define block shuffles for words. We mostly follow the exposition from [BP22], with the difference that their proof concerns words over alphabet $\{-, +\}$, excluding 0 from the alphabet. However, it can be readily seen that their proof also works in the setting below.

Fix the alphabet $\Sigma = \{0, -, +\}$. Given a word $w \in \Sigma^*$, define:

- $\mathsf{sum}(w)$, the *sum* of w, as the number of occurrences of $+$ in w, minus the number of occurrences of $-$ in w;
- $\mathsf{pmax}(w)$, the *prefix maximum* of w, as the maximum sum of any prefix of w; and
- $\mathsf{pmin}(w)$, the *prefix minimum* of w, as the minimum sum of any prefix of w.

Suppose the characters in a word w are colored with one of two colors, say red and blue; in such an instance we say that w is a *bichromatic word*. A *block* in such a word is a maximal subword comprising consecutive letters of w of the same color. Then a *block shuffle* of w is any word w' created from w by permuting the blocks of w such that within each color, the order of the characters remains the same as in w. Then the Dealternation Lemma for bichromatic words reads as follows:

Claim 4.8.30 ([BP22, Lemma 7.1])**.** *Let $w \in \Sigma^*$ be a bichromatic word. Let $a, b \geqslant 0$ be two integers with the following properties: $\mathsf{pmax}(w) \leqslant a$, and if u is a word created from w by restricting it to all letters of the same color, then $\mathsf{pmin}(u) \geqslant -b$. Then there exists a block shuffle w' of w such that $\mathsf{pmax}(w') \leqslant \mathsf{pmax}(w)$ and w' has at most $a + 4b + 2$ blocks in total.*

Let $f_{4.8.29}(\ell) = 5\ell + 4$. Consider an x-separable path $v_0v_1 \dots v_{p+1}$ and let $c_0, c_1, \dots, c_p \in \mathbb{Z}$ be the constants associated with the path. If the path comprises at most 2 blocks, the lemma follows trivially – we can choose $\mathcal{T}' = \mathcal{T}$ and σ to be the identity permutation. Suppose now the path contains $t \geqslant 3$ blocks: $[\ell_1, r_1], [\ell_2, r_2], \dots, [\ell_t, r_t]$, where $1 = \ell_1 < r_1 < \ell_2 < r_2 < \cdots < \ell_t < r_t = p$ and $\ell_{i+1} = r_i + 1$ for $i \in [t-1]$. Aiming to apply Claim 4.8.30, we construct a bichromatic word w as follows:

- For every $i \in [\ell_2, r_{t-1}]$, define the word w_i as follows:

$$w_i = \begin{cases} +^{c_i} & \text{if } c_i > 0; \\ -^{|c_i|} & \text{if } c_i < 0; \\ 0 & \text{if } c_i = 0. \end{cases}$$

  Then we set $w = w_{\ell_2} w_{\ell_2+1} \dots w_{r_{t-1}}$.

- For every $i \in [\ell_2, r_{t-1}]$, color the letters of w_i in w red if $i \in [\ell_j, r_j]$ for even j, and blue otherwise. (That is, we color the subwords of w corresponding to different x-blocks of the vertical path alternately; in other words, one color is allocated to the subwords corresponding to the x-empty nodes v'_i, and the other to the x-full nodes v'_i).

It is easy to see that block shuffles w' of w are in a natural bijection with boundary-preserving x-block shuffles along $v_0v_1 \dots v_{p+1}$: Any reordering of the blocks in w can be directly translated to a reordering of the x-blocks of the path preserving the first and the last x-block, and vice versa. Such a boundary preserving x-shuffle is said to be *prescribed* by w'.

The following claim about the prefix maximum of w follows straight from the definition.

Claim 4.8.31. $\mathsf{pmax}(w) = \max_{i \in [\ell_2 - 1, r_{t-1}]}(c_{\ell_2} + c_{\ell_2+1} + \ldots + c_i)$.

Note that the width of the edge $v_{\ell_2-1}v_{\ell_2}$ in the original decomposition $\mathcal{T}$ is (trivially) at least 0; and for every $i \in [\ell_2 - 1, r_{t-1}]$, the width of the edge v_iv_{i+1} is (by our assumption) at most ℓ (i.e., it exceeds the width of $v_{\ell_2-1}v_{\ell_2}$ by at most ℓ). Thus, by the x-separability of the path $v_0v_1 \dots v_{p+1}$ for the trivial block shuffle using the identity permutation σ, we have $c_{\ell_2} + c_{\ell_2+1} + \ldots + c_i \leqslant \ell$ for every i and hence $\mathsf{pmax}(w) \leqslant \ell$ by Claim 4.8.31.

Now, suppose we found a block shuffle w' of w with a smaller or equal prefix maximum. Then the block shuffle can be naturally translated to an x-block shuffle of $\mathcal{T}$ of width not greater than the width of $\mathcal{T}'$:

Claim 4.8.32. *Suppose w' is a block shuffle of w with $\mathsf{pmax}(w') \leqslant \mathsf{pmax}(w)$, and let $\mathcal{T}'$ be the decomposition formed from $\mathcal{T}$ by performing a boundary-preserving x-block shuffle prescribed by w'. Then the width of $\mathcal{T}'$ is at most the width of $\mathcal{T}$.*

Proof of the claim. Let σ be the recipe of the block shuffle prescribed by w'; note that for every $i \notin [\ell_2, r_{t-1}]$, it holds that $\sigma(i) = i$. By the same argument as in Claim 4.8.31, we have $\mathsf{pmax}(w') = \max_{i \in [\ell_2 - 1, r_{t-1}]}(c_{\sigma(\ell_2)} + c_{\sigma(\ell_2+1)} + \cdots + c_{\sigma(i)})$.

None of the edges outside of the path $v_{\ell_2-1}v_{\ell_2} \dots v_{r_{t-1}+1}$ are affected by a boundary-preserving x-block shuffle; that is, for any edge e outside of this path, the x-block shuffle preserves the partition of the leaves of T on either side of e. In particular, for every such edge e, the width of e remains unchanged. Hence it is enough to verify the widths of each of the edges $v_{\ell_2-1}v_{\sigma(\ell_2)} = v_{\sigma(\ell_2-1)}v_{\sigma(\ell_2)}$, $v_{\sigma(\ell_2)}v_{\sigma(\ell_2+1)}$, $\dots$, $v_{\sigma(r_{t-1})}v_{\sigma(r_{t-1}+1)} = v_{\sigma(r_{t-1})}v_{r_{t-1}+1}$.

Consider an edge $e = v_{\sigma(i)}v_{\sigma(i+1)}$ for $i \in [\ell_2 - 1, r_{t-1}]$. By the x-separability of $v_0v_1 \dots v_{p+1}$, the width of e is

$$c_{\sigma(0)} + c_{\sigma(1)} + \ldots + c_{\sigma(i)} = \sum_{j=0}^{\ell_2-1} c_j + \sum_{j=\ell_2}^{i} c_{\sigma(i)} \leqslant \sum_{j=0}^{\ell_2-1} c_j + \mathsf{pmax}(w') \leqslant \sum_{j=0}^{\ell_2-1} c_j + \mathsf{pmax}(w).$$

Let $i_{\max} \in [\ell_2 - 1, r_{t-1}]$ be such that $\mathsf{pmax}(w) = c_{\ell_2} + c_{\ell_2+1} + \ldots + c_{i_{\max}}$. Then

$$c_{\sigma(0)} + c_{\sigma(1)} + \ldots + c_{\sigma(i)} \leqslant c_0 + c_1 + \ldots + c_{i_{\max}};$$

that is, by the x-separability, the width of e is upper-bounded by the width of the edge $v_{i_{\max}}v_{i_{\max}+1}$ in the original decomposition $\mathcal{T}$, so in particular by the width of $\mathcal{T}$. ◁

It remains to bound the prefix minima of the restrictions of w to all letters of a single color.

Claim 4.8.33. *Let u be a word created by restricting w to all letters of the same color. Then* $\mathsf{pmin}(u) \geqslant -\ell$.

Proof of the claim. Assume that $u = w_{i_1} w_{i_2} \dots w_{i_z}$ for $i_1 < i_2 < \dots < i_z \in [\ell_2, r_{t-1}]$ such that the subwords $w_{i_1}, w_{i_2}, \dots, w_{i_z}$ all have the same color in w; equivalently, $\{v'_{i_1}, v'_{i_2}, \dots, v'_{i_z}\}$ is the subset of $\{v'_{\ell_2}, v'_{\ell_2+1}, \dots, v'_{r_{t-1}}\}$ comprising exactly the set of x-empty nodes or exactly the set of x-full nodes in $\mathcal{T}$.

By the construction of w, we have

$$\mathsf{pmin}(u) = \min_{j \in [0,z]} (c_{i_1} + c_{i_2} + \dots + c_{i_j}).$$

Now construct:

- a block shuffle w' of w by placing u at the front of w' and all the blocks of the opposite color at the back of w', in the same order as in w;
- a boundary-preserving x-block shuffle $\mathcal{T}'$ of $\mathcal{T}$ along $v_0 v_1 \dots v_{p+1}$ prescribed by w'; let also σ be the recipe of this shuffle. (In other words, $\mathcal{T}'$ is constructed by placing all non-boundary blocks comprising x-empty (resp. x-full) nodes next to each other.)

By construction, we have $\sigma(j) = j$ for all $j \in [0, \ell_2 - 1]$ and $\sigma(\ell_2 + j - 1) = i_j$ for all $j \in [z]$.

Obviously, the width of the edge $v_{\ell_2-1} v_{\ell_2}$ in $\mathcal{T}$ is not larger than ℓ; and by the x-separability of the vertical path $v_0 v_1 \dots v_{p+1}$ for the trivial block shuffle, it is equal to $c_0 + c_1 + \dots + c_{\ell_2-1}$. On the other hand, for every $j \in [0, z]$, the width of the edge $v_{\sigma(\ell_2+j-1)} v_{\sigma(\ell_2+j)}$ in $\mathcal{T}'$ is trivially at least 0; and by the x-separability applied to the block shuffle along σ, it is equal to

$$\sum_{q=0}^{\ell_2+j-1} c_{\sigma(q)} = \sum_{q=0}^{\ell_2-1} c_q + \sum_{q=1}^{j} c_{i_q} \leqslant \ell + \sum_{q=1}^{j} c_{i_q}.$$

Thus $c_{i_1} + c_{i_2} + \dots + c_{i_j} \geqslant -\ell$ for every $j \in [0, z]$. Hence, $\mathsf{pmin}(u) \geqslant -\ell$. ◁

The proof of the lemma follows now in a straightforward way: From Claims 4.8.31 and 4.8.33 it follows that $\mathsf{pmax}(w) \leqslant \ell$ and $\mathsf{pmin}(u) \geqslant -\ell$, where u is the restriction of w to the letters of any chosen color. Hence by Claim 4.8.30, there exists a block shuffle w' of w such that $\mathsf{pmax}(w') \leqslant \mathsf{pmax}(w)$ and w' has at most $5\ell + 2$ blocks in total. Then by Claim 4.8.32, the boundary-preserving x-block shuffle of $\mathcal{T}$ prescribed by w' produces a decomposition $\mathcal{T}'$ of width upper-bounded by the width of $\mathcal{T}$. Moreover, the path $v_{\sigma(0)} v_{\sigma(1)} \dots v_{\sigma(p+1)}$ in $\mathcal{T}'$ after the shuffle has at most $5\ell + 4$ x-blocks: the two boundary x-blocks and one additional x-block for each block of w'. □

It remains to lift the result of Lemma 4.8.29 to general x-shuffleable paths:

Proof of Lemma 4.8.27. We will show that every x-shuffleable path can be partitioned into a small number of x-static paths. Then the proof will follow from Lemmas 4.8.28 and 4.8.29.

Recall that in our setting, $\mathcal{T}$ and $\mathcal{T}^b$ are rooted rank decompositions of $\mathcal{V}$ of width at most ℓ ($\ell \geqslant 0$) and $v_0 v_1 \dots v_{p+1}$ is an x-shuffleable path in $\mathcal{T}$.

Claim 4.8.34. *There exists a partition of the interval $[1, p]$ into $t \leqslant 8\ell + 1$ subintervals $[\ell_1, r_1], [\ell_2, r_2], \dots, [\ell_t, r_t]$ such that, for every $i \in [t]$, either $\ell_i = r_i$ or the vertical path $v_{\ell_i - 1} v_{\ell_i} v_{\ell_i + 1} \dots v_{r_i} v_{r_i + 1}$ is x-static. Moreover, any two subintervals with $\ell_i \neq r_i$ are separated by a one-element subinterval.*

Proof of the claim. For every $i \in [0, p]$, define the *profile* of the edge $v_i v_{i+1}$ in $\mathcal{T}$ as the quadruple of integers $(\alpha_i, \beta_i, \gamma_i, \delta_i)$, where

$$\begin{aligned}
\alpha_i &= \dim(\langle \mathcal{L}_x(T)[v_i \vec{v_{i+1}}] \rangle \cap B_x), \\
\beta_i &= \dim(\langle \mathcal{L}_{\bar{x}}(T)[v_i \vec{v_{i+1}}] \rangle \cap B_x), \\
\gamma_i &= \dim(\langle \mathcal{L}_x(T)[\vec{v_{i+1}} v_i] \rangle \cap B_x), \\
\delta_i &= \dim(\langle \mathcal{L}_{\bar{x}}(T)[\vec{v_{i+1}} v_i] \rangle \cap B_x).
\end{aligned}$$

By construction, the sequences $(\alpha_i)_{i=0}^p$ and $(\beta_i)_{i=0}^p$ are nondecreasing, while the sequences $(\gamma_i)_{i=0}^p$ and $(\delta_i)_{i=0}^p$ are nonincreasing; moreover, each of the values $\alpha_i, \beta_i, \gamma_i, \delta_i$ range from 0 from ℓ since each value describes the dimension of a subspace of B_x (and $\dim(B_x) \leqslant \ell$ as $\mathcal{T}^b$ has width at most ℓ). Therefore, there exist at most $4\ell + 1$ different profiles among all the edges $v_i v_{i+1}$.

Say a vertex v_i ($i \in [p]$) is a *milestone* if the edges $v_{i-1} v_i$ and $v_i v_{i+1}$ have different profiles; observe that on the path $v_0 v_1 \dots v_{p+1}$, there are at most 4ℓ milestones. We construct a partition of $[1, p]$ into subintervals by:

- creating, for each milestone v_i, a one-element subinterval $[i,i]$; and
- adding to the partition all maximal subintervals of $[1,p]$ not containing any milestones.

It is obvious that the partition contains at most $8\ell+1$ subintervals. Now, let $[\ell,r]$ be some maximal subinterval of $[1,p]$ without any milestones. We claim that the path $v_{\ell-1}v_\ell\dots v_rv_{r+1}$ is x-static. Since none of the vertices $v_\ell,\dots,v_r$ are milestones, the profiles of the edges $v_{\ell-1}v_\ell$ and v_rv_{r+1} are equal. Since $\alpha_{\ell-1}=\alpha_r$ and $\langle\mathcal{L}_x(T)[v_r\vec{v_{r+1}}]\rangle\cap B_x\subseteq\langle\mathcal{L}_x(T)[v_{\ell-1}\vec{v_\ell}]\rangle\cap B_x$, we conclude that $\langle\mathcal{L}_x(T)[v_r\vec{v_{r+1}}]\rangle\cap B_x=\langle\mathcal{L}_x(T)[v_{\ell-1}\vec{v_\ell}]\rangle\cap B_x$; this verifies one of the equalities required by the definition of x-static paths. The remaining three equalities are proved analogously by analyzing the equalities $\beta_{\ell-1}=\beta_r$, $\gamma_{\ell-1}=\gamma_r$ and $\delta_{\ell-1}=\delta_r$. ◁

Let $[\ell_1,r_1],\dots,[\ell_t,r_t]$ be the partition of $[1,p]$ given by Claim 4.8.34, and suppose that $\ell_1\leqslant r_1<\ell_2\leqslant r_2<\dots<\ell_t\leqslant r_t$. We inductively construct a sequence of rank decompositions $\mathcal{T}_0=\mathcal{T},\mathcal{T}_1,\dots,\mathcal{T}_t$ with the following invariants:

- for every $1\leqslant j\leqslant i\leqslant t$, vertices $v_{\ell_j},\dots,v_{r_j}$ form – in some order – a path in $\mathcal{T}_i$ with at most $f_{4.8.29}(\ell)$ x-blocks; and
- for every $0\leqslant i<j\leqslant t$ with $\ell_j\neq r_j$, the vertical path $v_{\ell_j-1}v_{\ell_j}\dots v_{r_j}v_{r_j+1}$ is x-static in $\mathcal{T}_i$.

We construct this sequence as follows: Iterate the integers $i=1,\dots,t$. If $\ell_i=r_i$, then set $\mathcal{T}_i=\mathcal{T}_{i-1}$. Otherwise, since the vertical path $v_{\ell_i-1}v_{\ell_i}\dots v_{r_i}v_{r_i+1}$ is x-static in $\mathcal{T}_{i-1}$, it is also x-separable (Lemma 4.8.28); hence, we apply Lemma 4.8.29 to produce a boundary-preserving x-block shuffle $\mathcal{T}_i$ from $\mathcal{T}_{i-1}$, where the vertices $v_{\ell_i},v_{\ell_i+1},\dots,v_{r_i}$ form a vertical path with at most $f_{4.8.29}(\ell)$ x-blocks. Since each boundary-preserving x-block only modifies the decomposition locally along the path $v_{\ell_i-1}v_{\ell_i}\dots v_{r_i}v_{r_i+1}$, it can be easily verified that all the invariants are preserved by the update. Also, note that boundary-preserving x-block shuffles of $v_{\ell_i-1}v_{\ell_i}\dots v_{r_i+1}$ are also boundary-preserving x-block shuffles of $v_0v_1\dots v_{p+1}$ and a composition of (boundary-preserving) x-block shuffles is also a (boundary-preserving) x-block shuffle. We thus conclude that $\mathcal{T}_t$ is a boundary-preserving x-block shuffle of $\mathcal{T}$ along the path $v_0v_1\dots v_{p+1}$. Moreover, by the invariants, for every $j\in[t]$, the vertices $v_{\ell_j},\dots,v_{r_j}$ form a vertical path in $\mathcal{T}_t$ with at most $f_{4.8.29}(\ell)$ x-blocks. Since $t\leqslant 8\ell+1$, we find that the produced decomposition contains at most $f_{4.8.27}(\ell):=(8\ell+1)f_{4.8.29}(\ell)$ x-blocks along the vertical path $v_{\sigma(1)}\dots v_{\sigma(p)}$. Also the width of $\mathcal{T}_t$ is upper-bounded by the width of $\mathcal{T}$. □

4.8.6 Proof of the Local Dealternation Lemma

With all the required tools at hand, we can prove the Local Dealternation Lemma (Lemma 4.8.14).

Proof of Lemma 4.8.14. Let T^{M} be the x-mixed skeleton of $\mathcal{T}$; by our assumption, it has at most $f_{4.8.13}(\ell)$ nodes. If the skeleton is an empty tree, then by Lemma 4.8.9, the root r of $\mathcal{T}$ either x-full or x-empty. In either case, the lemma follows by setting $\mathcal{T}'=\mathcal{T}$, in which case the set $\mathcal{L}(\mathcal{T}')[x]$ is a disjoint union of at most one x-factor of $\mathcal{T}'$. From now on assume that the skeleton is nonempty.

The following observations are straightforward:

Claim 4.8.35. *Let $uv\in E(T^{\mathsf{M}})$, where u is a parent of v in T^{M} (i.e., u is an ancestor of v in T). Then the simple vertical path between u and v in T is x-shuffleable.*

Proof of the claim. Let w be an internal node of the path (so $w\notin V(T^{\mathsf{M}})$), w^+ be the child of w on the path and w' be the child of w not on the path. By Lemma 4.8.11, the node w^+ is x-mixed. Hence it cannot be that w' is x-mixed – otherwise, by definition, w would be an x-branch point. ◁

For the following observation, let r be the root of T and r^{M} be the root of T^{M}.

Claim 4.8.36. *If $r\neq r^{\mathsf{M}}$, then the simple vertical path between r and r^{M} in T is x-shuffleable.*

Proof of the claim. Let w, w^+ and w' be defined as in Claim 4.8.35. Since w^+ is an ancestor of r^{M}, the node w^+ is x-mixed. As in the previous claim, we conclude that w' cannot be x-mixed. ◁

We now create a new rank decomposition $\mathcal{T}'$ of $\mathcal{V}$ as follows: For every $uv \in E(T^{\mathsf{M}})$ in arbitrary order, perform on $\mathcal{T}$ a boundary-preserving x-block shuffle of the vertical path between u and v in T compliant with the statement of Lemma 4.8.27. Also, when $r \neq r^{\mathsf{M}}$, apply an analogous boundary-preserving x-block shuffle of the vertical path between r and r^{M} in $\mathcal{T}$. Let then $\mathcal{T}'$ be the decomposition after applying all the x-block shuffles. We will now verify that $\mathcal{T}'$ satisfies all the requirements of the lemma.

First, by Lemma 4.8.15 and the fact that each x-block shuffle is a composition of x-swaps, it follows that, for every $y \in V(T^b)$, the y-mixed skeletons of $\mathcal{T}$ and $\mathcal{T}'$ are equal; in particular, T^{M} is the x-mixed skeleton of $\mathcal{T}'$. Next, assume that $y \in V(T^b)$ with $y \not> x$. That every y-factor of $\mathcal{T}$ is also a y-factor of $\mathcal{T}'$ follows immediately from Lemma 4.8.24. It remains to show that the set $\mathcal{L}(\mathcal{T}^b)[x]$ can be decomposed into $f_{4.8.14}(\ell)$ x-factors of $\mathcal{T}'$, for some function $f_{4.8.14}$ yet to be defined. To this end, we will use the following simple claim:

Claim 4.8.37. *Let $v_0v_1 \dots v_{p+1}$ be an x-shuffleable path in $\mathcal{T}'$. Assume that the path $v_1 \dots v_p$ is comprised of $n \geqslant 1$ x-blocks. Then the set $\mathcal{L}_x(\mathcal{T}')[v_1] \setminus \mathcal{L}_x(\mathcal{T}')[v_{p+1}]$ can be decomposed into at most n x-context factors of $\mathcal{T}'$.*

Proof of the claim. For every $i \in [p]$, let v_i' be the child of v_i not on the path. Since $v_0v_1 \dots v_{p+1}$ is x-shuffleable, every node v_i' is either x-empty or x-full. Moreover, each x-block $[\ell, r] \subseteq [1, p]$ is either an x-empty block (and then $\mathcal{L}(\mathcal{T})[v_\ell] \setminus \mathcal{L}(\mathcal{T})[v_{r+1}]$ is disjoint from $\mathcal{V}_x$) or an x-full block (and then $\mathcal{L}(\mathcal{T})[v_\ell] \setminus \mathcal{L}(\mathcal{T})[v_{r+1}] \subseteq \mathcal{V}_x$; so in particular, $\mathcal{L}(\mathcal{T})[v_\ell] \setminus \mathcal{L}(\mathcal{T})[v_{r+1}]$ is an x-context factor of $\mathcal{T}'$). Therefore, $\mathcal{L}_x(\mathcal{T})[v_1] \setminus \mathcal{L}_x(\mathcal{T})[v_{p+1}] = (\mathcal{L}(\mathcal{T})[v_1] \setminus \mathcal{L}(\mathcal{T})[v_{p+1}]) \cap \mathcal{V}_x$ is a disjoint union of x-context factors of the form $\mathcal{L}(\mathcal{T})[v_\ell] \setminus \mathcal{L}(\mathcal{T})[v_{r+1}]$, ranging over all x-full blocks $[\ell, r] \subseteq [1, p]$. ◁

Let r be the root of T', and r^{M} be the root of T^{M}. Observe that every leaf l of T' can be uniquely assigned to one of the following groups:

- the group of leaves that are not descendants of r^{M} (if $r \neq r^{\mathsf{M}}$);
- for every x-leaf point $v \in V(T^{\mathsf{M}})$, the group of leaves that are descendants of v;
- for every edge $uv \in E(T^{\mathsf{M}})$, where u is an ancestor of v in T, the group of leaves that are descendants of w but not v, where w is the child of u on the path between u and v in T.

Thus, $\mathcal{L}(\mathcal{T}^b)[x] = \mathcal{L}_x(\mathcal{T}')[r]$ is the disjoint union of the following sets:

- $\mathcal{L}_x(\mathcal{T}')[r] \setminus \mathcal{L}_x(\mathcal{T}')[r^{\mathsf{M}}]$ (if $r \neq r^{\mathsf{M}}$);
- for every x-leaf point $v \in V(T^{\mathsf{M}})$, the set $\mathcal{L}_x(\mathcal{T}')[v]$;
- for every edge $uv \in E(T^{\mathsf{M}})$, where u is an ancestor of v in T, the set $\mathcal{L}_x(\mathcal{T}')[w] \setminus \mathcal{L}_x(\mathcal{T}')[v]$, where w is the child of u on the path between u and v in T.

If $r \neq r^{\mathsf{M}}$, then let $r_0r_1 \dots r_t$ be the vertical path between r and r^{M} ($r_0 = r$, $r_t = r^{\mathsf{M}}$, $t \geqslant 1$). Since we performed a boundary-preserving x-block shuffle along the vertical path $r_0r_1 \dots r_t$, we get that the vertical path $r_1 \dots r_{t-1}$ comprises at most $f_{4.8.27}(\ell)$ x-blocks; hence, by Claim 4.8.37, the set $\mathcal{L}_x(\mathcal{T}')[r_1] \setminus \mathcal{L}_x(\mathcal{T}')[r^{\mathsf{M}}]$ can be partitioned into at most $f_{4.8.27}(\ell)$ x-context factors of $\mathcal{T}'$. Let r' be the child of r not on the path from r to r^{M}; then r' is x-empty or x-full. If r' is x-full, we add one additional tree factor $\mathcal{L}(\mathcal{T}')[r']$ to the partition. So $\mathcal{L}_x(\mathcal{T}')[r] \setminus \mathcal{L}_x(\mathcal{T}')[r^{\mathsf{M}}] = \mathcal{L}_x(\mathcal{T}')[r'] \cup \big(\mathcal{L}_x(\mathcal{T}')[r_1] \setminus \mathcal{L}_x(\mathcal{T}')[r^{\mathsf{M}}]\big)$ can be partitioned into at most $f_{4.8.27}(\ell) + 1$ x-factors of $\mathcal{T}'$.

Next, for every edge $uv \in V(T^{\mathsf{M}})$, where u is an ancestor of v in T, let w be the child of u on the path from u to v in T'. Applying Claim 4.8.37, we get that the set $\mathcal{L}_x(\mathcal{T}')[w] \setminus \mathcal{L}_x(\mathcal{T}')[v]$ can be partitioned into $f_{4.8.27}(\ell)$ x-context factors of $\mathcal{T}'$. Finally, for every x-leaf point v in T', one child v^+ of v is x-full and the other child v^- is x-empty. So $\mathcal{L}_x(\mathcal{T}')[v] = \mathcal{L}(\mathcal{T}')[v^+]$ and the set $\mathcal{L}_x(\mathcal{T}')[v]$ is exactly an x-tree factor of $\mathcal{T}'$.

Summing up, we can partition the set $\mathcal{L}(\mathcal{T}^b)[x] = \mathcal{L}_x(\mathcal{T}')[r]$ into at most

$$f_{4.8.27}(\ell) \cdot (|E(T^{\mathsf{M}})| + 1) + |V(T^{\mathsf{M}})| + 1 \leqslant (f_{4.8.27}(\ell) + 1) f_{4.8.13}(\ell) + 1$$

x-factors of T'. This finishes the proof of the Local Dealternation Lemma and it is enough to set $f_{4.8.14}(\ell) = (f_{4.8.27}(\ell) + 1) f_{4.8.13}(\ell) + 1$. □

4.9 Using rank decomposition automata to compute closures

This section is dedicated to the proofs of Lemmas 4.6.2 and 4.4.7. Along the way, we produce two rank decomposition automata that will be used by us heavily throughout the proof:

- the *exact rankwidth automaton* (Section 4.9.1) that for two fixed integers k, ℓ verifies, given an annotated rank decomposition of width ℓ encoding a partitioned graph $(G, \mathcal{C})$, whether $(G, \mathcal{C})$ has rankwidth at most k; and
- the *closure automaton* (Section 4.9.2) that, roughly speaking, for an annotated tree decomposition $\mathcal{T}$ encoding a graph G and a prefix T_{pref} of $\mathcal{T}$, represents how a c-small k-closure of T_{pref} can look like in each subtree rooted at an edge $\vec{px} \in \vec{\mathsf{App}}_T(T_{\text{pref}})$.

The exact rankwidth automaton given in Section 4.9.1 will imply Lemma 4.6.2. Finally, in Section 4.9.4, we will use both automata to produce a data structure for minimal closures of Lemma 4.4.7.

In this section, we rely on the concepts and notation defined in Section 4.8.1, in particular the subspace arrangements of linear spaces and rank decompositions thereof.

4.9.1 Exact rankwidth automaton

In this subsection, we will present an implementation of the exact rankwidth automaton. As a consequence, we will also show that for any pair of integers $k, \ell \in \mathbb{N}$, one can determine – in linear time with respect to the size of the graph – whether a partitioned graph, encoded by an annotated rank decomposition of width at most ℓ, has rankwidth at most k. Moreover, in the positive case, in linear time we can recover a rank decomposition of the partitioned graph of width at most k (or in near-linear time if we require the output to be an annotated decomposition). The construction of the automaton crucially relies on the understanding of the cubic-time algorithm of Jeong, Kim and Oum [JKO21] computing optimum-width rank decompositions of graphs and, more generally, subspace arrangements. We proceed to give a summary of this algorithm below.

Summary of the algorithm of Jeong, Kim and Oum [JKO21]. The $\mathcal{O}_\ell(n^3)$ algorithm of [JKO21] for rankwidth of subspace arrangements uses at its core the following subroutine: given two integers k, ℓ with $\ell \geqslant k$, a subspace arrangement $\mathcal{V}$ of subspaces of $\mathbb{F}^d$, and a rank decomposition of $\mathcal{V}$ of width ℓ, determine whether a rank decomposition of $\mathcal{V}$ of width k exists; and if so, construct any such decomposition. This subroutine is an analog of a similar linear-time algorithm for tree decompositions of graphs by Bodlaender and Kloks [BK96]. Here, we provide a brief description of the subroutine in [JKO21].

Suppose $\mathcal{T}^b = (T^b, \lambda^b)$ is a rooted rank decomposition of $\mathcal{V}$ of width ℓ. Ideally, we would wish to compute, for every node $x \in V(\mathcal{T}^b)$, the set of all possible (unrooted) rank decompositions of $\mathcal{V}_x$ of width at most k. Such sets would be computed using a bottom-up dynamic programming scheme on $\mathcal{T}^b$ – the only slightly nontrivial part is understanding, for a node $x \in V(\mathcal{T}^b)$ with two children c_1, c_2, how to find the set of all rank decompositions of $\mathcal{V}_x$ of small width, given the corresponding sets of decompositions of $\mathcal{V}_{c_1}$ and $\mathcal{V}_{c_2}$. Obviously, this idea, while correct, is doomed to fail since a graph can (and usually will) have an exponential number of valid rank decompositions of small width.

Thus, [JKO21] mimics the insight of Bodlaender and Kloks that, for each rank decomposition of $\mathcal{V}_x$ of small width, we can record just essential information about it, which. Roughly speaking, this information is a heavily compressed version of the rank decomposition, with the details irrelevant to the subspaces in $\mathcal{V} \setminus \mathcal{V}_x$ stripped off. This information is named a *compact B-namu* in [JKO21].[16] Precisely, given a subspace B of $\mathbb{F}^d$, we define a B-namu as a tuple $(T, \alpha, \mathbf{w}, U)$, where: (i) T is a subcubic tree, possibly with some degree-2 nodes, (ii) U is a subspace of B, (iii) every oriented edge $\vec{uv} \in \vec{E}(T)$ is decorated with a subspace $\alpha(\vec{uv}) \subseteq U$, (iv) every edge $uv \in E(T)$ is decorated with an integer $\mathbf{w}(uv) \geqslant 0$. (There are a couple of additional restrictions on the values of $\alpha(\cdot)$ and $\mathbf{w}(\cdot)$ – that is, we have $\alpha(v_1\vec{v}_2) \subseteq \alpha(v_3\vec{v}_4)$ whenever $v_1\vec{v}_2$ is a predecessor of $v_3\vec{v}_4$ in T, and we have $\mathbf{w}(uv) \geqslant \dim(\alpha(\vec{uv}) \cap \alpha(\vec{vu}))$ for all $uv \in E(T)$ – but these will be unimportant for our purposes. Similarly, their definition of a B-namu includes additional objects that can be uniquely deduced from $(T, \alpha, \mathbf{w}, U)$.) The width of a B-namu is the maximum value of $\mathbf{w}(uv)$, or 0 if T is edgeless.

Note that there exists a natural way of turning a rank decomposition $\mathcal{T} = (T, \lambda)$ of $\mathcal{V}$ into a B-namu $(T, \alpha, \mathbf{w}, U)$: We define $\alpha(\vec{uv}) = B \cap \langle \mathcal{L}(\mathcal{T})[\vec{uv}] \rangle$, $\mathbf{w}(uv) = \dim(\langle \mathcal{L}(\mathcal{T})[\vec{uv}] \rangle \cap \langle \mathcal{L}(\mathcal{T})[\vec{vu}] \rangle)$, and $U = B \cap \langle \mathcal{V} \rangle$.

[16] In [BK96], an analogous piece of information is called a *characteristic*.

It is a straightforward exercise to verify that such a construction indeed produces a valid B-namu of width equal to the width of $\mathcal{T}$.

For the purposes of this summary we do not describe how to compress a B-namu into the equivalent compact B-namu of equal width [JKO21, Section 3.2]. Here, we only present the most essential takeaway of this process: Assuming bounded $\dim(B)$, compact B-namus of bounded width are small and all such B-namus can be generated quickly. Henceforth, let $U_k(B)$ denote the set of all compact B-namus of width at most k. Next, for any ordered basis $\mathfrak{B} = (\mathbf{v}_1, \ldots, \mathbf{v}_{|\mathfrak{B}|})$ of B, let $U_k(\mathfrak{B})$ denote the same set of compact B-namus, but where each subspace of B is represented in the basis $\mathfrak{B}$. (So in a B-namu represented in the ordered basis $\mathfrak{B}$ of B, every subspace $A \subseteq B$ is encoded by a sequence of $\dim(A) \cdot |\mathfrak{B}|$ bits c_{ij} for $i \in [\dim(A)]$, $j \in [|\mathfrak{B}|]$ as the subspace spanned by vectors $\sum_{j=1}^{|\mathfrak{B}|} c_{ij}\mathbf{v}_j$ for $i \in [\dim(A)]$.) For $t \in \mathbb{N}$, let U_k^t denote the set of all possible encodings of compact B-namus of width at most k in an ordered basis of size at most t. (So $U_k(\mathfrak{B}) \subseteq U_k^t$ for every ordered basis $\mathfrak{B}$ with $|\mathfrak{B}| \leqslant t$.) Then we have that:

Lemma 4.9.1 ([JKO21, Lemma 5.4])**.** *There exist functions $f_{4.9.1} : \mathbb{N}^2 \to \mathbb{N}$ and $g_{4.9.1}, h_{4.9.1} : \mathbb{N}^3 \to \mathbb{N}$ such that the following holds. Assume $\dim(B) = \ell \geqslant 0$. If $\Gamma = (T, \alpha, \mathbf{w}, U)$ is a compact B-namu of width at most k, then $|E(T)| \leqslant f_{4.9.1}(k, \ell)$. Moreover, we have that $|U_k(B)| \leqslant g_{4.9.1}(k, \ell, |\mathbb{F}|)$ and moreover, the entire set U_k^ℓ can be generated in time $h_{4.9.1}(k, \ell, |\mathbb{F}|)$.*

Now, given $\mathcal{T}^b$, Jeong, Kim and Oum aim to compute for each $x \in V(T^b)$ the *full set* at x of width k with respect to $\mathcal{T}^b$: essentially, the set of compact B_x-namus of all possible *totally pure* unrooted rank decompositions of $\mathcal{V}_x$ of width at most k.[17] This full set is denoted $\mathrm{FS}_k(x)$. Note that $\mathrm{FS}_k(x) \subseteq U_k(B_x)$. Since $\mathbb{F} = \mathrm{GF}(2)$ and $|B_x| \leqslant \ell$ (since $\mathcal{T}^b$ is a decomposition of width at most ℓ), we see that $|\mathrm{FS}_k(x)| \leqslant |U_k(B_x)| \leqslant g_{4.9.1}(k, \ell, 2)$. Similarly, for an ordered basis $\mathfrak{B}_x$ of B_x, let $\mathrm{FS}_k^{\mathfrak{B}_x}(x) \subseteq U_k(\mathfrak{B}_x) \subseteq U_k^{|\mathfrak{B}_x|}$ denote the set $\mathrm{FS}_k(x)$, but where all subspaces of B_x are represented in the ordered basis $\mathfrak{B}_x$ as described above.

In order to facilitate the efficient computation of the full sets, [JKO21] introduces the notion of a *transcript* of $\mathcal{T}^b$. Recall that the boundary space of x is defined as $B_x = \langle \mathcal{V}_x \rangle \cap \langle \mathcal{V} \setminus \mathcal{V}_x \rangle$. We now also define the space B'_x as follows:

$$B'_x = \begin{cases} B_x & \text{if } x \text{ is a leaf of } T^b, \\ B_x + B_{c_1} + B_{c_2} & \text{if } x \text{ is an internal node of } T^b \text{ with children } c_1 \text{ and } c_2. \end{cases}$$

([JKO21] equivalently uses $B_{c_1} + B_{c_2}$ in the second case, after having proved the inclusion $B_x \subseteq B_{c_1} + B_{c_2}$.) Then a *transcript* of $\mathcal{T}^b$ is formed from two sets of ordered bases $\{\mathfrak{B}_x\}_{x \in V(T^b)}$ and $\{\mathfrak{B}'_x\}_{x \in V(T^b)}$ under the following conditions for all $x \in V(T^b)$:

- $\mathfrak{B}_x$ is an ordered basis of B_x and $\mathfrak{B}'_x$ is an ordered basis of B'_x;
- $\mathfrak{B}_x$ is a prefix of $\mathfrak{B}'_x$.

Then, for any nonroot node x with parent y, we define the *transition matrix* of x as the unique $|\mathfrak{B}'_y| \times |\mathfrak{B}_x|$ matrix $M_{x\vec{y}}$ over $\mathbb{F}$ with the following property: Suppose $\mathbf{v}$ is a vector in B_x and that $\mathbf{v}' \in \mathbb{F}^{|\mathfrak{B}_x|}$ is the (unique) representation of $\mathbf{v}$ in the ordered basis $\mathfrak{B}_x$, that is, $\mathbf{v} = \sum_{i=1}^{|\mathfrak{B}_x|} \mathbf{v}'_i \cdot (\mathfrak{B}_x)_i$. Then $M_{x\vec{y}}\mathbf{v}'$ is the unique representation of $\mathbf{v}$ in the ordered basis $\mathfrak{B}'_y$. Intuitively, $M_{x\vec{y}}$ describes how the space B_x embeds as a subspace in B'_y. Notably, this description has bitsize bounded by $\mathcal{O}(\ell^2)$ since $|\mathfrak{B}_x| = \dim(B_x) \leqslant \ell$ and $|\mathfrak{B}'_y| = \dim(B'_y) \leqslant \mathcal{O}(\ell)$, even though B_x and B'_y are subspaces of the highly-dimensional space $\mathbb{F}^d$. This should be contrasted with the actual ordered bases $\mathfrak{B}_x$, $\mathfrak{B}'_y$: The representation of each ordered basis requires $\Omega(d)$ bits, and in our setting we will have $d = n$. Therefore, even storing the transcript $\{\mathfrak{B}_x\}_{x \in V(T^b)}$, $\{\mathfrak{B}'_x\}_{x \in V(T^b)}$ requires $\Omega(n^2)$ bits of storage, so we cannot hope to compute it in subquadratic time.

It is then proved that:

Lemma 4.9.2 (informal statement of [JKO21, Theorem 7.8])**.** *Suppose that the subspace arrangement $\mathcal{V}$ is suitably preprocessed and let $n = |\mathcal{V}|$. Moreover, assume that each subspace in $\mathcal{V}$ has dimension at most ℓ. Then given a rooted rank decomposition $\mathcal{T}^b$ of width at most ℓ, we can compute a transcript $(\{\mathfrak{B}_x\}_{x \in V(T^b)}, \{\mathfrak{B}'_x\}_{x \in V(T^b)})$ and the set of transition matrices $M_{x\vec{y}}$ in time $\mathcal{O}_\ell(n^2)$.*

[17] The notion of totally pure decompositions is defined in Section 4.8.1, however the exact definition is not relevant here. For this recap, it is enough to remember that a rank decomposition of $\mathcal{V}$ of width at most k exists if and only if a totally pure rank decomposition of $\mathcal{V}$ of width at most k also exists (Lemma 4.8.7).

The quadratic dependency on the size of the subspace arrangement in Lemma 4.9.2 is a bottleneck of the algorithm in [JKO21]. The reason the algorithm in Lemma 4.9.2 is inefficient is that it *does* determine the transcript explicitly; it is, however, not clear at all how to avoid this step when processing general subspace arrangements. Our contribution is to show that in the setting of rank decompositions of graphs, the transition matrices of some fixed transcript of $\mathcal{T}^b$ can be efficiently inferred from an annotated rank decomposition that encodes a graph.

Finally, Jeong et al. prove the following claim. Note that this statement is not present explicitly in their work, but it follows immediately from the analysis of their Algorithm 3.1 and their discussion of Proposition 7.10 in Section 7.5.

Lemma 4.9.3 ([JKO21]). *Suppose* $(\{\mathfrak{B}_x\}_{x\in V(T^b)}, \{\mathfrak{B}'_x\}_{x\in V(T^b)})$ *is a transcript of* $\mathcal{T}^b$ *and for every* $x \in V(T^b)$ *with parent* y, $M_{x\vec{y}}$ *is the transition matrix of* x *with respect to the transcript. Then, for any* $x \in V(T^b)$*:*

- *If* x *is a leaf of* T^b*, then* $\mathrm{FS}_k^{\mathfrak{B}_x}(x)$ *contains exactly one* B_x*-namu that can be computed knowing only the cardinality of* $\mathfrak{B}_x$ *in time* $\mathcal{O}_\ell(1)$*.*
- *If* x *is a nonleaf node of* T^b *with two children* c_1, c_2*, then* $\mathrm{FS}_k^{\mathfrak{B}_x}(x)$ *can be computed from* $\mathrm{FS}_k^{\mathfrak{B}_{c_1}}(c_1)$*,* $\mathrm{FS}_k^{\mathfrak{B}_{c_2}}(c_2)$*,* $M_{c_1\vec{x}}$*,* $M_{c_2\vec{x}}$ *and* $|\mathfrak{B}_x|$ *in time* $\mathcal{O}_\ell(1)$*.*

Finally, the authors show how to construct a rank decomposition of small width, having computed all full sets:

Lemma 4.9.4 ([JKO21, Proposition 7.12]). *Let* r *be the root of* T^b *and let* $n = |\mathcal{V}|$*. Then* $\mathcal{V}$ *admits a rank decomposition of width at most* k *if and only if* $\mathrm{FS}_k(r) \neq \emptyset$ *(equivalently,* $\mathrm{FS}_k^{\mathfrak{B}^r}(r) \neq \emptyset$*). If such a decomposition exists, then a (rooted) rank decomposition of* $\mathcal{V}$ *of width at most* k *can be constructed from the set of transition matrices* $M_{x\vec{y}}$*, where* $x \in V(T^b)$ *and* y *is the parent of* x*, and full sets* $\mathrm{FS}_k^{\mathfrak{B}_x}(x)$ *for* $x \in V(T^b)$ *in time* $\mathcal{O}_\ell(n)$*.*

Transcript and transition matrices in annotated rank decompositions. Assume we are given a rooted annotated rank decomposition $\mathcal{T}^b = (T^b, U^b, \mathcal{R}^b, \mathcal{E}^b, \mathcal{F}^b)$ of width ℓ encoding a partitioned graph $(G, \mathcal{C})$. Assume for convenience that the vertices of G are assigned integer labels from 1 to $n := |V(G)|$. Recall from Section 4.8.1 that $\mathcal{T}^b$ is isomorphic to a rank decomposition of the subspace arrangement $\mathcal{V} = \{A_C\}_{C\in\mathcal{C}}$ of width 2ℓ, where for every $C \in \mathcal{C}$, $A_C \subseteq \mathrm{GF}(2)^n$ is the canonical subspace of C, spanned by the vectors $\mathbf{e}_v$ and $\sum_{u\in N(v)} \mathbf{e}_u$ for all $v \in C$. Henceforth, without worrying about confusion, we will simultaneously treat $\mathcal{T}^b$ as an annotated rank decomposition of $(G, \mathcal{C})$ and as a rank decomposition of $\mathcal{V}$. Whenever we consider an edge $xp \in E(\mathcal{T}^b)$, where p is the parent of x, by the width of xp we mean its width $q \in [0, \ell]$ in the decomposition of $(G, \mathcal{C})$; so $\dim(B_x) = 2q$.

Our current aim is to define a specific transcript of $\mathcal{T}^b$ – which we shall name a *canonical transcript* of $\mathcal{T}^b$ – and then show that for any $x \in V(T^b)$ with parent p, the transition matrix $M_{x\vec{p}}$ with respect to the canonical transcript can be uniquely and efficiently deduced from the annotations around x in $\mathcal{T}^b$.

We begin with understanding the boundary space B_x for a node $x \in V(T^b)$. Recall that $B_x = \langle\mathcal{V}_x\rangle \cap \langle\mathcal{V} \setminus \mathcal{V}_x\rangle$. For convenience, we introduce the following shorthand notation: $\mathbf{e}_S := \sum_{u\in S} \mathbf{e}_u$ for any $S \subseteq V(G)$.

Lemma 4.9.5. *Let* x *be a nonroot node of* T^b *and* p *the parent of* x*, and let* $q \in [0, \ell]$ *be the width of the edge* xp *in* $\mathcal{T}^b$*. Let also* $S = \mathcal{L}(\mathcal{T}^b)[x\vec{p}] \subseteq V(G)$ *be the set of vertices of* G *assigned to leaf edges in the subtree of* x *in* $\mathcal{T}^b$*. Then:*

- *The subspace* $A_{x\vec{p}} := \langle\{\mathbf{e}_{N(v)\setminus S} \mid v \in \mathcal{R}^b(x\vec{p})\}\rangle$ *of* $GF(2)^n$ *has dimension* q*;*
- *The subspace* $A_{p\vec{x}} := \langle\{\mathbf{e}_{N(v)\cap S} \mid v \in \mathcal{R}^b(p\vec{x})\}\rangle$ *of* $GF(2)^n$ *has dimension* q*;*
- $B_x = A_{x\vec{p}} + A_{p\vec{x}}$ *and* $A_{x\vec{p}} \cap A_{p\vec{x}} = \{\mathbf{0}\}$*.*

Proof. Recall that the rank q of the edge xp is defined as the rank of the 0-1-matrix M describing adjacencies between vertices in S and vertices in $\overline{S}$ over GF(2). Supposing the rows of M are indexed by S and the columns are indexed by $\overline{S}$, we see that the row rank of M is exactly

$$\dim(\langle\{\mathbf{e}_{N(v)\setminus S} \mid v \in S\}\rangle) = \mathrm{rk}(M) = q.$$

(This is because for any $v \in S$, the vth row of M is given exactly by the vector $\mathbf{e}_{N(v)\setminus S}$, with the 0 entries of the vector corresponding to the elements of S removed.) Since $\mathcal{R}^b(\vec{xp})$ is a representative of S in G, we immediately have that $\{\mathbf{e}_{N(v)\setminus S} \mid v \in S\} = \{\mathbf{e}_{N(v)\setminus S} \mid v \in \mathcal{R}^b(\vec{xp})\}$ and the first statement of the lemma follows. The second point is proved analogously, only that we consider the column rank of M instead.

For the final point, recall that

$$\langle \mathcal{V}_x \rangle = \langle \{\mathbf{e}_v \mid v \in S\} \cup \{\mathbf{e}_{N(v)} \mid v \in S\} \rangle.$$

For every $v \in S$, we subtract from $\mathbf{e}_{N(v)}$ all vectors $\mathbf{e}_u$ with $u \in N(v) \cap S$; since such vectors belong to $\langle \mathcal{V}_x \rangle$, this operation does not change the subspace spanned by vectors and thus

$$\begin{aligned}\langle \mathcal{V}_x \rangle &= \langle \{\mathbf{e}_v \mid v \in S\} \cup \{\mathbf{e}_{N(v)\setminus S} \mid v \in \mathcal{R}^b(\vec{xp})\} \rangle \\ &= \langle \{\mathbf{e}_v \mid v \in S\} \rangle + \langle \{\mathbf{e}_{N(v)\setminus S} \mid v \in \mathcal{R}^b(\vec{xp})\} \rangle = \langle \{\mathbf{e}_v \mid v \in S\} \rangle + A_{\vec{xp}}.\end{aligned}$$

Similarly,

$$\langle \mathcal{V} \setminus \mathcal{V}_x \rangle = \langle \{\mathbf{e}_v \mid v \in \overline{S}\} \rangle + \langle \{\mathbf{e}_{N(v)\cap S} \mid v \in \mathcal{R}^b(\vec{px})\} \rangle = \langle \{\mathbf{e}_v \mid v \in \overline{S}\} \rangle + A_{\vec{px}}.$$

Since $A_{\vec{xp}} \subseteq \langle \{\mathbf{e}_v \mid v \in \overline{S}\} \rangle$, $A_{\vec{px}} \subseteq \langle \{\mathbf{e}_v \mid v \in S\} \rangle$, and $\langle \{\mathbf{e}_v \mid v \in S\} \rangle \cap \langle \{\mathbf{e}_v \mid v \in \overline{S}\} \rangle = \{\mathbf{0}\}$, we conclude that $A_{\vec{xp}} \cap A_{\vec{px}} = \{\mathbf{0}\}$ and $B_x = \langle \mathcal{V}_x \rangle \cap \langle \mathcal{V} \setminus \mathcal{V}_x \rangle = A_{\vec{xp}} + A_{\vec{px}}$. □

Next, given a sequence of vectors $(\mathbf{v}_1, \ldots, \mathbf{v}_m)$ of a linear space, define the *lexicographically earliest basis* as the subsequence $(\mathbf{v}_{i_1}, \ldots, \mathbf{v}_{i_t})$ of $(\mathbf{v}_1, \ldots, \mathbf{v}_m)$ that is an ordered basis of $\langle \{\mathbf{v}_1, \ldots, \mathbf{v}_m\} \rangle$ with the property that the sequence $(i_1, \ldots, i_t)$ is lexicographically smallest possible.

We now define the canonical transcript $(\{\mathfrak{B}_x\}_{x \in V(T^b)}, \{\mathfrak{B}'_x\}_{x \in V(T^b)})$ of $\mathcal{T}^b$.

- For every $x \in V(T^b)$, define the canonical ordered basis $\mathfrak{B}_x$ of B_x as follows. If x is the root of T^b, then $\mathfrak{B}_x$ is empty. Otherwise, let p be the parent of x in T^b and $q \in [0, \ell]$ be the rank of the edge xp. Consider the sequence of vectors $(\mathbf{e}_{N(v)\setminus S})_{v \in \mathcal{R}^b(\vec{xp})}$ with indexes sorted by $<$; that is, $\mathbf{e}_{N(u)\setminus S}$ appears before $\mathbf{e}_{N(v)\setminus S}$ if and only if $u < v$. Then let $\mathfrak{B}_{\vec{xp}}$ be the lexicographically earliest basis of this sequence (so $\mathfrak{B}_{\vec{xp}}$ is an ordered basis of $A_{\vec{xp}}$). Also define the ordered basis $\mathfrak{B}_{\vec{px}}$ of $A_{\vec{px}}$ as the lexicographically earliest basis of the analogous sequence of vectors $(\mathbf{e}_{N(v)\cap S})_{v \in \mathcal{R}^b(\vec{px})}$. Now define $\mathfrak{B}_x$ as the concatenation of $\mathfrak{B}_{\vec{xp}}$ and $\mathfrak{B}_{\vec{px}}$.
- Then, for every $x \in V(T^b)$, define the canonical ordered basis $\mathfrak{B}'_x$ of B'_x as follows. If x is a leaf of T^b, then $\mathfrak{B}'_x = \mathfrak{B}_x$. Otherwise, let $c_1 < c_2$ be the two children of x in T^b and set $\mathfrak{B}'_x$ to the lexicographically earliest basis of the concatenation of the sequences $\mathfrak{B}_x$, $\mathfrak{B}_{c_1}$ and $\mathfrak{B}_{c_2}$.

It is easy to verify that $(\{\mathfrak{B}_x\}_{x \in V(T^b)}, \{\mathfrak{B}'_x\}_{x \in V(T^b)})$ is indeed a transcript of $\mathcal{T}^b$. For all nonroot nodes x of $\mathcal{T}^b$ with parent p, define $M_{\vec{xp}}$ as the transition matrix of x with respect to the canonical transcript of $\mathcal{T}^b$. Our aim now is to show that each transition matrix $M_{\vec{xp}}$ can be recovered from the annotations around p in $\mathcal{T}$. For the following statement, recall the definitions of the transition signature and the edge signature from Section 4.5.

Lemma 4.9.6. *Let $x \in V(T^b)$ be a nonroot node and p be the parent of x in T^b. Then, in time $\mathcal{O}_\ell(1)$, one can construct $M_{\vec{xp}}$ from:*

- *the transition signature $\tau(\mathcal{T}^b, \vec{pp'})$ if p is a nonroot node of T^b with parent p'; or*
- *the edge signatures $\sigma(\mathcal{T}^b, \vec{c_1 r}), \sigma(\mathcal{T}^b, \vec{c_2 r})$ if $p = r$ is the root of T^b with children $c_1 < c_2$.*

In the proof, we will use the following simple observation. We say that two sequences of vectors of equal length $\mathbf{v}_1, \ldots, \mathbf{v}_m \in \mathbb{F}^d$ and $\mathbf{v}'_1, \ldots, \mathbf{v}'_m \in \mathbb{F}^{d'}$ are *linearly equivalent* if for every sequence of coefficients $a_1, \ldots, a_m \in \mathbb{F}$, we have that $\sum_{i=1}^m a_i \mathbf{v}_i = \mathbf{0}$ if and only if $\sum_{i=1}^m a_i \mathbf{v}'_i = \mathbf{0}$. Note that in this case, $(\mathbf{v}_{i_1}, \ldots, \mathbf{v}_{i_t})$ is the lexicographically earliest basis of $(\mathbf{v}_1, \ldots, \mathbf{v}_m)$ if and only if $(\mathbf{v}'_{i_1}, \ldots, \mathbf{v}'_{i_t})$ is the lexicographically earliest basis of $(\mathbf{v}'_1, \ldots, \mathbf{v}'_m)$. Moreover, if $j \in [m]$ and $a_1, \ldots, a_t \in F$, then $\mathbf{v}_j = \sum_{k=1}^t a_k \mathbf{v}_{i_k}$ if and only if $\mathbf{v}'_j = \sum_{k=1}^t a_k \mathbf{v}'_{i_k}$. Then:

Observation 4.9.7. *Let $\mathbf{v}_1, \ldots, \mathbf{v}_m$ be vectors of the same vector space $\mathbb{F}^d$ and let $a \in [d]$. Suppose one of the following conditions holds:*

- *$(\mathbf{v}_i)_a = 0$ for every $i \in [m]$, i.e., in all vectors, the ath entry is zero; or*

- *there exists a different index $b \in [d]$ such that $(\mathbf{v}_i)_a = (\mathbf{v}_i)_b$ for every $i \in [m]$, i.e., in all vectors, the ath entry and the bth entry coincide.*

Let $\mathbf{v}'_1, \mathbf{v}'_2, \ldots, \mathbf{v}'_m$ be the vectors of $\mathbb{F}^{d-1}$ formed by dropping the ath coordinate from each vector $\mathbf{v}_i$. Then the sequences $(\mathbf{v}_1, \ldots, \mathbf{v}_m)$ and $(\mathbf{v}'_1, \ldots, \mathbf{v}'_m)$ are linearly equivalent.

Also we will use the following algorithmic tool which is an easy application of Gaussian elimination:

Lemma 4.9.8. *Let $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m$ be vectors of the same vector space $\mathbb{F}^d$. Then in time $\mathcal{O}(md^2)$ one can compute:*

- *the lexicographically earliest basis $(\mathbf{v}_{i_1}, \ldots, \mathbf{v}_{i_t})$ of $(v_1, \ldots, \mathbf{v}_m)$, and*
- *for every $j \in [m]$, the unique representation of $\mathbf{v}_j$ in this basis (i.e., the coefficients $a_{j,1}, \ldots, a_{j,t} \in \mathbb{F}$ such that $\mathbf{v}_j = \sum_{k=1}^{t} a_{j,k}\mathbf{v}_{i_k}$).*

Therefore, we quickly get that:

Lemma 4.9.9. *Let $x \in V(T^b)$ be a nonroot vertex of T^b with parent p, and let $S = \mathcal{L}(\mathcal{T}^b)[\vec{xp}] \subseteq V(G)$ and $q \in [0, \ell]$ be the width of xp. Then, given the sets $\mathcal{R}^b(\vec{xp}), \mathcal{R}^b(\vec{px})$ and the bipartite graph $\mathcal{E}^b(xp)$, in time $\mathcal{O}_q(1)$ one can compute the canonical ordered basis $\mathfrak{B}_x$, represented implicitly as two sequences of vertices $v_1^{\vec{xp}}, \ldots, v_q^{\vec{xp}} \in \mathcal{R}^b(\vec{xp})$ and $v_1^{\vec{px}}, \ldots, v_q^{\vec{px}} \in \mathcal{R}^b(\vec{px})$ such that*

$$\mathfrak{B}_x = (\mathbf{e}_{N(v_1^{\vec{xp}})\setminus S}, \ldots, \mathbf{e}_{N(v_q^{\vec{xp}})\setminus S}, \mathbf{e}_{N(v_1^{\vec{px}})\cap S}, \ldots, \mathbf{e}_{N(v_q^{\vec{px}})\cap S}).$$

Proof. Recall that $\mathfrak{B}_x$ is the concatenation of $\mathfrak{B}_{\vec{xp}}$ and $\mathfrak{B}_{\vec{px}}$. Here, we only show how to compute $\mathfrak{B}_{\vec{xp}}$; the latter is determined analogously. Let $\mathcal{R}^b(\vec{xp}) = \{v_1 < v_2 < \cdots < v_m\}$, where $m = |\mathcal{R}^b(\vec{xp})| \leqslant 2^q$. Then $\mathfrak{B}_{\vec{xp}}$ is defined as the lexicographically earliest basis of the sequence $(\mathbf{e}_{N(v_1)\setminus S}, \ldots, \mathbf{e}_{N(v_m)\setminus S})$. By a repeated application of Observation 4.9.7, we produce a linearly equivalent sequence of vectors $(\mathbf{u}'_1, \ldots, \mathbf{u}'_m)$ by dropping from each vector of this sequence the coordinates corresponding to the vertices $u \in V(G)$ such that:

- $u \in S$ (since $(\mathbf{e}_{N(v_i)\setminus S})_u = 0$ for all $i \in [m]$); or
- $u \notin S$ and $u \notin \mathcal{R}^b(\vec{px})$ (since then there exists a representative $u' \in \mathcal{R}^b(\vec{px})$ of u such that $N(u) \cap S = N(u') \cap S$, or equivalently, $(\mathbf{e}_{N(v_i)\setminus S})_u = (\mathbf{e}_{N(v_i)\setminus S})_{u'}$ for all $i \in [m]$).

In other words, let $(\mathbf{u}'_1, \ldots, \mathbf{u}'_m)$ be the sequence of vectors in $\mathbb{F}^{|\mathcal{R}^b(\vec{px})|}$, where $\mathbf{u}'_i$ is constructed from $\mathbf{e}_{N(v_i)\setminus S}$ by dropping all coordinates not corresponding to the vertices of $\mathcal{R}^b(\vec{px})$. This sequence can be constructed explicitly in time $\mathcal{O}_q(1)$ using $\mathcal{R}^b(\vec{xp})$, $\mathcal{R}^b(\vec{px})$ and $\mathcal{E}^b(xp)$. Using Lemma 4.9.8, we find the lexicographically earliest basis $(\mathbf{u}'_{i_1}, \ldots, \mathbf{u}'_{i_q})$ of $(\mathbf{u}'_1, \ldots, \mathbf{u}'_m)$. Then by Observation 4.9.7, we have that $\mathfrak{B}_{\vec{xp}} = (\mathbf{e}_{N(v_{i_1})\setminus S}, \ldots, \mathbf{e}_{N(v_{i_q})\setminus S})$. □

We are now ready to prove Lemma 4.9.6.

Proof of Lemma 4.9.6. First suppose that p is the parent of x and p' is the parent of p in T^b. Let also $x^\star$ be the sibling of x, i.e., the other child of p in T^b. We showcase the proof in the case where $x < x^\star$, but the case $x > x^\star$ is analogous.

Recall that $\mathfrak{B}'_p$ is the lexicographically earliest basis of the concatenation of $\mathfrak{B}_p$, $\mathfrak{B}_x$ and $\mathfrak{B}_{x^\star}$ (where $\mathfrak{B}_p$ is the concatenation of $\mathfrak{B}_{\vec{pp'}}$ and $\mathfrak{B}_{\vec{p'p}}$; $\mathfrak{B}_x$ is the concatenation of $\mathfrak{B}_{\vec{xp}}$ and $\mathfrak{B}_{\vec{px}}$; and $\mathfrak{B}_{x^\star}$ is the concatenation of $\mathfrak{B}_{\vec{x^\star p}}$ and $\mathfrak{B}_{\vec{px^\star}}$). Note that the transition signature $\tau(\mathcal{T}^b, \vec{pp'})$ contains the representative sets $\mathcal{R}^b(\vec{xp})$, $\mathcal{R}^b(\vec{px})$, $\mathcal{R}^b(\vec{x^\star p})$, $\mathcal{R}^b(\vec{px^\star})$, $\mathcal{R}^b(\vec{pp'})$, $\mathcal{R}^b(\vec{p'p})$ and the bipartite graphs $\mathcal{E}^b(xp)$, $\mathcal{E}^b(x^\star p)$, $\mathcal{E}^b(pp')$, so we can use Lemma 4.9.9 to compute the implicit representations of each $\mathfrak{B}_p, \mathfrak{B}_x, \mathfrak{B}_{x^\star}$ in time $\mathcal{O}_\ell(1)$. Let S be the concatenation of these ordered bases. Let also S' be the sequence of vectors produced from S by dropping all the coordinates corresponding to vertices outside of $\mathcal{R}^b(\vec{xp}) \cup \mathcal{R}^b(\vec{x^\star p}) \cup \mathcal{R}^b(\vec{p'p})$.

Claim 4.9.10. *S and S' are linearly equivalent.*

Proof of the claim. Note that $V(G)$ is a disjoint union of $\mathcal{L}(\mathcal{T}^b)[\vec{xp}]$, $\mathcal{L}(\mathcal{T}^b)[\vec{x^\star p}]$ and $\mathcal{L}(\mathcal{T}^b)[\vec{p'p}]$. First suppose that $u \in \mathcal{L}(\mathcal{T}^b)[\vec{xp}]$, but $u \notin \mathcal{R}^b(\vec{xp})$. Then there exists a representative $u' \in \mathcal{R}^b(\vec{xp})$ such that $N(u) \cap \mathcal{L}(\mathcal{T}^b)[\vec{px}] = N(u') \cap \mathcal{L}(\mathcal{T}^b)[\vec{p'x}]$. We now claim that for every vector $\mathbf{v} \in S$, we have $\mathbf{v}_u = \mathbf{v}_{u'}$.

- If $\mathbf{v}$ belongs to the ordered basis $\mathfrak{B}_{x\vec{p}}$ (i.e., $\mathbf{v}$ is implicitly represented by a vertex of $\mathcal{R}^b(x\vec{p})$), then by definition $\mathbf{v}_s = 0$ for all $s \in \mathcal{L}(\mathcal{T}^b)[x\vec{p}]$. Hence $\mathbf{v}_u = \mathbf{v}_{u'} = 0$.
- Similarly, if $\mathbf{v}$ belongs to $\mathfrak{B}_{p\vec{p'}}$ (resp. $\mathfrak{B}_{p\vec{x^\star}}$), then the same argument follows from the fact that $\mathcal{L}(\mathcal{T}^b)[x\vec{p}]$ is a subset of $\mathcal{L}(\mathcal{T}^b)[p\vec{p'}]$ (resp. $\mathcal{L}(\mathcal{T}^b)[p\vec{x^\star}]$). So $\mathbf{v}_u = \mathbf{v}_{u'} = 0$.
- If $\mathbf{v}$ belongs to any of the ordered bases $\mathfrak{B}_{p\vec{x}}$, $\mathfrak{B}_{p'\vec{p}}$ or $\mathfrak{B}_{x^{\vec{\star}}p}$, then $\mathbf{v}_u = \mathbf{v}_{u'}$ follows from $N(u) \cap \mathcal{L}(\mathcal{T}^b)[p\vec{x}] = N(u') \cap \mathcal{L}(\mathcal{T}^b)[p'\vec{x}]$ and the fact that $\mathcal{L}(\mathcal{T}^b)[p\vec{x}]$ is a superset of both $\mathcal{L}(\mathcal{T}^b)[p'\vec{p}]$ and $\mathcal{L}(\mathcal{T}^b)[x^{\vec{\star}}p]$.

By case exhaustion we conclude that $\mathbf{v}_u = \mathbf{v}_{u'}$ for all vectors $\mathbf{v} \in S$, and so Observation 4.9.7 applies and the coordinate corresponding to the vertex u can be removed from all vectors of S while maintaining the linear equivalence. A symmetric proof for $u \in \mathcal{L}(\mathcal{T}^b)[x^{\vec{\star}}p] \setminus \mathcal{R}^b(x^{\vec{\star}}p)$ and $u \in \mathcal{L}(\mathcal{T}^b)[p'\vec{p}] \setminus \mathcal{R}^b(p'\vec{p})$ settles the claim. ◁

Now observe that S' can be constructed explicitly in time $\mathcal{O}_\ell(1)$: It is enough to determine, for each $e_1 \in \{x\vec{p}, p\vec{x}, x^{\vec{\star}}p, p\vec{x^\star}, p\vec{p'}, p'\vec{p}\}$ and a vector $\mathbf{u}$ in $\mathfrak{B}_{e_1}$ (implicitly represented by a vertex $u \in \mathcal{R}^b(e_1)$), and for each $e_2 \in \{x\vec{p}, x^{\vec{\star}}p, p'\vec{p}\}$ and $v \in \mathcal{R}^b(e_2)$, the value of $\mathbf{u}_v$. It can be easily observed that this value is equal to 1 if and only if e_1 and e_2 point towards each other (i.e., e_1 is a predecessor of e_2', where e_2' is the edge e_2 with its head and tail swapped), and $uv \in E(G)$. Both of these conditions can be easily verified using the transition signature of $p\vec{p'}$ in $\mathcal{T}^b$.

We now run the algorithm of Lemma 4.9.8 to find the lexicographically earliest basis $\mathfrak{B}_{S'}$ of S' in time $\mathcal{O}_\ell(1)$; moreover, this algorithm provides, for each vector $\mathbf{v} \in S'$, the representation of $\mathbf{v}$ in this basis. Since S and S' are linearly equivalent and $\mathfrak{B}'_p$ is the lexicographically earliest basis of S, we can easily recover, for each vector $\mathbf{v} \in \mathfrak{B}_x$, the representation of $\mathbf{v}$ in $\mathfrak{B}'_p$. These representations form the $|\mathfrak{B}'_p| \times |\mathfrak{B}_x|$ transition matrix $M_{x\vec{p}}$.

We now briefly discuss the case where $p = r$ is the root of T^b with two children $c_1 < c_2$ (so that $x \in \{c_1, c_2\}$). Note that $B_r = \{\mathbf{0}\}$, so $\mathfrak{B}_r$ is empty; moreover, $B_{c_1} = B_{c_2} = \langle \mathcal{V}_{c_1} \rangle \cap \langle \mathcal{V}_{c_2} \rangle$ and hence $\mathfrak{B}'_r = \mathfrak{B}_{c_1}$. The implicit representations of both $\mathfrak{B}_{c_1}$ and $\mathfrak{B}_{c_2}$ can be deduced from the edge signatures $\sigma(\mathcal{T}^b, c_1\vec{r})$, $\sigma(\mathcal{T}^b, c_2\vec{r})$ in time $\mathcal{O}_\ell(1)$ using Lemma 4.9.9. Thus both $M_{c_1\vec{r}}$ – the identity matrix of dimension $|\mathfrak{B}_{c_1}|$ – and $M_{c_2\vec{r}}$ – the transition matrix from the basis $\mathfrak{B}_{c_2}$ to $\mathfrak{B}_{c_1}$ – can be computed using only $\sigma(\mathcal{T}^b, c_1\vec{r})$ and $\sigma(\mathcal{T}^b, c_2\vec{r})$ in time $\mathcal{O}_\ell(1)$. □

Construction of the rank decomposition automaton. We have now gathered enough tools to prove the following statement.

Lemma 4.9.11. *Let $k, \ell \geqslant 0$ be integers with $k \leqslant \ell$. There exists a label-oblivious rank decomposition automaton $\mathcal{JKO}_{k,\ell} = (Q, \iota, \delta, \varepsilon)$ of width ℓ with evaluation time $\mathcal{O}_\ell(1)$ and $|Q| = \mathcal{O}_{k,\ell}(1)$, called the* exact rankwidth automaton*, with the following properties:*

Suppose that $\mathcal{T}^b$ is a rooted annotated rank decomposition of width at most ℓ that encodes a partitioned graph $(G, \mathcal{C})$. Let ρ be the run of $\mathcal{A}$ on $\mathcal{T}^b$. Then, for every $x \in V(\mathcal{T}^b)$, the full set $\mathrm{FS}_k^{\mathfrak{B}_x}(x)$ at x of width k with respect to $\mathcal{T}^b$ is equal to:

- *$\rho(x\vec{p})$ if x is not the root of T^b and p is the parent of x; or*
- *$\rho(\vartheta)$ if x is the root of T^b.*

Proof. Let $Q = 2^{U_k^\ell}$, i.e., every state in Q is a subfamily of the family of all possible encodings of compact B-namus of width at most k in an ordered basis of size at most ℓ. Note that since $\mathcal{T}^b$ has width at most ℓ, we get that for any node $x \in V(T^b)$ and any ordered basis $\mathfrak{B}_x$ of the boundary space B_x, we have $\mathrm{FS}_k^{\mathfrak{B}_x}(x) \subseteq U_k^\ell$ and so $\mathrm{FS}_k^{\mathfrak{B}_x}(x) \in Q$.

We define the initial mapping ι so that, for any leaf edge $l\vec{p} \in \vec{L}(T^b)$, we have that $\rho(l\vec{p}) = \mathrm{FS}_k^{\mathfrak{B}_l}(l)$. By Lemma 4.9.3, $\mathrm{FS}_k^{\mathfrak{B}_l}(l)$ only depends on the cardinality of $\mathfrak{B}_l$, which can be uniquely deduced from the edge signature $\sigma(\mathcal{T}, l\vec{p})$. Since ι accepts a leaf edge signature as an argument, such an initial mapping can be constructed.

The transition mapping δ is constructed as follows. Suppose x is not a leaf nor a root of $\mathcal{T}^b$ and let p be the parent of x in $\mathcal{T}^b$. Let also $c_1 < c_2$ be the two children of x in T^b. Then, we compute $\rho(x\vec{p}) = \mathrm{FS}_k^{\mathfrak{B}_x}(x)$ as follows. By Lemma 4.9.3, $\mathrm{FS}_k^{\mathfrak{B}_x}(x)$ can be deduced uniquely from $\mathrm{FS}_k^{\mathfrak{B}_{c_1}}(c_1)$, $\mathrm{FS}_k^{\mathfrak{B}_{c_2}}(c_2)$, $M_{c_1\vec{x}}$, $M_{c_2\vec{x}}$ and $|\mathfrak{B}_x|$. From Lemma 4.9.6 it follows that both $M_{c_1\vec{x}}$ and $M_{c_2\vec{x}}$ can be determined from the transition

signature $\tau(\mathcal{T}^b, \vec{xp})$. Also $|\mathfrak{B}_x|$ can be quickly deduced from the transition signature. On the other hand, $\mathrm{FS}_k^{\mathfrak{B}_{c_1}}(c_1)$ is simply $\rho(\vec{c_1x})$ and $\mathrm{FS}_k^{\mathfrak{B}_{c_2}}(c_2)$ is $\rho(\vec{c_2x})$. So we define the transition mapping δ so that $\mathrm{FS}_k^{\mathfrak{B}_x}(x) = \delta(\tau(\mathcal{T}^b, \vec{xp}), \mathrm{FS}_k^{\mathfrak{B}_{c_1}}(c_1), \mathrm{FS}_k^{\mathfrak{B}_{c_2}}(c_2))$.

For the final mapping ε, let $x = r$ be the root of T^b with children $c_1 < c_2$. Our aim is to determine $\rho(\vartheta) = \mathrm{FS}_k^{\mathfrak{B}_r}(r)$ from $\rho(\vec{c_1r}) = \mathrm{FS}_k^{\mathfrak{B}_{c_1}}(c_1)$, $\rho(\vec{rc_1}) = \rho(\vec{c_2r}) = \mathrm{FS}_k^{\mathfrak{B}_{c_2}}(c_2)$ and the edge signature $\sigma(\mathcal{T}^b, \vec{c_1r})$. By the definitions of runs of automata on rooted trees, the edge signature $\sigma(\mathcal{T}^b, \vec{c_2r})$ is uniquely determined by $\sigma(\mathcal{T}^b, \vec{c_1r})$. Again by Lemma 4.9.3, $\mathrm{FS}_k^{\mathfrak{B}_r}(r)$ can be deduced uniquely from $\mathrm{FS}_k^{\mathfrak{B}_{c_1}}(c_1)$, $\mathrm{FS}_k^{\mathfrak{B}_{c_2}}(c_2)$, $M_{\vec{c_1r}}$, $M_{\vec{c_2r}}$ and $|\mathfrak{B}_r| = 0$. And by Lemma 4.9.6, $M_{\vec{c_1r}}$ and $M_{\vec{c_2r}}$ can be computed in $\mathcal{O}_\ell(1)$ time given $\sigma(\mathcal{T}^b, \vec{c_1r})$ and $\sigma(\mathcal{T}^b, \vec{c_2r})$. Thus we define ε so that $\mathrm{FS}_k^{\mathfrak{B}_r}(r) = \varepsilon(\delta(\mathcal{T}^b, \vec{c_1r}), \mathrm{FS}_k^{\mathfrak{B}_{c_1}}(c_1), \mathrm{FS}_k^{\mathfrak{B}_{c_2}}(c_2))$.

Since ι, δ and ε can be computed from its arguments in time $\mathcal{O}_{k,\ell}(1)$, the proof is complete. □

Combining Lemma 4.9.11 with Lemma 4.9.4, we immediately obtain the following lemma.

Lemma 4.9.12. *Let $k, \ell \geqslant 0$ be integers. There exists an algorithm that, given as input an annotated rank decomposition $\mathcal{T}$ of width ℓ that encodes a partitioned graph $(G, \mathcal{C})$, in time $\mathcal{O}_\ell(|\mathcal{T}|)$ either:*

- *correctly determines that $(G, \mathcal{C})$ has rankwidth larger than k; or*
- *outputs a (non-annotated) rank decomposition of $(G, \mathcal{C})$ of width at most k.*

Which then by combining with Lemma 4.3.8 implies Lemma 4.6.2, which we restate here.

Lemma 4.6.2. *Let $k, \ell \geqslant 0$ be integers. There exists an algorithm that, given as input an annotated rank decomposition $\mathcal{T}$ of width ℓ that encodes a partitioned graph $(G, \mathcal{C})$, in time $\mathcal{O}_\ell(|\mathcal{T}| \log |\mathcal{T}|)$ either:*

- *correctly determines that $(G, \mathcal{C})$ has rankwidth larger than k; or*
- *outputs an annotated rank decomposition that encodes $(G, \mathcal{C})$ and has width at most k.*

4.9.2 Closure automaton

We move on to the description of another rank decomposition automaton – an automaton computing possible small closures within the subtrees of a given rank decomposition. This automaton, together with $\mathcal{JKO}$ from Lemma 4.9.11, will be used by us in the proof of Lemma 4.4.7. The description below should be considered to be an analog of a similar closure automaton for treewidth (Section 3.7.2). However, this construction of the automaton is noticeably more involved here: In the case of treewidth, it was enough to maintain, for each subtree T of the decomposition, a bounded-size family of small subsets of $V(G)$ (so the description of each subtree T simply had bounded size and could be manipulated explicitly). Here, given an annotated rank decomposition $\mathcal{T}$ of G, we will need to store, for each edge $\vec{xp} \in \vec{E}(\mathcal{T})$, a bounded-size family of *partitions* of $V(\mathcal{T})[\vec{xp}]$ into a small number of subsets. Since we cannot store the partitions of $V(\mathcal{T})[\vec{xp}]$ explicitly in an efficient manner, we first need to roll out a way of encoding such partitions succinctly. Intuitively, given a partition $\mathcal{C}$ of $V(\mathcal{T})[\vec{xp}]$, we want to select from each set $C \in \mathcal{C}$ a minimal representative R_C of C and encode the connections between R_C and $\overline{C}$ in G. The details follow below.

Let X be a nonempty finite set. We define an *indexed partition* of X as any sequence $\mathbb{C} = (X_1, \ldots, X_c)$ of (possibly empty) pairwise disjoint subsets of X with $X_1 \cup \ldots \cup X_c = X$. Then $\mathbb{C}$ is said to *represent* the (non-indexed) partition $\mathcal{C} = \{X_1, \ldots, X_c\} \setminus \{\emptyset\}$ of X.

Next, fix $c \in \mathbb{N}$. We say that a triple $\mathbb{H} = ((V_1, \ldots, V_c), H, \eta)$ is a (c, X)-*indexed graph* if:

- H is an undirected graph,
- $(V_1, \ldots, V_c)$ is an indexed partition of $V(H)$;
- for every $i \in [c]$, the subgraph $H[V_i]$ is edgeless; and
- $\eta : V(H) \to X$ is a labeling function.

Given a (c, X)-indexed graph $\mathbb{H} = ((V_1, \ldots, V_c), H, \eta)$, we define the *derived* partitioned graph $(H, \mathcal{D})$ by setting $\mathcal{D} = \{V_1, \ldots, V_c\} \setminus \{\emptyset\}$. Also, for convenience, define $V(\mathbb{H}) := V(H)$, $E(\mathbb{H}) := E(H)$, $G(\mathbb{H}) := H$, $V_i(\mathbb{H}) := V_i$ and $\eta(\mathbb{H}) := \eta$.

Two (c, X)-indexed graphs $\mathbb{H}_1 = ((V_1^1, \ldots, V_c^1), H_1, \eta_1)$, $\mathbb{H}_2 = ((V_1^2, \ldots, V_c^2), H_2, \eta_2)$ are isomorphic (denoted $\mathbb{H}_1 \sim^{c,X} \mathbb{H}_2$) if there exists an isomorphism $\pi : V(H_1) \to V(H_2)$ from H_1 to H_2 such that: (i) $\pi(V_i^1) = V_i^2$ for all $i \in [c]$, and (ii) $\eta_1(v) = \eta_2(\pi(v))$ for all $v \in V(H_1)$.

For $s \in \mathbb{N}$, we say that $\mathbb{H} = ((V_1, \ldots, V_c), H, \eta)$ is *s-small* if for every $i \in [c]$ and $x \in X$, we have $|V_i \cap \eta^{-1}(x)| \leqslant s$; i.e., each subset V_i contains at most s vertices of any given label. Thus if $\mathbb{H}$ is an s-small (c, X)-indexed graph, then $|V(\mathbb{H})| \leqslant cs|X|$. Note that the property of s-smallness of indexed graphs is preserved by isomorphism. Hence we define $\sim_s^{c,X}$ as the restriction of $\sim^{c,X}$ to only the classes containing s-small indexed graphs. It is easy to see that $\sim_s^{c,X}$ has $\mathcal{O}_{c,|X|,s}(1)$ distinct equivalence classes.

Now suppose that a graph G is encoded by an annotated rank decomposition $\mathcal{T} = (T, U, \mathcal{R}, \mathcal{E}, \mathcal{F})$ of width ℓ and let $\vec{xp} \in \vec{E}(T)$. Recall that $\mathcal{L}(\mathcal{T})[\vec{xp}]$ comprises the vertices of G assigned to the leaf edges of T that are closer to x than p, and that $\mathcal{R}(\vec{xp})$ is a minimal representative of $\mathcal{L}(\mathcal{T})[\vec{xp}]$ in G. We say that a $(c, \mathcal{R}(\vec{xp}))$-indexed graph $\mathbb{H} = ((V_1, \ldots, V_c), H, \eta)$ *respects $\mathcal{T}$ along $\vec{xp}$* if:

- $H = G[\{V_1, \ldots, V_c\}]$; and
- for each $v \in V(H)$, the label $\eta(v)$ is the unique vertex in $\mathcal{R}(\vec{xp})$ so that $N_G(v) \cap \mathcal{L}(\mathcal{T})[\vec{px}] = N_G(\eta(v)) \cap \mathcal{L}(\mathcal{T})[\vec{px}]$.

Observe that if the graph G and the decomposition $\mathcal{T}$ is fixed, then both the graph H and the labeling function η of an indexed graph respecting $\mathcal{T}$ along $\vec{xp}$ only depend on the choice of the sets $V_1, \ldots, V_c$.

Assuming $\mathbb{H} = ((V_1, \ldots, V_c), H, \eta)$ respects $\mathcal{T}$ along $\vec{xp}$, we say that it *encodes* an indexed partition $\mathbb{C} = (X_1, \ldots, X_c)$ of $\mathcal{L}(\mathcal{T})[\vec{xp}]$ if V_i is a minimal representative of X_i in G for each $i \in [c]$. It is straightforward to see that all indexed graphs encoding $\mathbb{C}$ are pairwise isomorphic: For each i the collection of neighborhoods $\{N(v) \setminus X_i\}_{v \in X_i}$ is uniquely determined by X_i, so V_i contains one vertex $v \in X_i$ for each distinct neighborhood $N(v) \setminus X_i$; and the resulting indexed graph is the same up to isomorphism regardless of the choice of v. Also, we say that $\mathbb{H}$ encodes a partition $\mathcal{C}$ if $\mathbb{H}$ encodes some indexed partition $\mathbb{C}$ representing $\mathcal{C}$.

If $\mathcal{C}$ is a partition of $\mathcal{L}(\mathcal{T})[\vec{xp}]$, then we define its *cost* to be the number of nodes in the subtree rooted at $\vec{xp}$ that are cut by $\mathcal{C}$; i.e., the number of oriented edges $\vec{e}$ that are predecessors of $\vec{xp}$ in T such that $\mathcal{L}(\mathcal{T})[\vec{e}]$ intersects more than one set of $\mathcal{C}$. We similarly define the cost of indexed partitions of $\mathcal{L}(\mathcal{T})[\vec{xp}]$.

Finally, for every equivalence class $\mathcal{K}$ of $\sim_s^{c,\mathcal{R}(\vec{xp})}$, let $A_{\mathcal{K}}$ be the set of pairs $(q, \mathbb{H})$, where $\mathbb{H} \in \mathcal{K}$ is an s-small $(c, \mathcal{R}(\vec{xp}))$-indexed graph encoding some partition $\mathcal{C}$ of $\mathcal{L}(\mathcal{T})[\vec{xp}]$ of cost q. Then we say that a set F is a *set of (c, s)-small representatives of $\mathcal{T}$ along $\vec{xp}$* if, for every equivalence class $\mathcal{K}$ of $\sim_s^{c,\mathcal{R}(\vec{xp})}$ with $A_{\mathcal{K}} \neq \emptyset$, F contains a single pair $(q, \mathbb{H}) \in A_{\mathcal{K}}$ with the minimum cost q. Note that the cardinality of F is bounded by the number of equivalence classes $\sim_s^{c,\mathcal{R}(\vec{xp})}$, which is bounded by $\mathcal{O}_{c,s,\ell}(1)$.

Our aim is now to prove that a rank decomposition automaton can compute, for each edge $\vec{xp}$, some set of (c, s)-small representatives of $\mathcal{T}$ along $\vec{xp}$ – which we will call $\mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp})$ from now on – and additional annotations allowing us to efficiently recover, for each $(q, \mathbb{H}) \in \mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp})$, an indexed partition of $\mathcal{L}(\mathcal{T})[\vec{xp}]$ of cost q encoded by $\mathbb{H}$.

Lemma 4.9.13. *For every triple of nonnegative integers c, s, ℓ, there exists a label-oblivious rank decomposition automaton $\mathcal{CR} = \mathcal{CR}_{c,s,\ell}$ with evaluation time $\mathcal{O}_{c,s,\ell}(1)$ with the following property. Suppose G is a graph encoded by an annotated rank decomposition $\mathcal{T} = (T, U, \mathcal{R}, \mathcal{E}, \mathcal{F})$ of width at most ℓ. Then the run ρ of $\mathcal{CR}$ on $\mathcal{T}$ satisfies that for every $\vec{xp} \in \vec{E}(\mathcal{T})$,*

$$\rho(\vec{xp}) = (\mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp}), \Phi),$$

where $\mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp})$ is a set of (c, s)-small representatives of $\mathcal{T}$ along $\vec{xp}$, and Φ is a mapping from $\mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp})$ such that:

- *if $\vec{xp}$ is a leaf oriented edge, then Φ maps each pair in $\mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp})$ to $\bot$; and*
- *if $\vec{xp}$ is a nonleaf oriented edge, where $\vec{xp}$ has two children $y_1\vec{x}$ and $y_2\vec{x}$, then for every $(q, \mathbb{H}) \in \mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp})$, we have $\Phi((q, \mathbb{H})) = ((q_1, \mathbb{H}_1), (q_2, \mathbb{H}_2))$ such that:*
  - *$(q_t, \mathbb{H}_t) \in \mathsf{reps}^{c,s}(\mathcal{T}, y_t\vec{x})$ for each $t \in [2]$;*
  - *for every indexed partition $(X_1^1, \ldots, X_c^1)$ of $\mathcal{L}(\mathcal{T})[y_1\vec{x}]$ of cost q_1 encoded by $\mathbb{H}_1$, and every indexed partition $(X_1^2, \ldots, X_c^2)$ of $\mathcal{L}(\mathcal{T})[y_2\vec{x}]$ of cost q_2 encoded by $\mathbb{H}_2$, the indexed partition $(X_1^1 \cup X_1^2, \ldots, X_c^1 \cup X_c^2)$ of $\mathcal{L}(\mathcal{T})[\vec{xp}]$ has cost q and is encoded by $\mathbb{H}$.*

Proof. We need to implement the following two procedures:

- for a leaf oriented edge $\vec{lp}$ of T with edge signature $\sigma(\mathcal{T}, \vec{lp})$, determine $\mathsf{reps}^{c,s}(\mathcal{T}, \vec{lp})$; and

- for a nonleaf oriented edge $\vec{xp}$ of T where $\vec{xp}$ has two children $\vec{y_1x}$, $\vec{y_2x}$, find $\mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp})$ and the mapping Φ as in the statement of the lemma, given $\mathsf{reps}^{c,s}(\mathcal{T}, \vec{y_1x})$, $\mathsf{reps}^{c,s}(\mathcal{T}, \vec{y_2x})$ and the transition signature $\tau(\mathcal{T}, \vec{xp})$. Here we inductively assume that for $t \in [2]$, $\mathsf{reps}^{c,s}(\mathcal{T}, \vec{y_tx})$ is a set of (c,s)-small representatives of $\mathcal{T}$ along $\vec{y_tx}$.

First, for a leaf edge $\vec{lp}$, observe that $|\mathcal{L}(\mathcal{T})[\vec{lp}]| = 1$ and the only vertex $v \in \mathcal{L}(\mathcal{T})[\vec{lp}]$ can be read from the edge signature $\sigma(\mathcal{T}, \vec{lp})$. Thus there exist exactly c nonisomorphic $(c, \mathcal{R}(\vec{lp}))$-indexed graphs $\mathbb{H}_1, \ldots, \mathbb{H}_c$ respecting $\mathcal{T}$ along $\vec{lp}$ and encoding an indexed partition of $\mathcal{L}(\mathcal{T})[\vec{lp}]$: For each $i \in [c]$, the indexed graph $\mathbb{H}_i$ is defined by the sequence of sets $(V_1^i, \ldots, V_c^i)$, where $V_i^i = \{v\}$ and $V_j^i = \emptyset$ for $j \neq i$. Moreover, $E(\mathbb{H}_i) = \emptyset$ and $\eta(\mathbb{H}_i)(v) = v$. Naturally, each $\mathbb{H}_i$ encodes a partition of $\mathcal{L}(\mathcal{T})[\vec{lp}]$ of cost 0. Hence $\mathsf{reps}^{c,s}(\mathcal{T}, \vec{lp})$ can be enumerated by brute force in time $\mathcal{O}_{c,s,\ell}(1)$.

Now assume $\vec{xp}$ is a nonleaf oriented edge and let $\vec{y_1x}$ and $\vec{y_2x}$ be the two children of $\vec{xp}$. For convenience, define $S_1 = \mathcal{L}(\mathcal{T})[\vec{y_1x}]$, $S_2 = \mathcal{L}(\mathcal{T})[\vec{y_2x}]$, and $S = \mathcal{L}(\mathcal{T})[\vec{xp}]$; we have that $S_1 \cap S_2 = \emptyset$ and $S = S_1 \cup S_2$.

We now define a function $\mathsf{Combine}(\mathbb{H}_1, \mathbb{H}_2)$, taking as arguments a $(c, \mathcal{R}(\vec{y_1x}))$-indexed graph $\mathbb{H}_1$ respecting $\mathcal{T}$ along $\vec{y_1x}$, and a $(c, \mathcal{R}(\vec{y_2x}))$-indexed graph $\mathbb{H}_2$ respecting $\mathcal{T}$ along $\vec{y_2x}$ and returning a $(c, \mathcal{R}(\vec{xp}))$-indexed graph $\mathbb{H}$ respecting $\mathcal{T}$ along $\vec{xp}$ as follows. Let us denote the input graphs by $\mathbb{H}_1 = ((V_1^1, \ldots, V_c^1), H_1, \eta_1)$ and $\mathbb{H}_2 = ((V_1^2, \ldots, V_c^2), H_2, \eta_2)$. Note that $V(H_1) \cap V(H_2) = \emptyset$. Define an auxiliary $(c, \mathcal{R}(\vec{xp}))$-indexed graph $\mathbb{H}' = ((V_1', \ldots, V_c'), H', \eta')$ as follows:

- $V_i' = V_i^1 \cup V_i^2$ for each $i \in [c]$;
- $V(H') = V(H_1) \cup V(H_2)$;
- $H'[V(H_1)] = H_1$ and $H'[V(H_2)] = H_2$;
- for $u \in V(H_1)$ and $v \in V(H_2)$, we have $uv \in E(H')$ if and only if u, v do not belong to the same set V_i' and moreover $\eta_1(u)\eta_2(v) \in E(G)$; and
- for $t \in [2]$ and $v \in V(H_t)$, we have $\eta'(v) = \mathcal{F}(y_ixp)(\eta(v))$.

A verification with the definitions shows that $\mathbb{H}'$ respects $\mathcal{T}$ along $\vec{xp}$. In particular, whenever $i \in [c]$ and $u, v \in V_i(\mathbb{H}')$ with $\eta'(u) = \eta'(v)$, we have that $N_G(u) \cap \mathcal{L}(\mathcal{T})[p\vec{x}] = N_G(v) \cap \mathcal{L}(\mathcal{T})[p\vec{x}]$. Also, $\mathsf{Combine}(\mathbb{H}_1, \mathbb{H}_2)$ can be constructed given $\mathbb{H}_1$ and $\mathbb{H}_2$ using only the transition signature $\tau(\mathcal{T}, \vec{xp})$. In particular, for $u \in V(H_1), v \in V(H_2)$, we have $\eta_1(u) \in \mathcal{R}(\vec{y_1x})$, $\eta_2(v) \in \mathcal{R}(\vec{y_2x})$, so whether $\eta_1(u)\eta_2(v) \in E(G)$ depends only on $\tau(\mathcal{T}, \vec{xp})$.

Then $\mathbb{H}$ is constructed from $\mathbb{H}'$ as follows. We begin with $\mathbb{H} = \mathbb{H}'$. Whenever there is an index $i \in [c]$ and two vertices $u, v \in V_i(\mathbb{H})$ such that $N_{H'}(u) = N_{H'}(v)$ and $\eta'(u) = \eta'(v)$, we remove one of the vertices from $V_i(\mathbb{H})$ (and therefore $\mathbb{H}$).

We now prove a string of properties of $\mathsf{Combine}$:

Claim 4.9.14. *Whenever $\mathbb{H}_1$ encodes an indexed partition $(X_1^1, \ldots, X_c^1)$ of S_1 and $\mathbb{H}_2$ encodes an indexed partition $(X_1^2, \ldots, X_c^2)$ of S_2, then $\mathsf{Combine}(\mathbb{H}_1, \mathbb{H}_2)$ encodes the indexed partition $(X_1^1 \cup X_1^2, \ldots, X_c^1 \cup X_c^2)$ of S.*

Proof of the claim. Take $\mathbb{H}_1 = ((V_1^1, \ldots, V_c^1), H_1, \eta_1)$, $\mathbb{H}_2 = ((V_1^2, \ldots, V_c^2), H_2, \eta_2)$ and define $\mathbb{H} := \mathsf{Combine}(\mathbb{H}_1, \mathbb{H}_2) = ((V_1, \ldots, V_c), H, \eta)$. Let also $\mathbb{H}' = ((V_1', \ldots, V_c'), H', \eta')$ be the auxiliary graph in the definition of $\mathsf{Combine}$. Since $\mathbb{H}'$ respects $\mathcal{T}$ along $\vec{xp}$ and $\mathbb{H}$ is an induced subgraph of $\mathbb{H}'$ (i.e., $V_i \subseteq V_i'$ for all $i \in [c]$), we find that also $\mathbb{H}$ respects $\mathcal{T}$ along $\vec{xp}$. Finally define $X_i = X_i^1 \cup X_i^2$ for $i \in [p]$.

First consider two vertices $u, v \in V(H')$ with $u \in V_i'$, $v \in V_j'$ and $i \neq j$. We will show that $uv \in E(H')$ if and only if $uv \in E(G)$. If $u \in V_i^t$ and $v \in V_j^t$ for some $t \in [2]$, this follows from the fact that $\mathbb{H}_t$ respects $\mathcal{T}$ along $c_t\vec{x}$: We have $H_t = G[\{V_1^t, \ldots, V_c^t\}]$, so $uv \in E(H_t)$ if and only if $uv \in E(G)$. Then the statement follows from $H'[V(H_t)] = H_t$. On the other hand, if $u \in V_i^1$ and $v \in V_j^2$, then by construction we have placed an edge $uv \in E(H')$ if and only if $\eta_1(u)\eta_2(v) \in E(G)$. Then observe that $\eta_1(u)$ is defined so that $N_G(u) \cap \mathcal{L}(\mathcal{T})[x\vec{y}_1] = N_G(\eta_1(u)) \cap \mathcal{L}(\mathcal{T})[x\vec{y}_1]$, and $\eta_2(v)$ is defined similarly: $N_G(v) \cap \mathcal{L}(\mathcal{T})[x\vec{y}_2] = N_G(\eta_2(v)) \cap \mathcal{L}(\mathcal{T})[x\vec{y}_2]$. Since $u, \eta_1(u) \in \mathcal{L}(\mathcal{T})[x\vec{y}_2]$ and $v, \eta_2(v) \in \mathcal{L}(\mathcal{T})[x\vec{y}_1]$, we get that $uv \in E(G)$ if and only if $\eta_1(u)\eta_2(v) \in E(G)$. The statement follows.

Then pick $i \in [c]$. We ought to show that V_i is a minimal representative of X_i in G. Let $t \in [2]$ and $v \in X_i^t$. Since $\mathbb{H}_t$ encodes an indexed partition $(X_1^t, \ldots, X_c^t)$ of S_t, there is $u \in V_i^t$ with $N_G(v) \setminus X_i^t = N_G(u) \setminus X_i^t$, and $u \in V_i'$ by construction. Also by construction, there exists $u' \in V_i$ such that $\eta'(u) = \eta'(u')$ and $N_{H'}(u) = N_{H'}(u')$. We have:

- $N_G(u) \cap \mathcal{L}(\mathcal{T})[\vec{px}] = N_G(u') \cap \mathcal{L}(\mathcal{T})[\vec{px}]$ (since $\eta'(u) = \eta'(u')$),
- $N_G(u) \cap X_j = N_G(u') \cap X_j$ for all $j \neq i$: Let $w \in X_j$. Pick $t' \in [2]$ for which $w \in X_j^{t'}$. Since $V_j^{t'}$ is a minimal representative of $X_j^{t'}$ in G, there is some $w' \in V_j^{t'}$ such that $N_G(w) \setminus X_j^{t'} = N_G(w') \setminus X_j^{t'}$. Since $uw' \in E(H') \Leftrightarrow u'w' \in E(H')$, we have $uw' \in E(G) \Leftrightarrow u'w' \in E(G)$ by the considerations above. As $u, u' \notin X_j^{t'}$, we conclude that

$$uw \in E(G) \Leftrightarrow uw' \in E(G) \Leftrightarrow u'w' \in E(G) \Leftrightarrow u'w \in E(G).$$

So $N_G(u') \setminus X_i = N_G(u) \setminus X_i = N_G(v) \setminus X_i$, where the last equality follows from $N_G(u) \setminus X_i^t = N_G(v) \setminus X_i^t$. It follows that $u' \in V_i$ represents v in X_i. As v was arbitrary, we conclude that V_i is a representative of X_i.

For minimality, observe that if $u, v \in V_i'$ with $N_G(u) \setminus X_i = N_G(v) \setminus X_i$, then also $\eta'(u) = \eta'(v)$ (since $N_G(u) \cap \mathcal{L}(\mathcal{T})[\vec{xp}] = N_G(v) \cap \mathcal{L}(\mathcal{T})[\vec{xp}]$) and $N_{H'}(u) = N_{H'}(v)$ (since $N_G(u) \cap X_j = N_G(v) \cap X_j$ for $j \neq i$). Thus the construction of $\mathbb{H}$ from $\mathbb{H}'$ would remove either u or v from the graph. ◁

Next, Combine preserves isomorphism in the following sense:

Claim 4.9.15. *For each $t \in [2]$, suppose that $\mathbb{H}_t$ and $\mathbb{H}_t^\star$ are $(c, \mathcal{R}(y_t\vec{x}))$-indexed graphs respecting $\mathcal{T}$ along $y_t\vec{x}$ such that $\mathbb{H}_t \sim^{c,\mathcal{R}(y_t\vec{x})} \mathbb{H}_t^\star$. Then $\mathsf{Combine}(\mathbb{H}_1, \mathbb{H}_2) \sim^{c,\mathcal{R}(\vec{xp})} \mathsf{Combine}(\mathbb{H}_1^\star, \mathbb{H}_2^\star)$.*

Proof of the claim. Let $\mathbb{H} := \mathsf{Combine}(\mathbb{H}_1, \mathbb{H}_2)$ and $\mathbb{H}'$ be the auxiliary graph in the construction of $\mathbb{H}$. Likewise, let $\mathbb{H}^\star := \mathsf{Combine}(\mathbb{H}_1^\star, \mathbb{H}_2^\star)$ and $(\mathbb{H}^\star)'$ be the auxiliary graph in the construction of $\mathbb{H}^\star$. Also let $\pi_1 : V(\mathbb{H}_1) \to V(\mathbb{H}_1^\star)$, $\pi_2 : V(\mathbb{H}_2) \to V(\mathbb{H}_2^\star)$ be the isomorphisms promised by the statement of the claim.

Observe that $\pi' : V(\mathbb{H}') \to V((\mathbb{H}^\star)')$ given by $\pi'|_{V(\mathbb{H}_1)} = \pi_1$ and $\pi'|_{V(\mathbb{H}_2)} = \pi_2$ is an isomorphism between $\mathbb{H}'$ and $(\mathbb{H}^\star)'$: This holds since for each $i \in [c]$, we have $\pi(V_i(\mathbb{H}')) = \pi(V_i(\mathbb{H}_1) \cup V_i(\mathbb{H}_2)) = \pi_1(V_i(\mathbb{H}_1)) \cup \pi_2(V_i(\mathbb{H}_2)) = V_i(\mathbb{H}_1^\star) \cup V_i(\mathbb{H}_2^\star) = V_i((\mathbb{H}^\star)')$ and, for each $v \in V(\mathbb{H}_t)$ with $t \in [2]$, $\eta(\mathbb{H}')(v) = \mathcal{F}(y_t xp)\,(\eta(\mathbb{H}_t)(v)) = \mathcal{F}(y_t xp)\,(\eta(\mathbb{H}_t^\star)(\pi_t(v))) = \eta((\mathbb{H}^\star)')(\pi_t(v))$. Also the same arguments as in Claim 4.9.14 show that π' gives an isomorphism of the graphs $G(\mathbb{H}')$ and $G((\mathbb{H}^\star)')$.

Since $\mathbb{H}'$ and $(\mathbb{H}^\star)'$ are isomorphic, it can be easily verified that the process of the construction of $\mathbb{H}$ from $\mathbb{H}'$ and $\mathbb{H}^\star$ from $(\mathbb{H}^\star)'$ preserves isomorphism. This finishes the proof. ◁

The following claim follows from a simple application of Claim 4.9.14.

Claim 4.9.16. *Suppose $\mathbb{H}$ encodes an indexed partition $(X_1, \ldots, X_c)$ of S. Then there exists a $(c, \mathcal{R}(y_1\vec{x}))$-indexed graph $\mathbb{H}_1$ and a $(c, \mathcal{R}(y_2\vec{x}))$-indexed graph $\mathbb{H}_2$ such that:*

- *for each $t \in [2]$, $\mathbb{H}_t$ encodes the indexed partition $(X_1 \cap S_t, \ldots, X_c \cap S_t)$ of S_t; and*
- $\mathsf{Combine}(\mathbb{H}_1, \mathbb{H}_2) \sim^{c,\mathcal{R}(\vec{xp})} \mathbb{H}$.

Proof of the claim. For $t \in [2]$, let $\mathbb{H}_t$ be any $(c, \mathcal{R}(y_t\vec{x}))$-indexed graph encoding the indexed partition $(X_1 \cap S_t, \ldots, X_c \cap S_t)$. Such an indexed graph must exist since it is enough to take $\mathbb{H}_t = (V_1^t, \ldots, V_c^t), H_t, \eta_t)$, where for $i \in [c]$, V_i^t is any minimal representative of $X_i \cap S_t$ in G, and the objects H_t, η_t are uniquely deduced from $V_1^t, \ldots, V_c^t$.

By Claim 4.9.14, $\mathsf{Combine}(\mathbb{H}_1, \mathbb{H}_2)$ encodes $(X_1, \ldots, X_c)$. Since indexed graphs encoding the same indexed partition of S are isomorphic, we conclude that $\mathsf{Combine}(\mathbb{H}_1, \mathbb{H}_2) \sim^{c,\mathcal{R}(\vec{xp})} \mathbb{H}$. ◁

We also notice the following claim binding the cost of an indexed partition of S to the costs of indexed partitions of S_1, S_2:

Claim 4.9.17. *Let $\mathbb{H}$ be a $(c, \mathcal{R}(\vec{xp}))$-indexed graph encoding an indexed partition $(X_1, \ldots, X_c)$ of S of cost $q \in \mathbb{N}$, and for each $t \in [2]$, $(X_1 \cap S_t, \ldots, X_c \cap S_t)$ be an indexed partition of S_t of cost $q_i \in \mathbb{N}$. Let $\delta \in \{0, 1\}$ be the indicator equal to 1 if and only if $\mathbb{H}$ has at least two nonempty parts (equivalently, at least two sets $X_1, \ldots, X_c$ are nonempty). Then $q = q_1 + q_2 + \delta$.*

Proof of the claim. Recall that q is the number of oriented edges $\vec{e}$ that are predecessors of $\vec{xp}$ such that $\mathcal{L}(\mathcal{T})[\vec{e}]$ intersects more than one set in $(X_1, \ldots, X_c)$. Noting that the two edges $y_1\vec{x}$ and $y_2\vec{x}$ are the two children of $\vec{xp}$, we see that q is the sum of the following values:

- for each $t \in [2]$, the number of predecessors $\vec{e}$ of $y_t\vec{x}$ such that $\mathcal{L}(\mathcal{T})[\vec{e}]$ intersects at least two sets in $(X_1, \ldots, X_c)$. Since $\mathcal{L}(\mathcal{T})[\vec{e}] \subseteq S_t$, this is equivalently the number of predecessors $\vec{e}$ of $y_t\vec{x}$ with $\mathcal{L}(\mathcal{T})[\vec{e}]$ intersecting at least two sets in $(X_1 \cap S_t, \ldots, X_c \cap S_t)$, or exactly q_t; and

- 1 if $\mathcal{L}(\mathcal{T})[\vec{px}] = S$ intersects at least two sets in $(X_1, \ldots, X_c)$, or 0 otherwise; equivalently, this indicator is equal to 1 if and only if at least two sets in $(X_1, \ldots, X_c)$ are nonempty. Since for each $i \in [c]$, the set X_i is nonempty if and only if $V_i(\mathbb{H})$ is nonempty, this indicator is equal to exactly δ.

Therefore, $q = q_1 + q_2 + \delta$. ◁

Finally, the following claim will enable us to compute $\mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp})$.

Claim 4.9.18. *Let $\mathcal{K}$ be an equivalence class of $\sim_s^{c,\mathcal{R}(\vec{xp})}$ and suppose $(q, \mathbb{H}) \in A_{\mathcal{K}}$ has the minimum possible cost q among all pairs in $A_{\mathcal{K}}$. Let $\delta \in \{0, 1\}$ be an indicator equal to 0 if $\mathbb{H}$ has at most one nonempty part, and 1 otherwise. Then there exist pairs $(q_1, \mathbb{H}_1) \in \mathsf{reps}^{c,s}(\mathcal{T}, y_1\vec{x})$ and $(q_2, \mathbb{H}_2) \in \mathsf{reps}^{c,s}(\mathcal{T}, y_2\vec{x})$ such that*

$$q = q_1 + q_2 + \delta,$$
$$\mathbb{H} \sim^{c,\mathcal{R}(\vec{xp})} \mathsf{Combine}(\mathbb{H}_1, \mathbb{H}_2).$$

Proof of the claim. Let $(q, \mathbb{H})$ and $\delta \in \{0, 1\}$ be as in the statement of the claim. By definition, $\mathbb{H}$ is an s-small $(c, \mathcal{R}(\vec{xp}))$-indexed graph and there exists an indexed partition $(X_1, \ldots, X_c)$ of S of cost q encoded by $\mathbb{H}$. For $t \in [2]$ and $j \in [c]$, define $X_j^t = X_j \cap S_t$, so that $(X_1^t, \ldots, X_c^t)$ is an indexed partition of S_t; let then $q_t \in \mathbb{N}$ be the cost of this partition. Then $q = q_1 + q_2 + \delta$ by Claim 4.9.17.

Let $\mathbb{H}_1, \mathbb{H}_2$ be $(c, \mathcal{R}(y_1\vec{x}))$-indexed and $(c, \mathcal{R}(y_2\vec{x}))$-indexed, respectively, graphs with the properties that $\mathsf{Combine}(\mathbb{H}_1, \mathbb{H}_2) \sim^{c,\mathcal{R}(\vec{xp})} \mathbb{H}$ and for each $t \in [2]$, $\mathbb{H}_t$ encodes $(X_1^t, \ldots, X_c^t)$. Note that such indexed graphs exist by Claim 4.9.16. For each $t \in [2]$, we claim that $\mathbb{H}_t$ is s-small. Suppose otherwise; let $\mathbb{H}_t = ((V_1^t, \ldots, V_c^t), H_t, \eta_t)$ so that V_j^t is a minimal representative of X_j^t for all $j \in [c]$. Then there is some index $j \in [c]$ and $s+1$ vertices $v_1, \ldots, v_{s+1} \in V_j^t$ such that:

- $\eta_t(v_1) = \ldots = \eta_t(v_{s+1})$,
- the neighborhoods $N_G(v_1) \cap \overline{X_j^t}, \ldots, N_G(v_{s+1}) \cap \overline{X_j^t}$ are pairwise different.

From $\eta_t(v_1) = \ldots = \eta_t(v_{s+1})$ and $\mathbb{H}_t$ respecting $\mathcal{T}$ along $y_t\vec{x}$, we also have $N_G(v_1) \cap \overline{S_t} = \ldots = N_G(v_{s+1}) \cap \overline{S_t}$. Since $X_j^t \subseteq S_t$, we infer that all the neighborhoods $N_G(v_1) \cap (S_t \setminus X_j^t), \ldots, N_G(v_{s+1}) \cap (S_t \setminus X_j^t)$ are pairwise different. So we have that:

- $v_1, \ldots, v_{s+1} \in X_j$,
- $N_G(v_1) \cap \overline{X_j}, \ldots, N_G(v_{s+1}) \cap \overline{X_j}$ are pairwise different (since $S_t \setminus X_j^t \subseteq \overline{X_j}$), and
- $\eta(v_1) = \ldots = \eta(v_{s+1})$ (since $N_G(v_1) \cap \overline{S} = \ldots = N_G(v_{s+1}) \cap \overline{S}$).

Therefore, any minimal representative of X_j in G must contain at least $s+1$ vertices with the same neighborhood $N_G(v_1) \cap \overline{S}$ in $\overline{S}$. This implies that $V_j(\mathbb{H})$ must contain at least $s+1$ vertices labeled $\eta(v_1)$ – a contradiction since we assumed $\mathbb{H}$ is s-small. So $\mathbb{H}_t$ is indeed s-small.

For each $t \in [2]$, $\mathbb{H}_t$ encodes the indexed partition $(X_1^t, \ldots, X_c^t)$ of cost q_t. Thus there is a pair $(q_t^\star, \mathbb{H}_t^\star) \in \mathsf{reps}^{c,s}(\mathcal{T}, y_1\vec{x})$ and $(q_2^\star, \mathbb{H}_2^\star) \in \mathsf{reps}^{c,s}(\mathcal{T}, y_2\vec{x})$ such that $q_t^\star \leqslant q_t$ and $\mathbb{H}_t^\star \sim^{c,\mathcal{R}(\vec{xp})} \mathbb{H}_t$ for each $t \in [2]$. Let also, for each $t \in [2]$, $(Y_1^t, \ldots, Y_c^t)$ be an indexed partition of S_t of cost $q_t^\star$ encoded by $\mathbb{H}_t^\star$. Then take $\mathbb{H}^\star := \mathsf{Combine}(\mathbb{H}_1^\star, \mathbb{H}_2^\star)$. By Claim 4.9.15, $\mathbb{H}^\star \sim^{c,\mathcal{R}(\vec{xp})} \mathbb{H}$; in particular, $\mathbb{H}^\star$ has at most one nonempty part if and only if $\mathbb{H}$ does. By Claim 4.9.14, $\mathbb{H}^\star$ encodes the indexed partition $(Y_1, \ldots, Y_c)$, where $Y_i = Y_i^1 \cup Y_i^2$ for $i \in [c]$. This partition has cost $q_1^\star + q_2^\star + \delta$ by Claim 4.9.17, so $(q_1^\star + q_2^\star + \delta, \mathbb{H}^\star) \in A_{\mathcal{K}}$. But since $(q, \mathbb{H})$ has the minimum cost among all pairs in $A_{\mathcal{K}}$, we get

$$q \leqslant q_1^\star + q_2^\star + \delta \leqslant q_1 + q_2 + \delta = q.$$

Therefore, $q_1^\star = q_1$ and $q_2^\star = q_2$ and thus $q = q_1 + q_2 + \delta$ for $(q_1, \mathbb{H}_1^\star) \in \mathsf{reps}^{c,s}(\mathcal{T}, y_1\vec{x})$, $(q_2, \mathbb{H}_2^\star) \in \mathsf{reps}^{c,s}(\mathcal{T}, y_2\vec{x})$ and $\mathbb{H} \sim^{c,\mathcal{R}(\vec{xp})} \mathsf{Combine}(\mathbb{H}_1^\star, \mathbb{H}_2^\star)$. ◁

Therefore, we compute the set $\mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp})$ as follows. We populate a set $\mathcal{W}$ comprising pairwise different pairs containing a nonnegative integer and a $(c, \mathcal{R}(\vec{xp}))$-indexed graph by:

- iterating all pairs $(q_1, \mathbb{H}_1) \in \mathsf{reps}^{c,s}(\mathcal{T}, y_1\vec{x})$ and $(q_2, \mathbb{H}_2) \in \mathsf{reps}^{c,s}(\mathcal{T}, y_2\vec{x})$,
- computing $\mathbb{H} = \mathsf{Combine}(\mathbb{H}_1, \mathbb{H}_2)$ and $q = q_1 + q_2 + \delta$, where $\delta = 1$ if $\mathbb{H}$ contains at least two nonempty parts, and $\delta = 0$ otherwise, and

- if $\mathbb{H}$ is s-small, adding a pair $(q, \mathbb{H})$ to $\mathcal{W}$.

Then we filter $\mathcal{W}$ as follows: whenever $\mathcal{W}$ contains pairs $(q, \mathbb{H})$ and $(q', \mathbb{H}')$ such that $q \leqslant q'$ and $\mathbb{H} \sim^{c,\mathcal{R}(\vec{xy})} \mathbb{H}'$, we drop $(q', \mathbb{H}')$ from $\mathcal{W}$. Naturally, this entire process (the construction of $\mathcal{W}$ and its subsequent filtering) can be carried out in time $\mathcal{O}_{c,s,\ell}(1)$. We finally set $\mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp}) \coloneqq \mathcal{W}'$. Naturally, by Claim 4.9.18, $\mathcal{W}'$ is a set of (c,s)-small representatives of $\mathcal{T}$ along $\vec{xp}$.

We conclude the proof by observing that, for every $(q, \mathbb{H}) \in \mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp})$, we can define the mapping $\Phi((q, \mathbb{H}))$ as any pair $((q_1, \mathbb{H}_1), (q_2, \mathbb{H}_2))$ for which $q = q_1 + q_2 + \delta$ and $\mathbb{H} = \mathsf{Combine}(\mathbb{H}_1, \mathbb{H}_2)$, where $\delta = 1$ if and only if $\mathbb{H}$ contains at least two nonempty parts. (Such a pair exists by the construction of $\mathcal{W}$.) Then $\Phi((q, \mathbb{H}))$ satisfies all the requirements of the lemma by Claims 4.9.14 and 4.9.17. □

4.9.3 State optimization problem for rank decomposition automata

In this subsection, we introduce an optimization problem for rank decomposition automata that will be used in the proof of Lemma 4.4.7. We will also show that this problem can be solved efficiently under the reasonable assumptions on the automaton.

Let $(S, +, \leqslant)$ be a totally ordered commutative semigroup, i.e., a commutative semigroup $(S, +)$ with a total order $\leqslant$ with the property that, for any $x, y, z \in S$ with $x \leqslant y$, we have $x + z \leqslant y + z$. Assume that $+$ can be evaluated in time β.

Let also $\mathcal{A} = (Q, \iota, \delta, \varepsilon)$ be a label-oblivious rank decomposition automaton of width ℓ with evaluation time β and a finite set of states. Suppose $\mathcal{T} = (T, U, \mathcal{R}, \mathcal{E}, \mathcal{F})$ is an unrooted annotated rank decomposition of width at most ℓ. We will call any function $\kappa : \vec{L}(T) \to Q$ a *leaf edge state mapping*. Given a leaf edge state mapping κ and an edge $\vec{ab} \in \vec{E}(T)$, we define the *κ-run* of $(\mathcal{A}, a, b)$ as the function $\rho_\kappa : \mathsf{pred}_T(\vec{ab}) \cup \mathsf{pred}_T(\vec{ba}) \cup \{\vartheta\} \to Q$ defined as follows:

- for each leaf edge $\vec{lp} \in \vec{L}(T)$ it holds that $\rho_\kappa(\vec{lp}) = \kappa(\vec{lp})$;
- for each nonleaf edge $\vec{tp}$ of T with children $\vec{c_1t}, \vec{c_2t}$, where $c_1 < c_2$, it holds that $\rho_\kappa(\vec{tp}) = \delta(\tau(\mathcal{T}, \vec{tp}), \rho_\kappa(\vec{c_1t}), \rho_\kappa(\vec{c_2t}))$;
- $\rho_\kappa(\vartheta) = \varepsilon(\delta(\mathcal{T}, \vec{ab}), \rho_\kappa(\vec{ab}), \rho_\kappa(\vec{ba}))$.

So, in other words, a κ-run of an automaton is defined similarly to a run of an automaton, only that the initial mapping ι of the automaton is ignored, and instead we fix the state $\rho_\kappa(\vec{lp})$ of each leaf edge $\vec{lp}$ to $\kappa(\vec{lp})$.

Moreover, let $\mathbf{c} : \vec{L}(T) \times Q \to S$ be a *cost function*. Then the cost of a leaf edge state mapping κ is defined as $\mathbf{c}(\kappa) \coloneqq \sum_{e \in \vec{L}(T)} \mathbf{c}(e, \kappa(e))$.

We now show that the optimization problem where, given a set $F \subseteq Q$ of states, we are to find a leaf edge state mapping κ of minimum cost for which $\rho_\kappa(\vartheta) \in F$, can be solved efficiently. The proof is a standard application of the dynamic programming technique.

Lemma 4.9.19. *Given:*

- *a totally ordered commutative semigroup $(S, +, \leqslant)$ with evaluation time β,*
- *a label-oblivious rank decomposition automaton $\mathcal{A} = (Q, \iota, \delta, \varepsilon)$ of width ℓ with evaluation time β and a finite set Q of states,*
- *an annotated rank decomposition $\mathcal{T} = (T, U, \mathcal{R}, \mathcal{E}, \mathcal{F})$ of width at most ℓ with n nodes,*
- *an edge $\vec{ab} \in \vec{E}(T)$,*
- *a cost function $\mathbf{c} : \vec{L}(T) \times Q \to S$, and*
- *a set $F \subseteq Q$ of* accepting states,

it is possible to determine in time $\mathcal{O}(|Q|^2 n\beta)$, whether there exists a leaf edge state mapping κ such that $\rho_\kappa(\vartheta) \in F$. If such a mapping exists, then it is also possible to determine any such mapping minimizing the value of $\mathbf{c}(\kappa)$.

Proof. Note that in the definition of ρ_κ before, the value $\rho_\kappa(\vec{uv})$ for an edge $\vec{uv} \in \vec{E}(T)$ only depends on the values of κ for $\vec{lp} \in \mathsf{pred}_T(\vec{uv})$. Therefore, without confusion we will write $\rho_{\kappa'}(\vec{uv})$ whenever κ' is a partial function defined on $\mathsf{pred}_T(\vec{uv}) \cap \vec{L}(T)$.

We want to compute, for every oriented edge $\vec{uv} \in \mathsf{pred}_T(\vec{ab}) \cup \mathsf{pred}_T(\vec{ba})$, the function $\mathrm{best}_{\vec{uv}} : Q \to S \cup \{\bot\}$ with the following property for every $f \in Q$: Suppose $K_{\vec{uv},f}$ is the set of all partial valuations $\kappa' : \mathsf{pred}_T(\vec{uv}) \cap \vec{L}(T) \to Q$ such that $\rho_{\kappa'}(\vec{uv}) = f$. Then $\mathrm{best}_{\vec{uv}}(f) = \bot$ if $K_{\vec{uv},f} = \emptyset$; otherwise, $\mathrm{best}_{\vec{uv}}(f)$ is equal to the minimum value of $\sum_{e \in \mathsf{pred}_T(\vec{uv}) \cap \vec{L}(T)} \mathbf{c}(e, \kappa'(e))$ over all $\kappa' \in K_{\vec{uv},f}$. It is easy to observe that:

- for a leaf edge $\vec{lp} \in \vec{L}(T)$, we have $\mathrm{best}_{\vec{lp}}(q) = \mathbf{c}(\vec{lp}, f)$ for each $f \in Q$;
- for a nonleaf edge $\vec{tp} \in \vec{E}(T)$ with children $\vec{c_1t}$ and $\vec{c_2t}$ with $c_1 < c_2$, we have, for every $f \in Q$,

$$\begin{aligned}\mathrm{best}_{\vec{tp}}(f) = \min\{&\mathrm{best}_{\vec{c_1t}}(f_1) + \mathrm{best}_{\vec{c_2t}}(f_2) \mid \\ &f_1, f_2 \in Q,\ \mathrm{best}_{\vec{c_1t}}(f_1) \neq \bot,\ \mathrm{best}_{\vec{c_2t}}(f_2) \neq \bot,\ f = \delta(\tau(\mathcal{T}, \vec{tp}), f_1, f_2)\};\end{aligned} \tag{4.12}$$

  where we set $\mathrm{best}_{\vec{tp}}(q) = \bot$ if the set on the right-hand side of Eq. (4.12) is empty. So given $\mathrm{best}_{\vec{c_1t}}$ and $\mathrm{best}_{\vec{c_2t}}$, we can compute $\mathrm{best}_{\vec{tp}}$ in time $\mathcal{O}(|Q|^2\beta)$.

Therefore, all functions $\mathrm{best}_{\vec{uv}}$ can be computed in time $\mathcal{O}(|Q|^2 n\beta)$ by a simple bottom-up dynamic programming on trees with a depth-first search on T. Similarly we define $\mathrm{best} : Q \to S \cup \{\bot\}$ with the following property for all $f \in Q$: let K_f be the set of valuations $\kappa : \vec{L}(T) \to Q$ such that $\rho_\kappa(\vartheta) = f$. Then $\mathrm{best}(f) = \bot$ if $K_f = \emptyset$, and otherwise $\mathrm{best}(f)$ is the minimum value of $\sum_{e \in \vec{L}(T)} \mathbf{c}(e, \kappa(e))$ over all $\kappa \in K_f$. As in Eq. (4.12), we get that

$$\begin{aligned}\mathrm{best}(f) = \min\{&\mathrm{best}_{\vec{ab}}(f_1) + \mathrm{best}_{\vec{ba}}(f_2) \mid \\ &f_1, f_2 \in Q,\ \mathrm{best}_{\vec{ab}}(f_1) \neq \bot,\ \mathrm{best}_{\vec{ba}}(f_2) \neq \bot,\ f = \varepsilon(\delta(\mathcal{T}, \vec{ab}), f_1, f_2)\};\end{aligned} \tag{4.13}$$

where $\mathrm{best}(f) = \bot$ is set if the set on the right-hand side of Eq. (4.13) is empty. Then best can be computed in time $\mathcal{O}(|Q|^2\beta)$ given $\mathrm{best}_{\vec{ab}}$ and $\mathrm{best}_{\vec{ba}}$. Now, if $\mathrm{best}(f) = \bot$ for all $f \in F$, then we return that no mapping κ with $\rho_\kappa(\vartheta) = q_0$ exists. Otherwise, such a mapping exists. Let $f_0 \in F$ be the argument minimizing $\mathrm{best}(f_0)$ among all $f \in F$ with $\mathrm{best}(f) \neq \bot$. By retracing the optimum choices done by the dynamic programming scheme using the top-bottom depth-first search on T, we fully recover a run ρ_κ for some $\kappa : \vec{L}(T) \to Q$ such that $\rho_\kappa(\vartheta) = f_0$ and $\mathbf{c}(\kappa)$ is minimum possible; and we recover κ by observing that for every $\vec{lp} \in \vec{L}(T)$, it holds that $\kappa(\vec{lp}) = \rho_\kappa(\vec{lp})$. □

Note that Lemma 4.9.19 can be easily generalized to the case where Q is an infinite set, but there exists a bound $q \in \mathbb{N}_{\geqslant 1}$ on the size of the set

$$\{\rho_\kappa(x) \mid \kappa : \vec{L}(T) \to Q\}$$

for all x. Then it can be verified that the optimization problem stated above can be solved in time $\mathcal{O}(q^2 n\beta)$.

4.9.4 Prefix-rebuilding data structure for minimal closures

In this subsection, we finally give a proof of Lemma 4.4.7. Before we begin, we describe an operation of *gluing* rank decompositions; a similar notion appears in the proof of Lemma 4.3.8.

Suppose we have two disjoint sets of vertices A, B and that $R_A \subseteq A$, $R_B \subseteq B$; we also have two partitioned graphs $(G_A, \mathcal{C}_A)$, $(G_B, \mathcal{C}_B)$ with vertex sets $A \cup R_B$ and $B \cup R_A$, respectively, such that $\{R_B\} \in \mathcal{C}_A$ and $\{R_A\} \in \mathcal{C}_B$. Suppose also $\mathcal{T}_A = (T_A, U_A, \mathcal{R}_A, \mathcal{E}_A, \mathcal{F}_A)$ and $\mathcal{T}_B = (T_B, U_B, \mathcal{R}_B, \mathcal{E}_B, \mathcal{F}_B)$ are annotated rank decompositions encoding $(G_A, \mathcal{C}_A)$ and $(G_B, \mathcal{C}_B)$, with the following properties: $V(T_A) \cap V(T_B) = \{x, y\}$ and there exists a leaf edge $\vec{xy} \in \vec{L}(T_A)$ and a leaf edge $\vec{yx} \in \vec{L}(T_B)$ such that:

- $\mathcal{L}(\mathcal{T}_A)[\vec{xy}] = R_B$ and $\mathcal{L}(\mathcal{T}_B)[\vec{yx}] = R_A$,
- $\mathcal{R}_A(\vec{xy}) = \mathcal{R}_B(\vec{xy}) = R_B$ and $\mathcal{R}_B(\vec{yx}) = \mathcal{R}_A(\vec{yx}) = R_A$, and
- $\mathcal{E}_A(xy) = \mathcal{E}_B(xy)$.

We then define the *gluing of $\mathcal{T}_A$ along xy with $\mathcal{T}_B$* as the annotated rank decomposition $\mathcal{T} = (T, U, \mathcal{R}, \mathcal{E}, \mathcal{F})$ as follows:

- $V(T) = V(T_A) \cup V(T_B)$ and $E(T) = E(T_A) \cup E(T_B)$,
- $U = U_A \cup U_B = A \cup B$,
- $\mathcal{R}|_{\vec{E}(T_A)} = \mathcal{R}_A$ and $\mathcal{R}|_{\vec{E}(T_B)} = \mathcal{R}_B$,
- $\mathcal{E}|_{E(T_A)} = \mathcal{E}_A$ and $\mathcal{E}|_{E(T_B)} = \mathcal{E}_B$, and
- $\mathcal{F}|_{\mathcal{P}_3(T_A)} = \mathcal{F}_A$ and $\mathcal{F}|_{\mathcal{P}_3(T_B)} = \mathcal{F}_B$.

It can be verified that $\mathcal{T}$ is an annotated rank decomposition encoding a partitioned graph $(G, \mathcal{C})$, where $V(G) = A \cup B$, $\mathcal{C} = (\mathcal{C}_A \setminus \{R_B\}) \cup (\mathcal{C}_B \setminus \{R_A\})$, $G[A] = G_A$, $G[B] = G_B$ and $uv \in E(G)$ for $u \in A$, $v \in B$ if and only if $u'v' \in \mathcal{E}(xy)$, where $u' \in R_A$ is the (unique) vertex such that $N_{G_A}(u) \cap R_B = N_{G_A}(u') \cap R_B$, and $v' \in R_B$ is the unique vertex such that $N_{G_B}(v) \cap R_A = N_{G_B}(v') \cap R_A$. Moreover, the width of $\mathcal{T}$ is trivially the maximum of the widths of $\mathcal{T}_A$ and $\mathcal{T}_B$.

We are now ready to prove Lemma 4.4.7, which we restate below for convenience.

Lemma 4.4.7. *There is an ℓ-prefix-rebuilding data structure that takes integer parameters $c \geqslant 1$ and $k \leqslant \ell$ at initialization, has overhead $\mathcal{O}_{c,\ell}(1)$, maintains a rooted annotated rank decomposition $\mathcal{T}$, and additionally supports the following query:*

- $\mathsf{Closure}(T_{\text{pref}})$*: Given a prefix T_{pref} of $\mathcal{T}$, either in time $\mathcal{O}_\ell(|T_{\text{pref}}|)$ returns that no c-small k-closure of T_{pref} exists, or for a minimal c-small k-closure $\mathcal{C}$ of T_{pref} in time $\mathcal{O}_\ell(|\mathsf{cut}_T(\mathcal{C})|)$ returns*
  - *the sets $\mathsf{cut}_T(\mathcal{C})$ and $\mathsf{aep}_T(\mathcal{C})$, and*
  - *a rooted rank decomposition (T^*, λ^*) of $(G[\mathcal{C}], \mathcal{C})$ of width at most $2k$, where λ^* is represented as a function $\lambda\colon \mathsf{aep}_T(\mathcal{C}) \to \vec{L}(T^\star)$.*

Proof. For the course of the proof, fix $s := 2^{2k}$ and the following label-oblivious rank decomposition automata:

- the exact rankwidth automaton $\mathcal{JKO} = \mathcal{JKO}_{2k,\, cs\ell+\ell}$, given by Lemma 4.9.11; and
- the closure automaton $\mathcal{CR} = \mathcal{CR}_{c,\, s,\, \ell}$ of width ℓ, given by Lemma 4.9.13.

Note that both $\mathcal{JKO}$ and $\mathcal{CR}$ have evaluation time $\mathcal{O}_{c,\ell}(1)$. Our data structure consists simply of an instance of $\mathcal{CR}$, maintained dynamically by the data structure of Lemma 4.5.1. Thus the initialization time of the data structure on a rooted annotated rank decomposition $\mathcal{T}$ is $\mathcal{O}_{c,\ell}(|\mathcal{T}|)$, each prefix-rebuilding update $\overline{u}$ is applied to the decomposition and the automaton in time $\mathcal{O}_{c,\ell}(|\overline{u}|)$, and each operation Run and $\mathsf{Valuation}$ runs in time $\mathcal{O}(1)$.

It remains to implement $\mathsf{Closure}(T_{\text{pref}})$. So suppose we are given as a query a leafless prefix T_{pref} of $\mathcal{T}$. Let $A = \vec{\mathsf{App}}_T(T_{\text{pref}})$ be the set of appendix edges of T_{pref} and let $\mathcal{C}_{\text{pref}} := \{\mathcal{R}(\vec{xp}) \mid \vec{xp} \in A\}$. We first perform a clean-up of the prefix T_{pref} of $\mathcal{T}$ by replacing all representatives on the annotations in T_{pref} with elements of $\bigcup \mathcal{C}_{\text{pref}}$:

Claim 4.9.20. *In time $\mathcal{O}_\ell(|T_{\text{pref}}|)$, one can produce a rooted annotated rank decomposition $\mathcal{T}_{\text{skel}} = (T_{\text{skel}}, U_{\text{skel}}, \mathcal{R}_{\text{skel}}, \mathcal{E}_{\text{skel}}, \mathcal{F}_{\text{skel}})$ encoding the partitioned graph $(G[\mathcal{C}_{\text{pref}}], \mathcal{C}_{\text{pref}})$ such that: (i) $T_{\text{skel}} = T[T_{\text{pref}} \cup \mathsf{App}_T(T_{\text{pref}})]$, (ii) for every $\vec{xp} \in A$, we have $\mathcal{L}(\mathcal{T}_{\text{skel}})[\vec{xp}] = \mathcal{R}(\vec{xp})$ and $\mathcal{R}_{\text{skel}}(\vec{xp}) = \mathcal{R}(\vec{xp})$.*

Proof of the claim. Follows immediately from Lemma 4.3.10 and its proof. ◁

Note that for each $\vec{xp} \in A$, $\mathcal{R}_{\text{skel}}(\vec{px})$ is a (minimal) representative of $\mathcal{L}(\mathcal{T})[\vec{px}]$ in G.

Auxiliary objects and definitions. For $(q, \mathbb{H}) \in \mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp})$, let the *cut-rank cost* of $\mathbb{H}$ with respect to $\vec{xp}$, denoted $\mathsf{ccost}(\mathbb{H}, \vec{xp})$, be the value computed as follows. Let $(H, \mathcal{D})$ be the partitioned graph derived from $\mathbb{H}$. Let also $\overline{\mathcal{D}} = \mathcal{D} \cup \{\mathcal{R}_{\mathrm{skel}}(p\vec{x})\}$ and $\overline{H} = G[\overline{\mathcal{D}}]$. Then $\mathsf{ccost}(\mathbb{H}, \vec{xp}) = \sum_{C \in \mathcal{D}} \mathsf{cutrk}_{\overline{H}}(C)$.

Consider Λ – the set of all mappings λ assigning to each edge $\vec{xp} \in A$ a member of $\mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp})$. For every $\vec{xp} \in A$, define $(q_\lambda(\vec{xp}), \mathbb{H}_\lambda(\vec{xp})) \coloneqq \lambda(\vec{xp})$, i.e., $q_\lambda(\vec{xp})$ and $\mathbb{H}_\lambda(\vec{xp})$ are the first and the second coordinate of $\lambda(\vec{xp})$. Let also $(H_\lambda(\vec{xp}), \mathcal{D}_\lambda(\vec{xp}))$ denote the partitioned graph derived from $\mathbb{H}_\lambda(\vec{xp})$. Also, let $\overline{\mathcal{D}}_\lambda(\vec{xp}) = \mathcal{D}_\lambda(\vec{xp}) \cup \{\mathcal{R}_{\mathrm{skel}}(p\vec{x})\}$ and $\overline{H}_\lambda(\vec{xp}) = G[\overline{\mathcal{D}}_\lambda(\vec{xp})]$. Next, set $r_\lambda(\vec{xp}) = \sum_{C \in \mathcal{D}_\lambda(\vec{xp})} \mathsf{cutrk}_{\overline{H}_\lambda(\vec{xp})}(C) = \mathsf{ccost}(\mathbb{H}_\lambda(\vec{xp}), \vec{xp})$. Then, for any $\lambda \in \Lambda$, define:

- $\mathcal{D}_\lambda \coloneqq \bigcup_{\vec{xp} \in A} \mathcal{D}_\lambda(\vec{xp})$; equivalently, $\mathcal{D}_\lambda$ is the union of all nonempty parts in all indexed graphs $\mathbb{H}_\lambda(\vec{xp})$ for $\vec{xp} \in A$;
- $G_\lambda \coloneqq G[\mathcal{D}_\lambda]$;
- $q_\lambda \coloneqq \sum_{\vec{xp} \in A} q_\lambda(\vec{xp})$;
- $r_\lambda \coloneqq \sum_{\vec{xp} \in A} r_\lambda(\vec{xp})$.

Reduction from finding minimal closures to the optimization of λ. For any k-closure $\mathcal{C}$ of T_{pref}, we shall say that it is *represented* by a family $\mathcal{D}$ of nonempty disjoint sets of $V(G)$ if $|\mathcal{D}| = |\mathcal{C}|$ and for every set $C \in \mathcal{C}$, $\mathcal{D}$ contains a representative D of C in G. We will now prove the following claim, implying that a representation of a minimal k-closure can be found by examining only families $\mathcal{C}_\lambda$:

Claim 4.9.21. *Let $\lambda \in \Lambda$ be such that the rankwidth of $(G_\lambda, \mathcal{D}_\lambda)$ is at most $2k$ and, among all such mappings λ, the value r_λ is minimum; and among those, q_λ is minimum. Then for every partition $\mathcal{C}$ of $V(G)$ defined as $\mathcal{C} = \bigcup_{\vec{xp} \in A} \mathcal{C}_{\vec{xp}}$, where $\mathcal{C}_{\vec{xp}}$ is a partition of $\mathcal{L}(\mathcal{T})[\vec{xp}]$ into at most c sets encoded by $\mathbb{H}_\lambda(\vec{xp})$ and of cost $q_\lambda(\vec{xp})$, $\mathcal{C}$ is a minimal c-small k-closure of T_{pref} represented by $\mathcal{D}_\lambda$. In particular, $\mathcal{D}_\lambda$ represents some minimal c-small k-closure of T_{pref}. Moreover, if no λ with the property above exists, then no c-small k-closure of T_{pref} exists.*

Proof of the claim. Fix $\lambda \in \Lambda$ with the property that the rankwidth of $(G_\lambda, \mathcal{D}_\lambda)$ is at most $2k$. For every $\vec{xp} \in A$, let $\mathcal{C}_{\vec{xp}}$ be a partition of $\mathcal{L}(\mathcal{T})[\vec{xp}]$ into at most c sets encoded by $\mathbb{H}_\lambda(\vec{xp})$ of cost $q_\lambda(\vec{xp})$. Then let $\mathcal{C}$ be the partition of $V(G) = \bigcup_{\vec{xp} \in A} \mathcal{L}(\mathcal{T})[\vec{xp}]$ defined as $\mathcal{C} = \bigcup_{\vec{xp} \in A} \mathcal{C}_{\vec{xp}}$. We claim that $\mathcal{C}$ is a c-small k-closure of T_{pref} such that $\sum_{C \in \mathcal{C}} \mathsf{cutrk}_G(C) = r_\lambda$ and the number of nodes of T cut by $\mathcal{C}$ is exactly $q_\lambda + |T_{\mathrm{pref}}|$.

- *$\mathcal{C}$ is a k-closure of T_{pref}:* Let $\vec{xp} \in A$. Since $\mathbb{H}_\lambda(\vec{xp})$ encodes $\mathcal{C}_{\vec{xp}}$, there exists a bijection $\chi_{\vec{xp}} : \mathcal{C}_{\vec{xp}} \to \mathcal{D}_\lambda(\vec{xp})$ such that for every $C \in \mathcal{C}_{\vec{xp}}$, $\chi_{\vec{xp}}(C)$ is a minimal representative of C in G. Thus there exists a bijection $\chi : \mathcal{C} \to \mathcal{D}_\lambda$ such that for every $C \in \mathcal{C}$, $\chi(C)$ is a minimal representative of C in G. Hence, the rankwidth of $(G[\mathcal{C}], \mathcal{C})$ is equal to the rankwidth of $(G_\lambda, \mathcal{D}_\lambda) = (G[\mathcal{D}_\lambda], \mathcal{D}_\lambda)$, which is bounded from above by $2k$. Moreover, by construction, for every $C \in \mathcal{C}$ we have $C \subseteq \mathcal{L}(\mathcal{T})[\vec{xp}]$ for some $\vec{xp} \in A$. Therefore, $\mathcal{C}$ is a k-closure of T_{pref}.

- *$\mathcal{C}$ is c-small*: For every $\vec{xp} \in A$, $\mathcal{C}_{\vec{xp}}$ is the subfamily of $\mathcal{C}$ forming a partition of $\mathcal{L}(\mathcal{T})[\vec{xp}]$. By construction, $|\mathcal{C}_{\vec{xp}}| \leqslant c$.

- $\sum_{C \in \mathcal{C}} \mathsf{cutrk}_G(C) = r_\lambda$: Choose $C \in \mathcal{C}$ and let $\vec{xp} \in A$ be such that $C \in \mathcal{C}_{\vec{xp}}$. As noted before, the bijection $\chi_{\vec{xp}} : \mathcal{C}_{\vec{xp}} \to \mathcal{D}_\lambda(\vec{xp})$ is such that for every $C \in \mathcal{C}_{\vec{xp}}$, $\chi_{\vec{xp}}(C)$ is a minimal representative of C in G. Let $R_C \coloneqq \chi_{\vec{xp}}(C)$. Also, $\mathcal{R}_{\mathrm{skel}}(p\vec{x})$ is a minimal representative of $\mathcal{L}(\mathcal{T})[p\vec{x}]$ in G. Since $\bigcup \mathcal{C}_{\vec{xp}} \cup \mathcal{L}(\mathcal{T})[p\vec{x}] = V(G)$, we find that

$$\mathsf{cutrk}_G(C) = \mathsf{cutrk}_{G[\bigcup \mathcal{D}_\lambda(\vec{xp}) \cup \{\mathcal{R}_{\mathrm{skel}}(p\vec{x})\}]}(R_C) = \mathsf{cutrk}_{\overline{H}_\lambda(\vec{xp})}(R_C).$$

  The statement now follows by summing the equation above for all $C \in \mathcal{C}$.

- *$\mathcal{C}$ cuts exactly $q_\lambda + |T_{\mathrm{pref}}|$ nodes of T*: Each node of T_{pref} must obviously be cut by every closure of T_{pref}. Then, for every $\vec{xp} \in A$, the value $q_\lambda(\vec{xp})$ denotes the cost of the partition $\mathcal{C}_{\vec{xp}}$ of $\mathcal{L}(\mathcal{T})[\vec{xp}]$, i.e., the number of nodes cut by $\mathcal{C}_{\vec{xp}}$ (equivalently, $\mathcal{C}$) in the subtree of T rooted at x. Therefore, $\mathcal{C}$ cuts $|T_{\mathrm{pref}}| + \sum_{\vec{xp} \in A} q_\lambda(\vec{xp}) = q_\lambda + |T_{\mathrm{pref}}|$ nodes of T.

Conversely, let $\mathcal{C}$ be a c-small k-closure of T_{pref} and suppose that $\sum_{C\in\mathcal{C}} \mathsf{cutrk}_G(C) = r$ and that $\mathcal{C}$ cuts q nodes of T. Our goal is to find a mapping $\lambda \in \Lambda$ such that the rankwidth of $(G_\lambda, \mathcal{D}_\lambda)$ is at most $2k$, and $r_\lambda = r$ and $q_\lambda \leqslant q - |T_{\mathrm{pref}}|$. It is easy to see that the verification of this claim will finish the proof.

For every $\vec{xp} \in A$, let $\mathcal{C}_{\vec{xp}} \subseteq \mathcal{C}$ comprise the parts of $\mathcal{C}$ that are subsets of $\mathcal{L}(\mathcal{T})[\vec{xp}]$. Since $\mathcal{C}$ is a c-small closure of T_{pref}, we have $\mathcal{C} = \bigcup_{\vec{xp}\in A} \mathcal{C}_{\vec{xp}}$ and $|\mathcal{C}_{\vec{xp}}| \leqslant c$ for all $\vec{xp} \in A$. Let $q'_{\vec{xp}}$ be the cost of $\mathcal{C}_{\vec{xp}}$, i.e., the number of the nodes in the subtree rooted at $\vec{xp}$ that are cut by $\mathcal{C}_{\vec{xp}}$. As discussed earlier in the course of the proof, we have $q = |T_{\mathrm{pref}}| + \sum_{\vec{xp}\in A} q'_{\vec{xp}}$.

For every $C \in \mathcal{C}_{\vec{xp}}$, we have $\mathsf{cutrk}_G(C) \leqslant 2k$ as $\mathcal{C}$ is a k-closure of T_{pref}. So for every $C \in \mathcal{C}_{\vec{xp}}$, we can find a minimal representative R_C of C in G of cardinality at most $2^{2k} = s$. Thus, we can define an s-small $(c, \mathcal{R}(\vec{xp}))$-indexed graph $\mathbb{H}'_{\vec{xp}} = ((V_1^{\vec{xp}}, \ldots, V_c^{\vec{xp}}), H_{\vec{xp}}, \eta_{\vec{xp}})$ encoding $\mathcal{C}_{\vec{xp}}$ by setting $\{V_1^{\vec{xp}}, \ldots, V_{|\mathcal{C}_{\vec{xp}}|}^{\vec{xp}}\} = \{R_C \mid C \in \mathcal{C}_{\vec{xp}}\}$, $V_{|\mathcal{C}_{\vec{xp}}|+1}^{\vec{xp}} = \cdots = V_c^{\vec{xp}} = \emptyset$, and choosing $H_{\vec{xp}}$ and $\eta_{\vec{xp}}$ so as to ensure that $\mathbb{H}'_{\vec{xp}}$ respects $\mathcal{T}$ along $\vec{xp}$ (as discussed before, such a choice is unique as soon as the sets $V_1^{\vec{xp}}, \ldots, V_c^{\vec{xp}}$ are determined). Now by definition of $\mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp})$, there exists a pair $(q_{\vec{xp}}, \mathbb{H}_{\vec{xp}}) \in \mathsf{reps}^{c,s}(\mathcal{T}, \mathcal{R}(\vec{xp}))$ such that $\mathbb{H}_{\vec{xp}} \sim^{c,\mathcal{R}(\vec{xp})} \mathbb{H}'_{\vec{xp}}$ and $q_{\vec{xp}} \leqslant q'_{\vec{xp}}$. Define then the mapping $\lambda \in \Lambda$ by setting $\lambda(\vec{xp}) = (q_{\vec{xp}}, \mathbb{H}_{\vec{xp}})$ for each $\vec{xp} \in A$. We claim that λ satisfies the required conditions.

In the following arguments, let $\pi_{\vec{xp}} : V(\mathbb{H}_{\vec{xp}}) \to V(\mathbb{H}'_{\vec{xp}})$ be any isomorphism from $\mathbb{H}_{\vec{xp}}$ to $\mathbb{H}'_{\vec{xp}}$. Let also $\pi : \bigcup_{\vec{xp}\in A} V(\mathbb{H}_{\vec{xp}}) \to \bigcup_{\vec{xp}\in A} V(\mathbb{H}'_{\vec{xp}})$ be defined by $\pi|_{V(\mathbb{H}_{\vec{xp}})} = \pi_{\vec{xp}}$ for each $\vec{xp} \in A$. By the properties of the isomorphism of indexed graphs, for every $\vec{xp} \in A$ and $v \in V(\mathbb{H}_{\vec{xp}})$, it holds that $N_G(v) \cap \mathcal{L}(\mathcal{T})[p\vec{x}] = N_G(\pi(v)) \cap \mathcal{L}(\mathcal{T})[p\vec{x}]$.

Define $\mathcal{D}'_\lambda \coloneqq \pi(\mathcal{D}_\lambda) = \{\pi(C) \mid C \in \mathcal{D}_\lambda\} = \{R_C \mid C \in \mathcal{C}\}$. We claim that $G[\mathcal{D}'_\lambda]$ is isomorphic to G_λ, with the isomorphism given by π. So let $u, v \in V(G_\lambda)$, aiming to show that $uv \in E(G_\lambda)$ if and only if $\pi(u)\pi(v) \in E(G[\mathcal{D}'_\lambda])$.

- Naturally, if u and v belong to the same part of $\mathcal{D}_\lambda$, then $\pi(u)$ and $\pi(v)$ belong to the same part of $\mathcal{D}'_\lambda$ and so $uv \notin E(G_\lambda)$ and $\pi(u)\pi(v) \notin E(G[\mathcal{D}'_\lambda])$.
- Otherwise, if $u, v \in V(\mathbb{H}_{\vec{xp}})$ for some $\vec{xp} \in A$ (but u, v belong to different parts), then $uv \in E(\mathbb{H}_{\vec{xp}})$ if and only if $\pi(u)\pi(v) \in E(\mathbb{H}'_{\vec{xp}})$, since π is an isomorphism from $\mathbb{H}_{\vec{xp}}$ to $\mathbb{H}'_{\vec{xp}}$. As both $\mathbb{H}_{\vec{xp}}$ and $\mathbb{H}'_{\vec{xp}}$ respect $\mathcal{T}$ along $\vec{xp}$, we have that $uv \in E(\mathbb{H}_{\vec{xp}})$ if and only if $uv \in E(G)$; and that $\pi(u)\pi(v) \in E(\mathbb{H}'_{\vec{xp}})$ if and only if $\pi(u)\pi(v) \in E(G)$. This settles this case.
- Finally, suppose $u \in V(\mathbb{H}_{x_1\vec{p}_1})$ and $v \in V(\mathbb{H}_{x_2\vec{p}_2})$ for $x_1 \neq x_2$. Then $u, \pi(u) \in \mathcal{L}(\mathcal{T})[p_2\vec{x}_2]$ and $v, \pi(v) \in \mathcal{L}(\mathcal{T})[p_1\vec{x}_1]$. From $N_G(u) \cap \mathcal{L}(\mathcal{T})[p_1\vec{x}_1] = N_G(\pi(u)) \cap \mathcal{L}(\mathcal{T})[p_1\vec{x}_1]$ we find that $uv \in E(G)$ if and only if $\pi(u)v \in E(G)$. And from $N_G(v) \cap \mathcal{L}(\mathcal{T})[p_2\vec{x}_2] = N_G(\pi(v)) \cap \mathcal{L}(\mathcal{T})[p_2\vec{x}_2]$ we get that $\pi(u)v \in E(G)$ if and only if $\pi(u)\pi(v) \in E(G)$ and we are done.

So $G[\mathcal{D}'_\lambda]$ is isomorphic to G_λ. We now verify the conditions required from λ.

- $(G_\lambda, \mathcal{D}_\lambda)$ *has rankwidth at most* $2k$*:* For each $\vec{xp} \in A$, by the construction of $\mathbb{H}'_{\vec{xp}}$, each part of $\mathbb{H}'_{\vec{xp}}$ is a subset (in fact, a minimal representative) of a unique set in $\mathcal{C}_{\vec{xp}}$. Hence $\mathcal{D}'_\lambda$ is formed from $\mathcal{C}$ by replacing each part $C \in \mathcal{C}$ with some minimal representative of C in G. Thus obviously, since $(G[\mathcal{C}], \mathcal{C})$ has rankwidth at most $2k$, then so does $(G[\mathcal{D}'_\lambda], \mathcal{D}'_\lambda)$. As $G[\mathcal{D}'_\lambda] = \pi(G[\mathcal{D}_\lambda])$ and $\mathcal{D}'_\lambda = \pi(\mathcal{D}_\lambda)$, also $(G[\mathcal{D}_\lambda], \mathcal{D}_\lambda)$ has rankwidth at most $2k$.
- $r_\lambda = r$*:* Let $\vec{xp} \in A$. Let $\overline{H}'_\lambda(\vec{xp}) = G[\overline{\mathcal{D}}'_\lambda(\vec{xp})]$, where $\overline{\mathcal{D}}'_\lambda(\vec{xp}) = \{R_C \mid C \in \mathcal{C}_{\vec{xp}}\} \cup \{\mathcal{R}_{\mathrm{skel}}(p\vec{x})\}$. Since R_C is a representative of C in G for each $C \in \mathcal{C}_{\vec{xp}}$ and $\mathcal{R}_{\mathrm{skel}}(p\vec{x})$ is a representative of $\mathcal{L}(\mathcal{T})[p\vec{x}]$ in G and $\bigcup \mathcal{C}_{\vec{xp}} \cup \mathcal{L}(\mathcal{T})[p\vec{x}] = V(G)$, we have, for every $C \in \mathcal{C}_{\vec{xp}}$,

$$\mathsf{cutrk}_G(C) = \mathsf{cutrk}_{\overline{H}'_\lambda(\vec{xp})}(R_C).$$

But now observe that the partitioned graphs $(\overline{H}_\lambda(\vec{xp}), \overline{\mathcal{D}}_\lambda(\vec{xp}))$ and $(\overline{H}'_\lambda(\vec{xp}), \overline{\mathcal{D}}'_\lambda(\vec{xp}))$ are isomorphic, with the isomorphism preserving $\mathcal{R}_{\mathrm{skel}}(p\vec{x})$ and mapping each vertex $v \in \bigcup \mathcal{D}_\lambda(\vec{xp}) = V(\mathbb{H}_{\vec{xp}})$ to $\pi(v)$. Therefore, for every $C \in \mathcal{C}_{\vec{xp}}$,

$$\mathsf{cutrk}_{\overline{H}'_\lambda(\vec{xp})}(R_C) = \mathsf{cutrk}_{\overline{H}_\lambda(\vec{xp})}(\pi^{-1}(R_C)).$$

Since $\mathcal{D}_\lambda(\vec{xp}) = \{\pi^{-1}(R_C) \mid C \in \mathcal{C}_{\vec{xp}}\}$, we conclude that

$$\begin{aligned}\sum_{C\in\mathcal{C}_{\vec{xp}}} \mathsf{cutrk}_G(C) &= \sum_{C\in\mathcal{C}_{\vec{xp}}} \mathsf{cutrk}_{\overline{H}_\lambda(\vec{xp})}(\pi^{-1}(R_C)) \\ &= \sum_{C\in\mathcal{D}_\lambda(\vec{xp})} \mathsf{cutrk}_{\overline{H}_\lambda(\vec{xp})}(C) = \mathsf{ccost}(\mathbb{H}_\lambda(\vec{xp}), \vec{xp}).\end{aligned} \tag{4.14}$$

We get the required equality by summing Eq. (4.14) for all $\vec{xp} \in A$ and recalling that $r = \sum_{C \in \mathcal{C}} \mathsf{cutrk}_G(C)$ and $r_\lambda = \sum_{\vec{xp} \in A} \mathsf{ccost}(\mathbb{H}_\lambda(\vec{xp}), \vec{xp})$.

- $q_\lambda \leqslant q - |T_{\mathrm{pref}}|$: This follows immediately from the facts that $q = |T_{\mathrm{pref}}| + \sum_{\vec{xp} \in A} q'_{\vec{xp}}$ and that $q_{\vec{xp}} \leqslant q'_{\vec{xp}}$ for each $\vec{xp} \in A$.

Therefore, the proof is complete. ◁

Rank decompositions of partitioned graphs $(G_\lambda, \mathcal{D}_\lambda)$. We now show how, for any mapping $\lambda \in \Lambda$, we produce a rank decomposition of the partitioned graph $(G_\lambda, \mathcal{D}_\lambda)$.

Consider an edge $\vec{xp} \in A$ and a pair $(q, \mathbb{H}) \in \mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp})$. For technical reasons, we will now rename vertices of $\mathbb{H}$ so as to ensure that $\mathbb{H}$ contains all vertices of $\mathcal{R}(\vec{xp})$. We construct a graph $\mathbb{H}^\star$ from $\mathbb{H}$ as follows: For every vertex $u \in \mathcal{R}(\vec{xp})$ such that $u \notin V(\mathbb{H})$, choose any vertex $v \in V(\mathbb{H})$ such that $\eta(\mathbb{H})(v) = u$ (such a vertex exists since $\mathbb{H}$ encodes some partition of $\mathcal{L}(\mathcal{T})[\vec{xp}]$ and $\mathcal{R}(\vec{xp})$ is a minimum representative of $\mathcal{L}(\mathcal{T})[\vec{xp}]$), and rename v to u. Let also $\pi_{\mathbb{H}}$ be the isomorphism from $\mathbb{H}^\star$ to $\mathbb{H}$ prescribed by the procedure above. Naturally, this construction ensures that $\mathbb{H}^\star$ is isomorphic to $\mathbb{H}$ (but we stress that there could be $u \in V_i(\mathbb{H}^\star)$ and $v \in V_j(\mathbb{H}^\star)$ with $i \neq j$ such that $uv \in E(\mathbb{H}^\star) \nLeftrightarrow uv \in E(G)$). By the properties of $\eta(\mathbb{H})$, we have that, for every $u \in \mathcal{R}(\vec{xp})$,

$$N_G(u) \cap \mathcal{L}(\mathcal{T})[p\vec{x}] = N_G(\pi_{\mathbb{H}}(u)) \cap \mathcal{L}(\mathcal{T})[p\vec{x}]. \tag{4.15}$$

Given an edge $\vec{xp} \in A$ and a pair $(q, \mathbb{H}) \in \mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp})$, define now an annotated rank decomposition derived from $\mathbb{H}^\star$, denoted $\mathcal{T}_{\mathbb{H}^\star}$, as follows. Recall that $(H, \mathcal{D})$ is the partitioned graph derived from $\mathbb{H}$ and $\overline{\mathcal{D}} = \mathcal{D} \cup \{\mathcal{R}_{\mathrm{skel}}(p\vec{x})\}$, and $\overline{H} = G[\overline{\mathcal{D}}]$. Then define $(\overline{H}^\star, \overline{\mathcal{D}}^\star)$ as the partitioned graph created from $(\overline{H}, \overline{\mathcal{D}})$ by renaming each vertex $v \in V(H)$ to $\pi_{\mathbb{H}}^{-1}(v)$. Note that by the construction of $\overline{H}^\star$, we have that $\mathcal{R}_{\mathrm{skel}}(\vec{xp}) \cup \mathcal{R}_{\mathrm{skel}}(p\vec{x}) \subseteq V(\overline{H}^\star)$ and moreover $\mathcal{R}_{\mathrm{skel}}(p\vec{x}) \in \overline{\mathcal{D}}^\star$. Then let $\mathcal{T}_{\mathbb{H}^\star} = (T_{\mathbb{H}^\star}, U_{\mathbb{H}^\star}, \mathcal{R}_{\mathbb{H}^\star}, \mathcal{E}_{\mathbb{H}^\star}, \mathcal{F}_{\mathbb{H}^\star})$ be an arbitrary annotated rank decomposition of $(\overline{H}^\star, \overline{\mathcal{D}}^\star)$ with the following properties:

- $V(T_{\mathbb{H}^\star}) \cap V(T_{\mathrm{skel}}) = \{x, p\}$ and $p\vec{x}$ is a leaf edge of $T_{\mathbb{H}^\star}$;
- $\mathcal{L}(\mathcal{T}_{\mathbb{H}^\star})[p\vec{x}] = \mathcal{R}_{\mathbb{H}^\star}(p\vec{x}) = \mathcal{R}_{\mathrm{skel}}(p\vec{x})$ and $\mathcal{R}_{\mathbb{H}^\star}(\vec{xp}) = \mathcal{R}_{\mathrm{skel}}(\vec{xp}) = \mathcal{R}(\vec{xp})$.

It can be easily seen that such a decomposition exists and can be constructed from $\mathbb{H}$ and the annotations on the edge xp of $\mathcal{T}$ in time $\mathcal{O}_{c,\ell}(1)$. Observe also that $\mathcal{E}_{\mathbb{H}^\star}(xp) = \mathcal{E}_{\mathrm{skel}}(xp)$: For any pair of vertices $u \in \mathcal{R}_{\mathrm{skel}}(\vec{xp})$, $v \in \mathcal{R}_{\mathrm{skel}}(p\vec{x})$ we have $uv \in E(\mathcal{E}_{\mathbb{H}^\star}(xp))$ if and only if $\pi_{\mathbb{H}}(u)v \in E(G)$ by the definition of $\mathbb{H}^\star$. But by Eq. (4.15), $\pi_{\mathbb{H}}(u)v \in E(G)$ if and only if $uv \in E(G)$, which holds if and only if $uv \in E(\mathcal{E}_{\mathrm{skel}}(xp))$. Next, since $|\bigcup \overline{\mathcal{D}}^\star| = |V(\mathbb{H}^\star)| + |\mathcal{R}_{\mathrm{skel}}(p\vec{x})| \leqslant cs\ell + \ell$, the width of $\mathcal{T}_{\mathbb{H}^\star}$ is bounded by $cs\ell + \ell$. Let also $\mathcal{T}_{\mathbb{H}}$ be the decomposition formed from $\mathcal{T}_{\mathbb{H}^\star}$ by renaming all vertices $v \in V(\mathbb{H}^\star)$ of the graph encoded by the decomposition back to $\pi_{\mathbb{H}}(v)$. Naturally, $\mathcal{T}_{\mathbb{H}}$ encodes $(\overline{H}, \overline{\mathcal{D}}) = (G[\overline{\mathcal{D}}], \overline{\mathcal{D}})$.

Next, for any $\lambda \in \Lambda$, define the following rank decompositions:

- $\mathcal{T}^\star_\lambda$ – the decomposition formed by gluing $\mathcal{T}_{\mathrm{skel}}$ along xp with each decomposition $\mathcal{T}_{\mathbb{H}^\star_{\lambda(\vec{xp})}} = (T_{\mathbb{H}^\star_{\lambda(\vec{xp})}}, U_{\mathbb{H}^\star_{\lambda(\vec{xp})}}, \mathcal{R}_{\mathbb{H}^\star_{\lambda(\vec{xp})}}, \mathcal{E}_{\mathbb{H}^\star_{\lambda(\vec{xp})}}, \mathcal{F}_{\mathbb{H}^\star_{\lambda(\vec{xp})}})$ for $\vec{xp} \in A$ in arbitrary order; this gluing is possible since for every $\vec{xp} \in A$, we have $\vec{xp} \in \vec{L}(T_{\mathrm{skel}})$, $p\vec{x} \in \vec{L}(T_{\mathbb{H}^\star})$, $\mathcal{R}_{\mathbb{H}^\star_{\lambda(\vec{xp})}}(p\vec{x}) = \mathcal{R}_{\mathrm{skel}}(p\vec{x})$, $\mathcal{R}_{\mathbb{H}^\star_{\lambda(\vec{xp})}}(\vec{xp}) = \mathcal{R}_{\mathrm{skel}}(\vec{xp})$ and $\mathcal{E}_{\mathbb{H}^\star_{\lambda(\vec{xp})}}(xp) = \mathcal{R}_{\mathrm{skel}}(xp)$. It is easy to see that $\mathcal{T}^\star_\lambda$ encodes some partitioned graph with vertex set $\bigcup_{\vec{xp} \in A} V(H^\star_{\lambda(\vec{xp})})$. Moreover, its width is bounded by $cs\ell + \ell$ as discussed at the introduction of the notion of gluing decompositions.
- $\mathcal{T}_\lambda$ – the decomposition formed from $\mathcal{T}^\star_\lambda$ by renaming every vertex v in the partitioned graph encoded by $\mathcal{T}^\star_\lambda$ such that $v \in V(H^\star_{\lambda(\vec{xp})})$ for $\vec{xp} \in A$ back to $\pi_{\mathbb{H}_{\lambda(\vec{xp})}}(v)$. Of course, the width of $\mathcal{T}_\lambda$ is also bounded by $cs\ell + \ell$.

The following observation follows straight from the analysis of the construction of $\mathcal{T}_\lambda$ and $\mathcal{T}^\star_\lambda$.

Observation 4.9.22. *$\mathcal{T}_\lambda$ encodes the partitioned graph $(G_\lambda, \mathcal{D}_\lambda)$, and $\mathcal{T}^\star_\lambda$ encodes a partitioned graph isomorphic to $(G_\lambda, \mathcal{D}_\lambda)$.*

Optimizing λ. At this point of time, we have reduced the problem to finding a mapping $\lambda \in \Lambda$ with the rankwidth of $(G_\lambda, \mathcal{D}_\lambda)$ bounded by $2k$, such that the pair (r_λ, q_λ) is lexicographically minimum possible. In the sequel, we will show how this can be done using the exact rankwidth automaton $\mathcal{JKO} = (Q, \iota, \delta, \varepsilon)$.

We now briefly sketch the idea. A brute-force search for an optimum λ would look as follows: Recall that $\mathcal{T}_\lambda$ is an annotated rank decomposition of $(G_\lambda, \mathcal{D}_\lambda)$ of width $cs\ell + \ell$. Hence running $\mathcal{JKO}$ on $\mathcal{T}_\lambda$ will correctly determine whether the rankwidth of the encoded partitioned graph $(G_\lambda, \mathcal{D}_\lambda)$ is at most $2k$. Repeating this procedure for all possible $\lambda \in \Lambda$ yields all viable mappings λ; for each of these, we can easily compute the values r_λ and q_λ – each of these is of the form $q_\lambda = \sum_{\vec{xp} \in A} f_{\vec{xp}}(\lambda(\vec{xp}))$ and $r_\lambda = \sum_{\vec{xp} \in A} g_{\vec{xp}}(\lambda(\vec{xp}))$ for some functions $f_{\vec{xp}}$, $g_{\vec{xp}}$ that can be evaluated efficiently given $\lambda(\vec{xp})$. Thus we can find the optimum mapping λ.

Note that in the description above, instead of the annotated decomposition $\mathcal{T}_\lambda$ encoding $(G_\lambda, \mathcal{D}_\lambda)$, we could have used an annotated decomposition $\mathcal{T}^\star_\lambda$ encoding a partitioned graph isomorphic to $(G_\lambda, \mathcal{D}_\lambda)$. Then $\mathcal{JKO}$, when run on $\mathcal{T}^\star_\lambda$, will return that the encoded partitioned graph has rankwidth at most $2k$ if and only if it would do so when run on $\mathcal{T}_\lambda$. This choice has an important consequence: All annotated decompositions $\mathcal{T}^\star_\lambda$ have *the same* annotated prefix. Formally, given two annotated rank decompositions $\mathcal{T}_1 = (T_1, U_1, \mathcal{R}_1, \mathcal{E}_1, \mathcal{F}_1)$ and $\mathcal{T}_2 = (T_2, U_2, \mathcal{R}_2, \mathcal{E}_2, \mathcal{F}_2)$ and a set $S \subseteq V(T_1) \cap V(T_2)$, we say that $\mathcal{T}_1$ and $\mathcal{T}_2$ *agree on* S if

$$\begin{aligned} T_1[S] &= T_2[S], \\ \mathcal{R}_1|_{\vec{E}(T_1[S])} &= \mathcal{R}_2|_{\vec{E}(T_2[S])}, \\ \mathcal{E}_1|_{E(T_1[S])} &= \mathcal{E}_2|_{E(T_2[S])}, \\ \mathcal{F}_1|_{\mathcal{P}_3(T_1[S])} &= \mathcal{F}_2|_{\mathcal{P}_3(T_2[S])}. \end{aligned}$$

Then, for any $\lambda_1, \lambda_2 \in \Lambda$, the decompositions $\mathcal{T}^\star_{\lambda_1}$ and $\mathcal{T}^\star_{\lambda_2}$ agree on $T'_{\mathrm{pref}} := T_{\mathrm{pref}} \cup \mathsf{App}_T(T_{\mathrm{pref}})$. This observation will allow us to reuse the partial runs of $\mathcal{JKO}$, which will enable us to find the optimum mapping λ by means of a dynamic programming on the rooted subtree induced by T'_{pref} (precisely, using Lemma 4.9.19). The details can be found below.

Let $\vec{xp} \in A$ and $(q, \mathbb{H}) \in \mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp})$. Define the state $\xi_{\mathbb{H}^\star} \in Q$ of $\mathcal{JKO}$ as follows. Recall that $\vec{px}$ is the unique leaf edge of $\mathcal{T}_{\mathbb{H}^\star}$ such that $\mathcal{L}(\mathcal{T}_{\mathbb{H}^\star})[\vec{px}] = \mathcal{R}_{\mathrm{skel}}(\vec{px})$ (and so $\mathcal{R}_{\mathbb{H}^\star}(\vec{px}) = \mathcal{R}_{\mathrm{skel}}(\vec{px})$ and $\mathcal{R}_{\mathbb{H}^\star}(\vec{xp}) = \mathcal{R}_{\mathrm{skel}}(\vec{xp})$). Then let $\rho_{\mathbb{H}^\star}$ be the run of $\mathcal{JKO}$ on $(\mathcal{T}_{\mathbb{H}^\star}, x, p)$, and set $\xi_{\mathbb{H}^\star} := \rho_{\mathbb{H}^\star}(\vec{xp})$. Note that $\xi_{\mathbb{H}^\star}$ can be determined in time $\mathcal{O}_{c,\ell}(1)$.

We now claim that in a run of $\mathcal{JKO}$ on $\mathcal{T}^\star_\lambda$ for some $\lambda \in \Lambda$, the partial runs on the glued decompositions $\mathcal{T}_{\mathbb{H}^\star_{\lambda(\vec{xp})}}$ are exactly the recorded states $\xi_{\mathbb{H}^\star_{\lambda(\vec{xp})}}$.

Claim 4.9.23. *Let $\lambda \in \Lambda$ and $\vec{xp} \in A$. If ρ is the run of $\mathcal{JKO}$ on $\mathcal{T}^\star_\lambda$, then $\rho(\vec{xp}) = \xi_{\mathbb{H}^\star_{\lambda(\vec{xp})}}$.*

Proof of the claim. Observe that the set $B := V(\mathcal{T}_{\mathbb{H}^\star_{\lambda(\vec{xp})}})$ comprises exactly p and the set of descendants of x in $T^\star_\lambda$. Moreover, by the construction of $\mathcal{T}^\star_\lambda$ (and the properties of gluing decompositions), we get that the decompositions $\mathcal{T}_{\mathbb{H}^\star_{\lambda(\vec{xp})}}$ and $\mathcal{T}^\star_\lambda$ agree on B. We immediately infer that $\rho(\vec{xp}) = \rho_{\mathbb{H}^\star_{\lambda(\vec{xp})}}(\vec{xp}) = \xi_{\mathbb{H}^\star_{\lambda(\vec{xp})}}$. ◁

Aiming to use Lemma 4.9.19 in our case, let $\overline{\mathbb{Z}} := \mathbb{Z} \cup \{+\infty\}$ and define the totally ordered commutative semigroup $(S, +, \leqslant)$, where $S = \overline{\mathbb{Z}} \times \overline{\mathbb{Z}}$, $+$ is the coordinate-wise sum and $\leqslant$ is the lexicographic order on S. Then define the cost function $\mathbf{c} : A \times Q \to S$ by setting, for every $\vec{xp} \in A$ and $f \in Q$, the value

$$\mathbf{c}(\vec{xp}, f) = \min\{(\mathsf{ccost}(\mathbb{H}, \vec{xp}), q) \mid (\mathbb{H}, q) \in \mathsf{reps}^{c,s}(\vec{xp}),\ \xi_{\mathbb{H}^\star} = f\}; \tag{4.16}$$

where we set $\mathbf{c}(\vec{xp}, f) = (+\infty, +\infty)$ if the set on the right-hand side of Eq. (4.16) is empty. Let also $F \subseteq Q$ be the set of states of $\mathcal{JKO}$ accepting that the input decomposition describes a partitioned graph of rankwidth at most $2k$; or equivalently, F is the set of states representing nonempty full sets of width $2k$ at a root of an input decomposition.

We now show that the results of Lemma 4.9.19 will be enough to determine the existence of λ with the rankwidth of $(G_\lambda, \mathcal{D}_\lambda)$ bounded by $2k$, and in the case any such λ exists – to determine an optimum mapping λ.

Claim 4.9.24. *Suppose $\lambda \in \Lambda$ is such that $(G_\lambda, \mathcal{D}_\lambda)$ has rankwidth at most $2k$. Let $\kappa : A \to Q$ be defined as $\kappa(\vec{xp}) = \xi_{\mathbb{H}^\star_{\lambda(\vec{xp})}}$ for each $\vec{xp} \in A$, and let ρ_κ be the κ-run of $\mathcal{A}$ on $\mathcal{T}_{\mathrm{skel}}$. Then $\rho_\kappa(\vartheta) \in F$ and $\mathbf{c}(\kappa) \leqslant (r_\lambda, q_\lambda)$.*

Proof of the claim. Let also ρ be the run of $\mathcal{JKO}$ on $\mathcal{T}^\star_\lambda$. By Claim 4.9.23, we have $\rho(\vec{xp}) = \xi_{\mathbb{H}^\star_{\lambda(\vec{xp})}} = \kappa(\vec{xp})$. Since $\mathcal{T}^\star_\lambda$ and $\mathcal{T}_{\mathrm{skel}}$ agree on $V(\mathcal{T}_{\mathrm{skel}})$, we infer that $\rho_\kappa(\vartheta) = \rho(\vartheta)$. Therefore, $\rho_\kappa(\vartheta) \in F$ if and only if $\rho(\vartheta) \in F$, which only holds when the full set of $\mathcal{T}^\star_\lambda$ at the root r of width at most $2k$ is nonempty (i.e., $(G_\lambda, \mathcal{D}_\lambda)$ has rankwidth at most $2k$). So $\rho_\kappa(\vartheta) \in F$. Since $\mathbf{c}(\vec{xp}, \kappa(\vec{xp})) \leqslant (r_\lambda(\vec{xp}), q_\lambda(\vec{xp}))$ for each $\vec{xp} \in A$ (by Eq. (4.16)), $\mathbf{c}(\kappa) = \sum_{\vec{xp}\in A} \mathbf{c}(\vec{xp}, \kappa(\vec{xp}))$, $r_\lambda = \sum_{\vec{xp}\in A} r_\lambda(\vec{xp})$ and $q_\lambda = \sum_{\vec{xp}\in A} q_\lambda(\vec{xp})$, we conclude that $\mathbf{c}(\kappa) \leqslant (r_\lambda, q_\lambda)$. ◁

Claim 4.9.25. *Suppose there exists a leaf edge state mapping $\kappa : A \to Q$ such that $\mathbf{c}(\kappa) \neq (+\infty, +\infty)$ and, for the κ-run ρ_κ of $\mathcal{A}$ on $(\overline{\mathcal{T}}_{\mathrm{skel}}, r_1, r_2)$, we have $\rho_\kappa(\vartheta) \in F$. Then there exists $\lambda \in \Lambda$ such that $(G_\lambda, \mathcal{D}_\lambda)$ has rankwidth at most $2k$ and $(r_\lambda, q_\lambda) = \mathbf{c}(\kappa)$. Moreover, λ can be constructed in time $\mathcal{O}_{c,\ell}(|T_{\mathrm{pref}}|)$.*

Proof of the claim. Construct a valuation $\lambda \in \Lambda$ as follows. For every $\vec{xp} \in A$, choose $\lambda(\vec{xp})$ to be such a pair $(\mathbb{H}, q) \in \mathsf{reps}^{c,s}(\vec{xp})$ that $\xi_{\mathbb{H}^\star} = \kappa(\vec{xp})$ and $(\mathsf{ccost}(\mathbb{H}, \vec{xp}), q) = \mathbf{c}(\vec{xp}, \kappa(\vec{xp}))$. Repeating the same argument involving Claim 4.9.23 as before, we find that since $\rho_\kappa(\vartheta) \in F$, we have that $(G_\lambda, \mathcal{D}_\lambda)$ has rankwidth at most $2k$. We also easily verify that $\mathbf{c}(\kappa) = (r_\lambda, q_\lambda)$. ◁

Apply now Lemma 4.9.19 for the automaton $\mathcal{JKO}$, the semigroup $(S, +, \leqslant)$, the decomposition $\mathcal{T}_{\mathrm{skel}}$, the cost function $\mathbf{c}$, and the set of accepting states F. The algorithm of Lemma 4.9.19 runs in time $\mathcal{O}_{c,\ell}(|T_{\mathrm{skel}}|) = \mathcal{O}_{c,\ell}(|T_{\mathrm{pref}}|)$ and returns one of the following:

- *there is no mapping $\kappa : A \to Q$ such that $\rho_\kappa(\vartheta) \in F$ where ρ_κ is the κ-run on $\mathcal{T}_{\mathrm{skel}}$, or the cost of all such mappings is $(+\infty, +\infty)$.* Then by Claim 4.9.24 there exists no $\lambda \in \Lambda$ with the property that the rankwidth of $(G_\lambda, \mathcal{D}_\lambda)$ is at most $2k$; hence we can return that T_{pref} has no c-small k-closure.
- *$\kappa : A \to Q$ is the minimum-cost mapping such that $\rho_\kappa(\vartheta) \in F$ where ρ_κ is the κ-run on $\mathcal{T}_{\mathrm{skel}}$, and the cost of the mapping is finite.* Then we reconstruct the mapping $\lambda \in \Lambda$ in time $\mathcal{O}_{c,\ell}(|T_{\mathrm{pref}}|)$ such that $(r_\lambda, q_\lambda) = \mathbf{c}(\kappa)$ using Claim 4.9.25. By Claim 4.9.24, such a mapping has the minimum value of r_λ; and among all such optimal mappings, it also has the minimum possible value of q_λ.

Finally, using Claim 4.9.21, we conclude that:

Corollary 4.9.26. *In time $\mathcal{O}_{c,\ell}(|T_{\mathrm{pref}}|)$, we can:*

- *correctly decide that T_{pref} has no c-small k-closure; or*
- *find a mapping $\lambda \in \Lambda$ such that $\mathcal{D}_\lambda$ represents some minimal c-small k-closure of T_{pref}. Moreover, for every partition $\mathcal{C}$ of $V(G)$ defined as $\mathcal{C} = \bigcup_{\vec{xp}\in A} \mathcal{C}_{\vec{xp}}$, where $\mathcal{C}_{\vec{xp}}$ is a partition of $\mathcal{L}(\mathcal{T})[\vec{xp}]$ into at most c sets encoded by $\mathbb{H}_\lambda(\vec{xp})$ and of cost $q_\lambda(\vec{xp})$, $\mathcal{C}$ is a minimal c-small k-closure of T_{pref}.*

Reconstructing the closure. Having found λ, we want now to reconstruct any minimal c-small k-closure $\mathcal{C}$ of T_{pref}. Recall that, since we cannot afford to compute $\mathcal{C}$ explicitly (since a closure is essentially an arbitrary partition of $V(G)$), we are required to return the closure in a compact form – precisely, the sets $\mathsf{cut}_T(\mathcal{C})$ and $\mathsf{aep}_T(\mathcal{C})$, that is the prefix of T cut by $\mathcal{C}$ and the appendix edge partition of $\mathcal{C}$. The procedure should work in time $\mathcal{O}_{c,\ell}(|\mathsf{cut}_T(\mathcal{C})|)$.

Let $\vec{xp} \in A$ and recall that $(q_\lambda(\vec{xp}), \mathbb{H}_\lambda(\vec{xp})) = \lambda(\vec{xp}) \in \mathsf{reps}^{c,s}(\mathcal{T}, \vec{xp})$. We will now present a subroutine finding a partition $\mathcal{C}_{\vec{xp}}$ of $\mathcal{L}(\mathcal{T})[\vec{xp}]$, represented implicitly as $\mathsf{aep}_T(\mathcal{C}_{\vec{xp}})$, so that $\mathcal{C}_{\vec{xp}}$ is of cost $q_\lambda(\vec{xp})$ and is encoded by $\mathbb{H}_\lambda(\vec{xp})$.

At the start of the subroutine, we initialize a sequence of initially empty pairwise disjoint subsets $(V_1, \ldots, V_c)$ of $\mathcal{L}(\mathcal{T})[\vec{xp}]$; eventually, $(V_1, \ldots, V_c)$ will form an indexed partition of $\mathcal{L}(\mathcal{T})[\vec{xp}]$. The sets $V_1, \ldots, V_c$ are represented implicitly by sets $E_1, \ldots, E_c$ of oriented edges of T with the property that $E_i = \mathsf{aes}_T(V_i)$ for each $i \in [c]$. We now implement a recursive function $\textsc{Populate}(\vec{ab}, q, \mathbb{H})$ that, under the assumptions that $\vec{ab}$ is a predecessor of $\vec{xp}$ in T and $(q, \mathbb{H}) \in \mathsf{reps}^{c,s}(\mathcal{T}, \vec{ab})$, adds to each set $V_1, \ldots, V_c$ a subset $V_1^{\vec{ab}}, \ldots, V_c^{\vec{ab}}$, respectively, so that $(V_1^{\vec{ab}}, \ldots, V_c^{\vec{ab}})$ is an indexed partition of $\mathcal{L}(\mathcal{T})[\vec{ab}]$ of cost q encoded by $\mathbb{H}$. (Note that such an indexed partition must exist by the assumptions.) In the implementation, we consider two cases.

- If $q = 0$, then no node of the subtree of T rooted at $\vec{ab}$ may be cut by $(V_1^{\vec{ab}}, \ldots, V_c^{\vec{ab}})$. That is, the entire subset $\mathcal{L}(\mathcal{T})[\vec{ab}]$ belongs to one of the sets $V_j^{\vec{ab}}$. Here, the value j can be found in constant time since it is exactly the unique index j such that $V_j(\mathbb{H}) \neq \emptyset$. So we add $\vec{ab}$ to E_j and we are done.

- If $q \geqslant 1$, then some nodes of the subtree of T rooted at $\vec{ab}$ are cut by $(V_1^{\vec{ab}}, \ldots, V_c^{\vec{ab}})$; in particular, one of these nodes must be a, and moreover, $\vec{ab}$ cannot be a leaf edge of T and so $\vec{ab}$ has two children $y_1\vec{a}, y_2\vec{a}$. In constant time (using the dynamic data structure of Lemma 4.5.1 maintaining $\mathcal{CR}$ on $\mathcal{T}$ dynamically), we read the value $\rho(\vec{ab})$, where ρ is the run of $\mathcal{CR}$ on $\mathcal{T}$. By Lemma 4.9.13, the value $\rho(\vec{ab})$ contains a mapping Φ; let $((q_1, \mathbb{H}_1), (q_2, \mathbb{H}_2)) = \Phi((q, \mathbb{H}))$ such that $(q_1, \mathbb{H}_1) \in \mathsf{reps}^{c,s}(\mathcal{T}, y_1\vec{a})$ and $(q_2, \mathbb{H}_2) \in \mathsf{reps}^{c,s}(\mathcal{T}, y_2\vec{a})$. We then run $\textsc{Populate}(y_1\vec{a}, q_1, \mathbb{H}_1)$ and $\textsc{Populate}(y_2\vec{a}, q_2, \mathbb{H}_2)$ and add a to $\mathsf{cut}_T(\mathcal{C})$.

  The two recursive calls add to the sets $V_1, \ldots, V_c$ the subsets $V_1^{y_1\vec{a}}, \ldots, V_c^{y_1\vec{a}}$ and $V_1^{y_2\vec{a}}, \ldots, V_c^{y_2\vec{a}}$, respectively, with the property that for each $t \in [2]$, the sequence $(V_1^{y_t\vec{a}}, \ldots, V_c^{y_t\vec{a}})$ is an indexed partition of $\mathcal{L}(\mathcal{T})[y_t\vec{a}]$ of cost q_t encoded by $\mathbb{H}_t$. So again by Lemma 4.9.13, the sequence $V_1^{\vec{ab}}, \ldots, V_c^{\vec{ab}}$ given by $V_j^{\vec{ab}} = V_j^{y_1\vec{a}} \cup V_j^{y_2\vec{a}}$ is an indexed partition of $\mathcal{L}(\mathcal{T})[\vec{ab}]$ of cost q encoded by $\mathbb{H}$. Since the recursive calls already added each set $V_j^{\vec{ab}}$ to V_j, we are done.

Thus running $\textsc{Populate}(\vec{xp}, q_\lambda(\vec{xp}), \mathbb{H}_\lambda(\vec{xp}))$ will create an indexed partition $(V_1, \ldots, V_c)$ of $\mathcal{L}(\mathcal{T})[\vec{xp}]$ of cost $q_\lambda(\vec{xp})$ encoded by $\mathbb{H}_\lambda(\vec{xp})$; the partition is stored implicitly as sets $E_1, \ldots, E_c$. So letting $\mathcal{C}_{\vec{xp}} = \{V_1, \ldots, V_c\} \setminus \{\emptyset\}$, the nonempty sets in $E_1, \ldots, E_c$ form $\mathsf{aep}_T(\mathcal{C}_{\vec{xp}})$. Tracing the execution of $\textsc{Populate}$, it is easy to verify that this set $\mathsf{aep}_T(\mathcal{C}_{\vec{xp}})$ can be computed in time $\mathcal{O}_{c,\ell}(|\mathsf{cut}_{\vec{xp}}(\mathcal{C}_{\vec{xp}})| + 1)$, where $\mathsf{cut}_{\vec{xp}}(\mathcal{C}_{\vec{xp}})$ is the set of nodes of T that are children of $\vec{xp}$ that are cut by $\mathcal{C}_{\vec{xp}}$.

Now let $\mathcal{C} \coloneqq \bigcup_{\vec{xp} \in A} \mathcal{C}_{\vec{xp}}$, so that $\mathcal{C}$ is encoded by $\mathsf{aep}_T(\mathcal{C}) \coloneqq \bigcup_{\vec{xp} \in A} \mathsf{aep}_T(\mathcal{C}_{\vec{xp}})$. Then by Corollary 4.9.26, $\mathcal{C}$ is indeed a minimal c-small k-closure of T_{pref}. The set of nodes cut by $\mathcal{C}$ is exactly $\mathsf{cut}_T(\mathcal{C}) = T_{\mathrm{pref}} \cup \bigcup_{\vec{xp} \in A} \mathsf{cut}_{\vec{xp}}(\mathcal{C}_{\vec{xp}})$. The set $\mathsf{aep}_T(\mathcal{C})$ can be found by invoking the function $\textsc{Populate}(\vec{xp}, q_\lambda(\vec{xp}), \mathbb{H}_\lambda(\vec{xp}))$ for each $\vec{xp} \in A$ separately and gathering the nonempty sets of edges after each call. The time complexity of all recursive calls is bounded by

$$\mathcal{O}_{c,\ell}\Big(\sum_{\vec{xp} \in A} |\mathsf{cut}_{\vec{xp}}(\mathcal{C}_{\vec{xp}})| + 1\Big) \leqslant \mathcal{O}_{c,\ell}\Big(|T_{\mathrm{pref}}| + \sum_{\vec{xp} \in A} |\mathsf{cut}_{\vec{xp}}(\mathcal{C}_{\vec{xp}})|\Big) = \mathcal{O}_{c,\ell}(|\mathsf{cut}_T(\mathcal{C})|),$$

since $|A| = |T_{\mathrm{pref}}| + 1$. This finishes the description of the effective reconstruction of $\mathsf{cut}_T(\mathcal{C})$ and $\mathsf{aep}_T(\mathcal{C})$.

Obtaining the decomposition of the closure. The final object we are required to return is a rank decomposition $(T^\star, \lambda^\star)$ of $(G[\mathcal{C}], \mathcal{C})$ of width at most $2k$. Remembering that the partition $\mathcal{C}$ reconstructed a moment ago is represented by $\mathcal{D}_\lambda$, we observe that the task at hand can be accomplished by:

- computing a rank decomposition $(T^\square, \lambda^\square)$ of $(G_\lambda, \mathcal{D}_\lambda)$ of width at most $2k$, and
- producing a rank decomposition $(T^\star, \lambda^\star)$ of $(G[\mathcal{C}], \mathcal{C})$ by setting $T^\star \coloneqq T^\square$ and setting $\lambda^\star(C) \coloneqq \lambda^\square(R_C)$ for every $C \in \mathcal{C}$, where $R_C \in \mathcal{C}_\lambda$ is a representative of C in G.

The former step is done by constructing the annotated decomposition $\mathcal{T}_\lambda$ of $(G_\lambda, \mathcal{D}_\lambda)$ (of width at most $cs\ell + \ell = \mathcal{O}_{c,\ell}(1)$) explicitly in time $\mathcal{O}_{c,\ell}(|T_{\mathrm{pref}}|)$. Since the rankwidth of $(G_\lambda, \mathcal{D}_\lambda)$ – equal to the rankwidth of $(G[\mathcal{C}], \mathcal{C})$ – is at most $2k$, we apply Lemma 4.9.12 in time $\mathcal{O}_{c,\ell}(|T_{\mathrm{pref}}|)$ and we are done. The latter step can then be performed in time $\mathcal{O}(|T_{\mathrm{pref}}|)$ as long as $\mathcal{C}$ is represented by $\mathsf{aep}_T(\mathcal{C})$; or in other words, $\lambda^\star$ is represented as a function $\lambda\colon \mathsf{aep}_T(\mathcal{C}) \to \vec{L}(T^\star)$. This concludes the proof of Lemma 4.4.7. □

4.10 Cliquewidth

In this section we recall the definition of cliquewidth, show that annotated rank decompositions can be translated into cliquewidth expressions, show that automata working on cliquewidth expressions can be translated into rank decomposition automata, and use this to translate known dynamic programming algorithms on cliquewidth to rank decomposition automata.

4.10.1 Definition and k-expressions

Recall that a tuple $\mathcal{G} = (G, V_1, \ldots, V_k)$ is a *k-graph* if G is a graph and $V_1, \ldots, V_k$ are (potentially empty) disjoint subsets of $V(G)$ whose union equals $V(G)$. For two k-graphs $\mathcal{G}^1$ and $\mathcal{G}^2$ where $\mathcal{G}^1$ and $\mathcal{G}^2$ are vertex-disjoint, we define the disjoint union $\mathcal{G}^1 \oplus \mathcal{G}_2$. For a k-graph $\mathcal{G} = (G, V_1, \ldots, V_k)$ and $i, j \in [k]$ with $i \neq j$, we denote by $\eta(i,j)(\mathcal{G})$ the k-graph obtained from $\mathcal{G}$ by adding a biclique between V_i and V_j. Also,

$\pi(i,j)(\mathcal{G})$ for $i,j \in [k]$ with $i \neq j$ denotes the k-graph obtained from $\mathcal{G}$ by renaming i into j. Then a graph has cliquewidth at most k if it can be constructed from single-vertex k-graphs by using these operations.

More formally, we let $\mathsf{op}_k = \{\oplus\} \cup \bigcup_{i,j\in[k]|i\neq j}\{\eta(i,j), \pi(i,j)\}$ denote the set of operations on k-graphs. We define that *k-expression* is a triple $\mathsf{Expr} = (T, U, \mu)$, where T is a rooted tree whose every node has at most two children and $\mu\colon V(T) \to U \cup \mathsf{op}_k$ is a labeling of its nodes so that

- the restriction $\mu|_{L(T)}$ of μ to the leaves of T is a bijection $\mu|_{L(T)}\colon L(T) \to U$,
- every node t with one child is labeled with $\mu(t) \in \mathsf{op}_k \setminus (U \cup \{\oplus\})$ for some $i,j \in [k]$ with $i \neq j$, and
- every node t with two children is labeled with $\mu(t) = \oplus$.

We recursively define that a node $t \in V(T)$ *encodes* a k-graph $\zeta(t) = (G, V_1, \ldots, V_k)$ if

- t is a leaf, G is the graph with a single vertex $\mu(t)$, and $V_1 = V(G)$,
- t has one child c and $\zeta(t) = \mu(t)(\zeta(c))$, or
- t has two children c_1, c_2 and $\zeta(t) = \zeta(c_1) \oplus \zeta(c_2)$.

We say that Expr encodes a graph G if its root encodes a k-graph $(G, V_1, \ldots, V_k)$ for some $V_1, \ldots, V_k$. We note that if Expr encodes G, then $V(G) = U$. Now the more formal definition of cliquewidth is that the cliquewidth of G is the smallest k so that there exists a k-expression that encodes G.

Then we prove that an annotated rank decompositions of width k that encodes a graph G can be turned in $\mathcal{O}_k(n)$ time to a $(2^{k+1}-1)$-expression that encodes G. Our proof follows the original construction of Oum and Seymour [OS06], but optimizes it to linear time in the case of annotated rank decompositions. The definitions and auxiliary lemmas used for proving this will also be used in the next subsection for translating automata working on k-expressions to automata working on annotated rank decompositions. We will use some definitions that are introduced in Section 4.5.1.

Let $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ be an annotated rank decomposition that encodes a graph G and has width ℓ. We start with an observation that allows to optimize the k of the expression by one.

Observation 4.10.1. *Let $\vec{xy} \in \vec{E}(T)$. There are at most $2^\ell - 1$ vertices $v \in \mathcal{R}(\vec{xy})$ so that $N_{\mathcal{E}(xy)}(v)$ is nonempty.*

Proof. Let M be the $|\mathcal{R}(\vec{xy})| \times |\mathcal{R}(\vec{yx})|$ matrix describing adjacencies of $\mathcal{E}(xy)$. We have that the rank of M is at most ℓ, so it has a row-basis of size ℓ. All other rows can be written as linear combinations of this row-basis with coefficients 0 and 1, so there are at most $2^\ell - 1$ different nonzero rows. □

Then let $k = 2 \cdot 2^\ell - 1$. We define the k-graph associated with an oriented edge $\vec{xy} \in \vec{E}(T)$ to be the k-graph

$$\mathcal{G}(\vec{xy}) = (G(\vec{xy}), V_1(\vec{xy}), \ldots, V_k(\vec{xy})),$$

so that $G(\vec{xy}) = G[\mathcal{L}(\mathcal{T})[\vec{xy}]]$ and where the sets $V_1(\vec{xy}), \ldots, V_k(\vec{xy})$ are defined as follows. Let $\xi_{\vec{xy}}\colon \mathcal{R}(\vec{xy}) \to [2^\ell]$ be the injective function that maps each $v \in \mathcal{R}(\vec{xy})$ to $\xi_{\vec{xy}}(v) \in [2^\ell]$ so that

- if $N_{\mathcal{E}(xy)}(v) = \emptyset$ then $\xi_{\vec{xy}}(v) = 2^\ell$, and
- otherwise $\xi_{\vec{xy}}(v)$ is the number $i \in [2^\ell - 1]$ so that there are exactly $i-1$ vertices $u \in \mathcal{R}(\vec{xy})$ with $u < v$ and $N_{\mathcal{E}(xy)}(u) \neq \emptyset$.

Let $v \in V(G(\vec{xy}))$. There exists unique $r_v \in \mathcal{R}(\vec{xy})$ so that $N_G(r_v) \cap \mathcal{L}(\mathcal{T})[\vec{yx}] = N_G(v) \cap \mathcal{L}(\mathcal{T})[\vec{yx}]$. We assign v to the set $V_{\xi_{\vec{xy}}(r_v)}$. This concludes the definition of $\mathcal{G}(\vec{xy})$. We observe that $\xi_{\vec{xy}}$ can be computed from $\mathcal{R}(\vec{xy})$ and $\mathcal{E}(xy)$ in time $\mathcal{O}_\ell(1)$.

Then we show that these graphs can be inductively constructed on the rank decomposition by operations in op_k.

Lemma 4.10.2. *Let $\vec{xy} \in \vec{E}(T)$ be a nonleaf oriented edge and $\vec{c_1x}, \vec{c_2x}$ be the children of $\vec{xy}$. The k-graph $\mathcal{G}(\vec{xy})$ can be produced by a sequence of $\mathcal{O}(k^2)$ operations in op_k from the k-graphs $\mathcal{G}(\vec{c_1x})$ and $\mathcal{G}(\vec{c_2x})$. Moreover, this sequence of operations depends only on the transition signature $\tau(\mathcal{T}, \vec{xy})$ and can be computed given it in $\mathcal{O}_\ell(1)$ time.*

Proof. We give the construction of $\mathcal{G}(x\vec{y})$ from $\mathcal{G}(c_1\vec{x})$ and $\mathcal{G}(c_2\vec{x})$. Because $|\mathcal{R}(c_i\vec{x})| \leqslant 2^\ell$, the sets $V_{2^\ell+1}(c_i\vec{x}), \ldots, V_{2\cdot 2^\ell-1}(c_i\vec{x})$ are empty for both $c_i \in \{c_1, c_2\}$. We start by applying the operations $\pi(j, j+2^\ell)$ for all $j \in [2^\ell - 1]$ to the k-graph $\mathcal{G}(c_2\vec{x})$. Let $\mathcal{G}'(c_2\vec{x})$ be the resulting k-graph. Then, let $\mathcal{G}''(x\vec{y}) = \mathcal{G}(c_1\vec{x}) \oplus \mathcal{G}'(c_2\vec{x})$. For each $u \in \mathcal{R}(c_1\vec{x})$ and $v \in \mathcal{R}(c_2\vec{x})$, we know whether $uv \in E(G)$ by inspecting $\mathcal{F}(c_1xc_2)$ and $\mathcal{E}(xc_2)$, and we know that $uv \notin E(G)$ if $\xi_{c_1\vec{x}}(u) = 2^\ell$ or $\xi_{c_2\vec{x}}(v) = 2^\ell$. If $uv \in E(G)$, we apply the operation $\eta(\xi_{c_1\vec{x}}(u), \xi_{c_2\vec{x}}(v) + 2^\ell)$ to $\mathcal{G}''(x\vec{y})$.

It remains to rename the labels of the representatives. Assume $\ell \geqslant 1$ since otherwise there is nothing to do. We construct a function $f : [k] \to [2^\ell]$ so that for each $u \in \mathcal{R}(c_1\vec{x})$ we have $f(\xi_{c_1\vec{x}}(u)) = \xi_{x\vec{y}}(\mathcal{F}(c_1xy)(u)))$; and similarly, for each $v \in \mathcal{R}(c_2\vec{x})$ with $\xi_{c_2\vec{x}}(v) \neq 2^\ell$ we have $f(\xi_{c_2\vec{x}}(v) + 2^\ell) = \xi_{x\vec{y}}(\mathcal{F}(c_2xy)(v)))$. Since $k > 2^\ell$, it is straightforward to produce a sequence of $\mathcal{O}(k)$ operations $\pi(\cdot,\cdot)$ that, in total, remaps each label $i \in [k]$ to the label $f(i)$.

We observe that this sequence of operations depends only on $\tau(\mathcal{T}, x\vec{y})$ and can be computed from it in $\mathcal{O}_\ell(1)$ time. It remains to prove that it correctly produces the k-graph $\mathcal{G}(x\vec{y})$. Let $\mathcal{G}^* = (G^*, V_1^*, \ldots, V_k^*)$ denote the k-graph resulting from the operations. We prove that $\mathcal{G}^*$ and $\mathcal{G}(x\vec{y})$ are equal.

Let us first check that $G^* = G[\mathcal{L}(\mathcal{T})[x\vec{y}]]$. We have $V(G^*) = V(G(x\vec{y}))$ by construction. Let $\mathcal{G}''(x\vec{y}) = (G''(x\vec{y}), V_1'', \ldots, V_k'')$. We have $V_1'' \cup \ldots \cup V_{2^\ell}'' \subseteq \mathcal{L}(\mathcal{T})[c_1\vec{x}]$ and $V_{2^\ell+1}'' \cup \ldots \cup V_{2\cdot 2^\ell-1}'' \subseteq \mathcal{L}(\mathcal{T})[c_2\vec{x}]$. Therefore, our operations did not add edges between the pairs of vertices in $\mathcal{L}(\mathcal{T})[c_1\vec{x}]$, nor between the pairs of vertices in $\mathcal{L}(\mathcal{T})[c_2\vec{x}]$, so we have that $G^*[\mathcal{L}(\mathcal{T})[c_1\vec{x}]] = G(x\vec{y})[\mathcal{L}(\mathcal{T})[c_1\vec{x}]]$ and $G^*[\mathcal{L}(\mathcal{T})[c_2\vec{x}]] = G(x\vec{y})[\mathcal{L}(\mathcal{T})[c_2\vec{x}]]$. It remains to check edges between $\mathcal{L}(\mathcal{T})[c_1\vec{x}]$ and $\mathcal{L}(\mathcal{T})[c_2\vec{x}]$. By our construction we have that edges between $u \in \mathcal{R}(c_1\vec{x})$ and $v \in \mathcal{R}(c_2\vec{x})$ are as claimed. Suppose that $v \in \mathcal{L}(\mathcal{T})[c_1\vec{x}]$ and $r_v \in \mathcal{R}(c_1\vec{x})$ is the node so that $N_G(r_v) \cap \mathcal{L}(\mathcal{T})[x\vec{c}_1] = N_G(v) \cap \mathcal{L}(\mathcal{T})[x\vec{c}_1]$. We have that v and r_v are in the same set V_l'', and therefore $N_{G^*}(v) \cap \mathcal{L}(\mathcal{T})[c_2\vec{x}] = N_{G^*}(r_v) \cap \mathcal{L}(\mathcal{T})[c_2\vec{x}]$. Therefore, because the neighborhood of r_v to $\mathcal{L}(\mathcal{T})[c_2\vec{x}]$ is correct and $\mathcal{L}(\mathcal{T})[c_2\vec{x}] \subseteq \mathcal{L}(\mathcal{T})[x\vec{c}_1]$, we deduce that the neighborhood of v to $\mathcal{L}(\mathcal{T})[c_2\vec{x}]$ is also correct.

Let us then check that $V_j^* = V_j(x\vec{y})$ for all $j \in [k]$. Consider $v \in \mathcal{R}(c_1\vec{x})$. By definitions of annotated rank decompositions we have that $N_G(v) \cap \mathcal{L}(\mathcal{T})[y\vec{x}] = N_G(\mathcal{F}(c_1xy)(v)) \cap \mathcal{L}(\mathcal{T})[y\vec{x}]$, which readily implies that $v \in V_j^*$ if and only if $v \in V_j(x\vec{y})$. Then consider $v \in \mathcal{L}(\mathcal{T})[c_1\vec{x}]$, and again let $r_v \in \mathcal{R}(c_1\vec{x})$ be the node so that $N_G(r_v) \cap \mathcal{L}(\mathcal{T})[x\vec{c}_1] = N_G(v) \cap \mathcal{L}(\mathcal{T})[x\vec{c}_1]$. We have that v and r_v are in the same set V_l'', so they end up in the same set V_j^*. Because $\mathcal{L}(\mathcal{T})[y\vec{x}] \subseteq \mathcal{L}(\mathcal{T})[x\vec{c}_1]$, we have that $N_G(r_v) \cap \mathcal{L}(\mathcal{T})[y\vec{x}] = N_G(v) \cap \mathcal{L}(\mathcal{T})[y\vec{x}]$, so the correctness of v follows from the correctness of r_v. The proof for $v \in \mathcal{L}(\mathcal{T})[c_2\vec{x}]$ is similar. □

Then, with similar arguments we can show that a k-graph representing G can be constructed from $\mathcal{G}(x\vec{y})$ and $\mathcal{G}(y\vec{x})$ for some edge $xy \in E(T)$. We omit the proof as it is similar to the proof of Lemma 4.10.2.

Lemma 4.10.3. *Let $xy \in E(T)$. The k-graph $(G, V(G), \emptyset, \ldots, \emptyset)$ can be produced by a sequence of $\mathcal{O}(k^2)$ operations in op_k from the k-graphs $\mathcal{G}(x\vec{y})$ and $\mathcal{G}(y\vec{x})$. Moreover, this sequence of operations depends only on the edge signature $\sigma(\mathcal{T}, x\vec{y})$.*

Now we are ready to give the algorithm to translate annotated rank decompositions into k-expressions.

Lemma 4.10.4. *There is an algorithm that given an annotated rank decomposition $\mathcal{T}$ of width ℓ that encodes a graph G, in time $\mathcal{O}_\ell(|\mathcal{T}|)$ outputs a $(2^{\ell+1}-1)$-expression that encodes G.*

Proof. Let $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ and $k = (2^{\ell+1} - 1)$, and let us use the definitions introduced in this subsection. We choose an arbitrary edge $ab \in E(T)$. By using Lemma 4.10.2, we compute for each nonleaf oriented edge $x\vec{y} \in \mathsf{pred}_T(\vec{ab}) \cup \mathsf{pred}_T(\vec{ba})$ a rooted tree with $\mathcal{O}(k^2)$ nodes, so that the internal nodes are labeled with operations in op_k and the two leaves are labeled with the two child edges $c_1\vec{x}$ and $c_2\vec{x}$ of $x\vec{y}$, so that it corresponds to a sequence of operations in op_k that turn $\mathcal{G}(c_1\vec{x})$ and $\mathcal{G}(c_2\vec{x})$ into $\mathcal{G}(x\vec{y})$. We also use Lemma 4.10.3 to compute the rooted tree with $\mathcal{O}(k^2)$ nodes, so that the internal nodes are labeled with operations in op_k and the two leaves are labeled with a and b, so that it corresponds to a sequence of operations in op_k that turn $\mathcal{G}(\vec{ab})$ and $\mathcal{G}(\vec{ba})$ into $(G, V(G), \emptyset, \ldots, \emptyset)$. For each leaf edge $\vec{lp} \in \vec{L}(T)$ we compute the k-expression with at most one operation that turns the k-graph $(G[\mathcal{R}(\vec{lp}), V_1, \emptyset, \ldots, \emptyset])$ into $\mathcal{G}(\vec{lp})$. Now, we observe that by gluing these $\mathcal{O}(|\mathcal{T}|)$ trees we computed together, we obtain a k-expression that encodes G. This takes in total $\mathcal{O}_\ell(|\mathcal{T}|)$ time. □

4.10.2 Automata on k-expressions

We then define automata working on k-expressions. Our definitions do not strictly follow any literature as they are geared to our notation and the goal of proving Lemma 4.10.5, but can be seen as equivalent to definitions given by Courcelle and Engelfriet [CE12].

A k-expression automaton is a 6-tuple $\mathcal{A} = (Q, \Gamma, \iota, \chi, \psi, \phi)$ that consists of

- a state set Q,
- a vertex label set Γ,
- an initial mapping ι that maps a single-vertex graph labeled with $\gamma \in \Gamma$ to a state $\iota(\gamma) \in Q$,
- a transition mapping ψ that maps every pair of form (μ, q), where $\mu \in \mathsf{op}_k \setminus \{\oplus\}$ and $q \in Q$ to a state $\psi(\mu, q) \in Q$,
- a transition mapping χ that maps every pair of states $(q_1, q_2) \in Q \times Q$ to a state $\chi(q_1, q_2) \in Q$, and
- a final mapping ϕ that maps each state $q \in Q$ to a state $\phi(q) \in Q$.

The evaluation time of the automaton is the maximum running time to compute the functions ι, ψ, χ, and ϕ given their arguments.

Let $\mathsf{Expr} = (T, V(G), \mu)$ be a k-expression that encodes a graph G and $\alpha\colon V(G) \to \Gamma$ a vertex-labeling of G with Γ. The *run* of $\mathcal{A}$ on the pair (Expr, α) is the unique mapping $\rho\colon V(T) \to Q$ so that

- for each leaf $l \in L(T)$ it holds that $\rho(l) = \iota(\alpha(\mu(t)))$,
- for each node $t \in V(T)$ that has one child c it holds that $\rho(t) = \psi(\mu(t), \rho(c))$, and
- for each node $t \in V(T)$ that has two children c_1, c_2 with $c_1 < c_2$ it holds that $\rho(t) = \chi(\rho(c_1), \rho(c_2))$.

The valuation of $\mathcal{A}$ on (Expr, α) is $\phi(\rho(r))$, where r is the root of T. We say that $\mathcal{A}$ is *expression-oblivious* if its valuation on (Expr, α) depends only on the graph G encoded by Expr and the labeling α. In that case, we call this also the valuation of $\mathcal{A}$ on (G, α). The purpose of the final mapping ϕ in the definition is to be able to make k-expression automata expression-oblivious, for example, if the purpose of $\mathcal{A}$ is to decide whether G satisfies some graph property, then the image of ϕ could be just $\{\bot, \top\}$, while Q could be much larger in order to represent intermediate computations.

We are now ready to prove that k-expression automata can be translated into rank decomposition automata. This is not surprising since the construction of $(2^{\ell+1}-1)$-expression from a rank decomposition of width ℓ in Lemma 4.10.4 works in a local manner. The proof uses definitions of rank decomposition automata from Section 4.5.1.

Lemma 4.10.5. *Let $\ell \in \mathbb{N}$ and $k = 2^{\ell+1} - 1$. Given an expression-oblivious k-expression automaton $\mathcal{A}_{\mathsf{ex}} = (Q, \Gamma, \iota, \chi, \psi, \phi)$ with evaluation time β, it is possible to construct a rank decomposition automaton $\mathcal{A}_{\mathsf{rd}} = (Q, \Gamma, \iota', \delta, \varepsilon)$ of width ℓ and evaluation time $\mathcal{O}_\ell(\beta)$, so that if $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ is an annotated rank decomposition that encodes a graph G and has width at most ℓ, $\alpha\colon V(G) \to \Gamma$ is a vertex-labeling of G with Γ, and $a, b \in V(T)$ is a pair of adjacent nodes in T, then the valuation of $\mathcal{A}_{\mathsf{rd}}$ on $(\mathcal{T}, a, b, \alpha)$ is the same as the valuation of $\mathcal{A}_{\mathsf{ex}}$ on (G, α).*

Proof. We use the definitions of $\mathcal{G}(\vec{xy})$ and $\xi_{\vec{xy}}$ introduced in Section 4.10.1. By Lemma 4.10.2 we can associate with each $\vec{xy} \in \vec{E}(T)$ a k-expression $\mathsf{Expr}(\vec{xy}) = (T^{\mathsf{ex}}(\vec{xy}), \mathcal{L}(\mathcal{T})[\vec{xy}], \mu(\vec{xy}))$ so that the root of $T^{\mathsf{ex}}(\vec{xy})$ encodes $\mathcal{G}(\vec{xy})$, and if $\vec{xy}$ is nonleaf then $\mathsf{Expr}(\vec{xy})$ is constructed by combining $\mathsf{Expr}(\vec{c_1x})$ and $\mathsf{Expr}(\vec{c_2x})$ by $\mathcal{O}_\ell(1)$ operations in op_k that depend only on $\tau(\mathcal{T}, \vec{xy})$. In particular, if T_{pref} is the prefix of $T^{\mathsf{ex}}(\vec{xy})$ so that the connected components of $T^{\mathsf{ex}}(\vec{xy}) - T_{\mathrm{pref}}$ are $T^{\mathsf{ex}}(\vec{c_1x})$ and $T^{\mathsf{ex}}(\vec{c_2x})$, then the pair $\mathsf{Expr}(\tau(\mathcal{T}, \vec{xy})) = (T^{\mathsf{ex}}(\vec{xy})[T_{\mathrm{pref}} \cup \mathsf{App}(T_{\mathrm{pref}})], \mu(\vec{xy})|_{T_{\mathrm{pref}}})$ depends only on $\tau(\mathcal{T}, \vec{xy})$. The tree $T^{\mathsf{ex}}(\vec{xy})[T_{\mathrm{pref}} \cup \mathsf{App}(T_{\mathrm{pref}})]$ has exactly two leaves that correspond to the roots of $\mathsf{Expr}(\vec{c_1x})$ and $\mathsf{Expr}(\vec{c_2x})$, and we let names of these leaves be l_1 and l_2 so that l_i corresponds to c_i (note that $\tau(\mathcal{T}, \vec{xy})$ includes the subtree $T[\{x, y, c_1, c_2\}]$ so this is allowed). If $\vec{xy}$ is a leaf edge then $\mathcal{T}^{\mathsf{ex}}(\vec{xy})$ is the k-expression consisting of at most two nodes that encodes $\mathcal{G}(\vec{xy})$.

Then we define the automaton $\mathcal{A}_{\mathsf{rd}} = (Q, \Gamma, \iota', \delta, \varepsilon)$. Like indicated by the notation, the sets Q and Γ are the same as for the automaton $\mathcal{A}_{\mathsf{ex}} = (Q, \Gamma, \iota, \chi, \psi, \phi)$. The function ι' is defined as follows: Let σ be an edge signature $\sigma = (\mathcal{R}^a_\sigma, \mathcal{R}^b_\sigma, \mathcal{E}_\sigma)$ and γ a function $\gamma\colon \mathcal{R}^a_\sigma \to \Gamma$. If $\mathcal{R}^a_\sigma$ is a set consisting of a single vertex v, we set $\iota'(\sigma, \gamma) = \iota(\gamma(v))$. Otherwise, we set $\iota'(\sigma, \gamma)$ to be an arbitrary state in Q. Note that $\mathcal{A}_{\mathsf{rd}}$ is required to work only on annotated rank decompositions that encode graphs, for which the latter case never happens.

The mapping $\delta(\tau, q_1, q_2)$, where τ is a transition signature and $q_1, q_2 \in Q$ is defined as follows. We take the pair $\mathsf{Expr}(\tau) = (T^*, \mu^*)$ defined earlier in the course of the proof. Let $L(T^*) = \{l_1, l_2\}$. Then we take the run of $\mathcal{A}_{\mathsf{ex}}$ on (T^*, μ^*), defined as a function $\rho\colon V(T^*) \to Q$ so that for the two leaves l_1, l_2 we have $\rho(l_1) = q_1$ and $\rho(l_2) = q_2$, and for other nodes the run is defined as per the usual definition of a run of $\mathcal{A}_{\mathsf{ex}}$. Then, we set $\delta(\tau, q_1, q_2) = \rho(r)$, where r is the root of T^*. Before defining ε we can observe that the following claim follows from our construction.

Observation 4.10.6. *Let $\vec{xy} \in \vec{E}(T)$ and $\rho_{\mathsf{rd}} \colon \mathsf{pred}_T(\vec{xy}) \to Q$ be the run of $\mathcal{A}_{\mathsf{rd}}$ on $(\mathcal{T}, \vec{xy}, \alpha)$. Let also $\rho_{\mathsf{ex}} \colon V(T^{\mathsf{ex}}(\vec{xy})) \to Q$ be the run of $\mathcal{A}_{\mathsf{ex}}$ on $\mathsf{Expr}(\vec{xy})$. Then $\rho_{\mathsf{rd}}(\vec{xy}) = \rho_{\mathsf{ex}}(r(\vec{xy}))$, where $r(\vec{xy})$ is the root of $T^{\mathsf{ex}}(\vec{xy})$.*

Next we define ε. By Lemma 4.10.2, the k-graph $\mathcal{G} = (G, V(G), \emptyset, \ldots, \emptyset)$ can be constructed from the k-graphs $\mathcal{G}(\vec{xy})$ and $\mathcal{G}(\vec{yx})$ by $\mathcal{O}_\ell(1)$ applications of operations in op_k that depend only on the edge signature $\sigma(\mathcal{T}, \vec{xy})$. Therefore, we can similarly define a k-expression $\mathsf{Expr}(x, y) = (T^{\mathsf{ex}}(\vec{xy}), \mathcal{L}(\mathcal{T})[\vec{xy}], \mu(\vec{xy}))$ that encodes $\mathcal{G}$ and is constructed by combining $\mathsf{Expr}(\vec{xy})$ and $\mathsf{Expr}(\vec{yx})$ by $\mathcal{O}_\ell(1)$ operations in op_k that depend only on $\sigma(\mathcal{T}, \vec{xy})$. We can also define a pair $\mathsf{Expr}(\sigma(\mathcal{T}, \vec{xy}))$ to describe how exactly these k-expressions should be combined.

Now, $\varepsilon(\sigma, q_1, q_2)$ can be constructed from $\mathsf{Expr}(\sigma)$ similarly as δ was constructed from $\mathsf{Expr}(\tau)$ and finally applying the mapping ϕ, so that the valuation of $\mathcal{A}_{\mathsf{rd}}$ on $(\mathcal{T}, x, y, \alpha)$ is the same as the valuation of $\mathcal{A}_{\mathsf{ex}}$ on $(\mathsf{Expr}(x, y), \alpha)$. Now because $\mathcal{A}_{\mathsf{ex}}$ is expression-oblivious, the valuation of $\mathcal{A}_{\mathsf{ex}}$ on $(\mathsf{Expr}(x, y), \alpha)$ is the valuation of $\mathcal{A}_{\mathsf{ex}}$ on (G, α), which concludes the correctness of the construction. In the constructions of the functions δ and ε we apply the functions χ,ψ, and ϕ $\mathcal{O}_\ell(1)$ times, so the evaluation time of $\mathcal{A}_{\mathsf{rd}}$ is $\mathcal{O}_\ell(\beta)$. □

We note that the properties of $\mathcal{A}_{\mathsf{rd}}$ asserted in the statement of Lemma 4.10.5 imply that it is decomposition-oblivious.

4.10.3 CMSO_1

We use definitions of CMSO_1 logic given in Section 4.5.2. The following theorem was given in [CMR00] (see also [CE12, Section 6]).

Theorem 4.10.7 ([CMR00])**.** *There is an algorithm that given a CMSO_1 sentence φ with p free set variables and $k \in \mathbb{N}$, in time $\mathcal{O}_{\varphi,k}(1)$ constructs an decomposition-oblivious k-expression automaton $\mathcal{A} = (Q, \Gamma, \iota, \chi, \psi, \phi)$ so that $\Gamma = 2^{[p]}$, the valuation of $\mathcal{A}$ on (G, α) is $\top \in Q$ if and only if $(G, \alpha) \models \varphi$, the number of states is $|Q| \leqslant \mathcal{O}_{\varphi,k}(1)$, and the evaluation time is $\mathcal{O}_{\varphi,k}(1)$.*

By combining Lemma 4.10.5 and Theorem 4.10.7, we immediately obtain the following.

Lemma 4.5.2. *There is an algorithm that given a CMSO_1 sentence φ with p free set variables and $\ell \in \mathbb{N}$, in time $\mathcal{O}_{\varphi,\ell}(1)$ constructs a decomposition-oblivious rank decomposition automaton $\mathcal{A} = (Q, \Gamma, \iota, \delta, \varepsilon)$ of width ℓ so that $\Gamma = 2^{[p]}$, the valuation of $\mathcal{A}$ on (G, α) is $\top \in Q$ if and only if $(G, \alpha) \models \varphi$, the number of states is $|Q| \leqslant \mathcal{O}_{\varphi,\ell}(1)$, and the evaluation time is $\mathcal{O}_{\varphi,\ell}(1)$.*

Then we prove Lemma 4.5.3 by using Lemma 4.5.2.

Lemma 4.5.3. *There is an algorithm that given a $\mathsf{LinCMSO}_1$ sentence φ with p free set variables and $\ell \in \mathbb{N}$, in time $\mathcal{O}_{\varphi,\ell}(1)$ constructs a decomposition-oblivious rank decomposition automaton $\mathcal{A} = (Q, \Gamma, \iota, \delta, \varepsilon)$ of width ℓ so that $\Gamma = 2^{[p]}$, the valuation of $\mathcal{A}$ on (G, α) is equal to the value of φ on (G, α), and the evaluation time is $\mathcal{O}_{\varphi,\ell}(1)$.*

Proof. Denote $\varphi = (\phi, f)$, where ϕ is a CMSO_1 sentence with $p + q$ free variables, where p is the number of free variables of φ. Let $f(x_1, \ldots, x_q) = c_0 + c_1 x_1 + \ldots + c_q x_q$. We first use Lemma 4.5.2 to turn ϕ into a rank decomposition automaton $\mathcal{A}' = (Q', \Gamma', \iota', \delta', \varepsilon')$ of width ℓ.

Let $\mathcal{T} = (T, V(G), \mathcal{R}, \mathcal{E}, \mathcal{F})$ be an annotated rank decomposition that encodes a graph G, $\Gamma = 2^{[p]}$, and $\alpha \colon V(G) \to \Gamma$ a vertex-labeling of G. Then, for a set $X \subseteq V(G)$ and a vertex labeling $\alpha' \colon X \to 2^{[p+1,p+q]}$, we define $\mathsf{val}(X, \alpha') = f(|X_1|, \ldots, |X_q|)$ where $X_i = \{v \in X \mid i + p \in \alpha'(v)\}$. We also denote by $\alpha|_X \cup \alpha'$ the function $\alpha|_X \cup \alpha' \colon X \to 2^{[p+q]}$ with $(\alpha|_X \cup \alpha')(v) = \alpha|_X(v) \cup \alpha'(v)$ for all $v \in X$. Then for every pair $(\vec{xy}, s)$ with $\vec{xy} \in \vec{E}(T)$ and $s \in Q'$, we define $\mathsf{maxval}(\vec{xy}, s)$ to be the maximum value of $\mathsf{val}(\mathcal{L}(\mathcal{T})[\vec{xy}], \alpha')$ over all functions $\alpha' \colon \mathcal{L}(\mathcal{T})[\vec{xy}] \to 2^{[p+1,p+q]}$ so that the valuation of $\mathcal{A}'$ on $(\mathcal{T}, \vec{xy}, \alpha \cup \alpha')$ is s, or $-\infty$ if no such α' exists.

Now, the state set of $\mathcal{A}$ is the set of all functions $g \colon Q' \to \mathbb{Z} \cup \{-\infty\}$, and we can define the transitions of $\mathcal{A}$ so that the valuation of $\mathcal{A}$ on $(\mathcal{T}, \vec{xy}, \alpha)$ is the function $g_{\vec{xy}}$ that maps each $s \in Q'$ to $\mathsf{maxval}(\vec{xy}, s)$. In particular, for nonleaf edges $\vec{xy}$ with child edges $c_1\vec{x}$ and $c_2\vec{x}$ this can be done by setting for each $s \in Q'$ the value $g_{\vec{xy}}(s)$ to be the maximum of $g_{c_1\vec{x}}(s_1) + g_{c_2\vec{x}}(s_2) - c_0$ so that $\delta(\tau(\mathcal{T}, \vec{xy}), s_1, s_2) = s$. The construction of the initial mapping ι is straightforward. We observe that we can construct the final mapping similarly, so that valuation of $\mathcal{A}$ on (G, α) is equal to the maximum value of $\mathsf{val}(V(G), \alpha')$ over all functions $\alpha' \colon V(G) \to 2^{[p+q]}$ so that the valuation of $\mathcal{A}'$ on $(G, \alpha \cup \alpha')$ is $\top$, and if no such α' exists, the valuation is $\bot$. This gives evaluation time $\mathcal{O}(|Q'|^2 \cdot \beta)$, where β is the evaluation time of $\mathcal{A}'$, resulting in $\mathcal{O}_{\ell,\varphi}(1)$ evaluation time. □

Let us then also prove Lemma 4.3.6 here.

Lemma 4.3.6. *There is an algorithm that given an annotated rank decomposition $\mathcal{T}$ of width ℓ that encodes a partitioned graph $(G,\mathcal{C})$, a graph H, and a function $\gamma\colon V(G)\to 2^{V(H)}$, in time $\mathcal{O}_{\ell,H}(|\mathcal{T}|)$ either returns a witness of H as a labeled induced subgraph of (G,γ) or returns that (G,γ) does not contain H as a labeled induced subgraph.*

Proof. We first turn $\mathcal{T}$ into an annotated rank decomposition $\mathcal{T}'=(T',V(G),\mathcal{R}',\mathcal{E}',\mathcal{F}')$ that encodes the graph G (instead of the partitioned graph $(G,\mathcal{C})$). This can be done in $\mathcal{O}_\ell(|\mathcal{T}|)$ time by adding a subtree of size $\mathcal{O}_\ell(1)$ below each leaf of $\mathcal{T}$.

Let $|V(H)|=p$ and let us index the vertices of H by $u_1,\ldots,u_p$. We write a CMSO_1 sentence φ of length $|\varphi|\leqslant\mathcal{O}_H(1)$ with $2p$ free variables so that $(G,X_1,\ldots,X_p,Y_1,\ldots,Y_p)\models\varphi$ if and only if $|X_i|=1$ and $X_i\subseteq Y_i$ for all $i\in[p]$, and $G[X_1\cup\ldots\cup X_p]$ is isomorphic to H with an isomorphism that maps the single vertex $v_i\in X_i$ to u_i. We use Lemma 4.5.2 to construct a rank decomposition automaton $\mathcal{A}=(Q,\Gamma,\iota,\delta,\varepsilon)$ so that for all adjacent nodes $x,y\in V(T')$, the valuation of $\mathcal{A}$ on $(\mathcal{T}',x,y,\alpha)$ is $\top$ if and only if $\alpha\colon V(G)\to 2^{[2p]}$ is a vertex-labeling corresponding to $X_1,\ldots,X_p,Y_1,\ldots,Y_p$ so that $(G,X_1,\ldots,X_p,Y_1,\ldots,Y_p)\models\varphi$.

We construct labeling $\alpha\colon V(G)\to 2^{[2p]}$ so that $\alpha(v)\cap[p]=\emptyset$ and $p+i\in\alpha(v)$ if and only if $u_i\in\gamma(v)$. Then, if f is a function $f\colon[p]\to V(G)\cup\{\bot\}$, we denote by $\alpha+f$ the function $(\alpha+f)\colon V(G)\to 2^{[2p]}$ so that $(\alpha+f)(v)=\alpha(v)\cup\{i\mid f(i)=v\}$. Now, for each oriented edge $\vec{xy}$ of T' denote by $g_{\vec{xy}}$ the function that maps each $q\in Q$ to a function $g_{\vec{xy}}(q)\colon[p]\to\mathcal{L}(\mathcal{T}')[\vec{xy}]\cup\{\bot\}$ so that the valuation of $\mathcal{A}$ on $(\mathcal{T}',\vec{xy},\alpha+g_{\vec{xy}}(q))$ is q, or to $\bot$ if no such function exists. Now we can construct an auxiliary automaton $\mathcal{A}'$ that computes $g_{\vec{xy}}$ for each oriented edge $\vec{xy}$ of T' directed towards an arbitrarily chosen root, and finally from that construct a function $f\colon[p]\to V(G)\cup\{\bot\}$ so that the valuation of $\mathcal{A}$ on $(\mathcal{T}',x,y,(\alpha+f))$ is $\top$, or find that no such f exists. By construction, such f corresponds to a witness of H as a labeled induced subgraph of (G,γ). □

4.11 Conclusions

We gave a data structure for maintaining bounded-width rank decompositions of dynamic graphs of bounded rankwidth in subpolynomial time per update. We also used this data structure to give an almost-linear time parameterized algorithm for computing an optimum-width rank decomposition of a given graph. Along the way, we proved several auxiliary structural and algorithmic results for rankwidth. An important conceptual contribution of our result appears to be the definition of annotated rank decompositions, together with efficient algorithms for manipulating them and for translating dynamic programming from other representations of rank decompositions to annotated rank decompositions. We then discuss future research directions and make some additional remarks about our results.

As for dynamic treewidth, the obvious interesting open problem is to improve the dynamic algorithm of Theorem 1.3.4 to work in $\mathcal{O}_k(\log^{\mathcal{O}(1)}n)$ time per update, instead of the current $2^{\mathcal{O}_k(\sqrt{\log n\log\log n})}$ time. This would also improve the algorithm of Theorem 1.3.6 to $\mathcal{O}_k(n\log^{\mathcal{O}(1)}n)+\mathcal{O}(m)$ time. The natural path to solve it would be to first improve the dynamic treewidth algorithm presented in Chapter 3, and then generalize the result to rankwidth. However, we note that the tools developed in Section 4.4 appear to give a cleaner and more elegant framework for dynamic rankwidth than the framework for dynamic treewidth from Chapter 3 is, so it could make sense to approach dynamic treewidth via dynamic rankwidth, or perhaps via dynamic branchwidth.

In Theorem 1.3.5 we gave a framework for applying edge updates defined by CMSO_1 sentences. In this framework, the time required to apply the update is at least linear in the number of vertices incident to the edges updated. It would be interesting to explore whether this limitation could be lifted for some types of edge updates. In particular, would there exist a framework for updating many edges at once, where the update time could be sublinear in the number of vertices incident to the edges updated?

Rankwidth of graphs is related to branchwidth of matroids, so it would be interesting to explore whether our techniques could be extended into that setting. We note that by the connection proved by Oum [Oum05], all rankwidth algorithms directly apply to branchwidth of binary matroids when the binary matroid is represented by its fundamental graph, so Theorem 1.3.6 gives an improvement in this setting. However, our techniques do not seem to directly apply to the more interesting setting of linear matroids represented by matrices.

In Theorem 1.3.5 we support operations that take some partial vertex-labeling as an input. We note that Theorems 1.3.4 and 1.3.5 can be easily extended to the setting where instead of a graph, we maintain

a vertex-labeled graph with a bounded number of labels that can be accessed by the $\mathsf{LinCMSO}_1$ formulas. This extension can be done simply by gadgeteering: We can add some number of degree-1 neighbors to each vertex to encode the label of that vertex. These gadgeteering techniques also appear applicable for extending our results to the setting of rankwidth/cliquewidth of more general binary relational structures, with an approximation factor depending on the exact definition of rankwidth in that setting.

Lastly, we remark that our dynamic algorithm works in space $\mathcal{O}_k(n)$, and the algorithm of Theorem 1.3.6 in space $\mathcal{O}_k(n) + \mathcal{O}(m)$. In particular, the dynamic algorithm could be interesting from the viewpoint of models of computation with limited space, as its space complexity can be sublinear in the total size $n + m$ of the graph. In a similar vein, we could consider the complexity of computing rankwidth of dense graphs (in which the number of edges is quadratic in the number of vertices) in the Word RAM model. Since every n-vertex graph can be encoded in $\mathcal{O}(\frac{n^2}{\log n})$ RAM words, it is potentially viable to compute the rankwidth of an n-vertex graph in $\mathcal{O}(\frac{n^2}{\log n})$ time, improving upon [FK22] and Theorem 1.3.6.

Chapter 5

Dynamic Baker's technique

Baker's technique, also known as *shifting* or *layering* and proposed by Baker in 1994 [Bak94], is the most fundamental technique for designing *efficient polynomial-time approximation schemes* (EPTASes) – approximation schemes with running time of the form $f(\varepsilon) \cdot n^{\mathcal{O}(1)}$ – for problems on planar graphs, or more generally in topologically restricted graph classes. In terms of restrictions on the input graph, the basic approach works as long as the considered class of graphs $\mathcal{C}$ has *bounded local treewidth*: The treewidth of any connected $G \in \mathcal{C}$ is bounded by a function of the radius of G. This includes planar graphs (with the function being 3 times the radius) and, as proved by Eppstein [Epp00], also all *apex-minor-free* graph classes: classes that exclude a fixed *apex graph* – a graph that can be made planar by removing one vertex – as a minor. However, further generalizations of the technique apply also in the setting of H-minor-free graphs for any fixed H [Gro03], and even beyond [Dvo18, Dvo20, Dvo22]. In terms of versatility, the range of applicability of Baker's technique is surprisingly wide, and can be even captured by meta-theorems concerning optimization problems expressible in first-order logic [DGKS06, Dvo22]. Importantly, the spectrum of applicability includes maximization problems of packing nature such as Maximum Weight Independent Set – in a vertex-weighted graph G, find a set I of maximum possible weight consisting of pairwise nonadjacent vertices – and minimization problems of covering nature such as Minimum Weight Dominating Set – find a set of vertices D of minimum possible weight such that every vertex outside of D has a neighbor in D.

During the last 30 years, the basic principle behind Baker's technique has inspired countless important developments in the area of approximation schemes, see e.g. [CPP19, EKM14, FKS19, KPR93] besides the works mentioned above. It has also found multiple important applications in parameterized algorithms, see e.g. the discussion in [CFK+15, Section 7.7.3].

Our contribution. In this chapter we investigate the following question: To what extent shifting – or more precisely – Baker's technique – can be applied in the context of dynamic algorithms? For the sake of focus, we restrict attention to the aforementioned two basic problems: Maximum Weight Independent Set and Minimum Weight Dominating Set.

Consider a fully dynamic graph data structure that maintains a graph G with nonnegative weights on vertices under the following updates:

- $\mathsf{AddEdge}(u, v)$, $\mathsf{RemoveEdge}(u, v)$: Adds or removes an edge between vertices u, v; and
- $\mathsf{UpdateWeight}(u, \alpha)$: Changes the weight of a vertex u to $\alpha \in \mathbb{R}_{\geqslant 0}$.

Recall from the Introduction that a data structure for dynamic graphs is *$\mathcal{C}$-restricted*, for a graph class $\mathcal{C}$, if it works under the promise that at all times, the graph stored in the data structure belongs to $\mathcal{C}$. With these definitions in place, we can recall our main result from the Introduction:

Theorem 1.3.7 ([KNPS24]). *Let $\mathcal{C}$ be a fixed apex-minor-free class of graphs and let $\varepsilon > 0$. Then there exists a $\mathcal{C}$-restricted fully dynamic graph data structure that in addition to maintaining a graph $G \in \mathcal{C}$, supports the following queries:*

- $\mathsf{QueryMWIS}()$*: Outputs a nonnegative real p satisfying $(1-\varepsilon)\mathsf{OPT}_{\mathsf{IS}} \leqslant p \leqslant \mathsf{OPT}_{\mathsf{IS}}$, where $\mathsf{OPT}_{\mathsf{IS}}$ is the maximum weight of an independent set in G; and*
- $\mathsf{QueryMWDS}()$*: Outputs a nonnegative real p satisfying $\mathsf{OPT}_{\mathsf{DS}} \leqslant p \leqslant (1+\varepsilon)\mathsf{OPT}_{\mathsf{DS}}$, where $\mathsf{OPT}_{\mathsf{DS}}$ is the minimum weight of a dominating set in G. This query is supported only under the additional assumption that at all times, the maximum degree of G is bounded by a constant Δ.*

The initialization time on a given n-vertex graph $G \in \mathcal{C}$ is $f(\varepsilon) \cdot n^{1+o(1)}$, and each update takes amortized time $f(\varepsilon) \cdot n^{o(1)}$, where $f(\varepsilon)$ is doubly-exponential in $\mathcal{O}(1/\varepsilon^2)$. Each query takes $\mathcal{O}(1)$ time.

Note that in general graphs, Maximum Weight Independent Set and Minimum Weight Dominating Set do not admit EPTASes in the static setting if $\mathsf{P} \neq \mathsf{NP}$, even with a restriction on the maximum degree of a vertex of a graph [CC08, Tre01]. Hence a structural restriction of the class $\mathcal{C}$ in Theorem 1.3.7 is necessary.

We also remark that for the *unweighted* Maximum Independent Set problem, where each vertex carries a unit weight, there is a simple data structure with amortized update time $2^{\mathcal{O}(1/\varepsilon)}$ working as follows. Once every $\varepsilon n/8$ updates recompute a $(1-\varepsilon/2)$-approximate solution from scratch using Baker's technique. Between the recomputations, whenever an edge is added to the graph, remove any of its endpoints from the solution provided both were included. This works because by the Four Color Theorem, the optimum solution has always size at least $n/4$, and within $\varepsilon n/8$ updates the value of the optimum may change by at most $\varepsilon n/8$. This is why we focus on the weighted variant of the problem, to make it nontrivial.

Structure of the chapter. In Section 5.1, we present a technical overview of the techniques used in the proof of Theorem 1.3.7. Next, in Section 5.2, we give preliminary results used in the proof of our result. Then the proof of Theorem 1.3.7 follows. We present data structures for the considered problems in separate sections: We show a dynamic data structure for Maximum Weight Independent Set in Section 5.3, and we follow with a data structure for Minimum Weight Dominating Set in Section 5.4. We conclude and discuss open problems in Section 5.5.

5.1 Overview

The main idea behind the proof of Theorem 1.3.7 is to maintain a tree of data structures, where each data structure is responsible for some minor H of G and has $\mathcal{O}(1/\varepsilon)$ child data structures, responsible for the $\mathcal{O}(1/\varepsilon)$ choices of offsets in the Baker scheme applied to H. While we eventually create a tree of data structures with $L = \Theta\left(\varepsilon \cdot \frac{\log\log n}{\log\log\log n}\right)$ levels, let us explain the construction for $L = 2$ levels; this corresponds to achieving amortized update time $f(\varepsilon) \cdot \widetilde{\mathcal{O}}(\sqrt{n})$. Also, let us focus on the Maximum Weight Independent Set problem.

Let $k := \lceil 1/\varepsilon \rceil$. There will be a parent data structure $\mathbb{D}^{\text{main}}$ responsible for the whole graph G. Upon the initialization of $\mathbb{D}^{\text{main}}$, we apply Baker's scheme to G: We compute a partition $V_0, \ldots, V_{k-1}$ of the vertex set of G so that V_i comprises vertices whose distance from some fixed vertex s is congruent to i modulo k. (If G is disconnected, every connected component has its own source vertex s.) Standard analysis of Baker's technique yields that the treewidth of the graph $G_i := G - V_i$ is at most $3k$, so we compute a tree decomposition $\mathcal{T}_i$ of G_i of width $\mathcal{O}(k)$ and depth $\mathcal{O}(\log n)$ using classic results [BH98]. By dynamic programming on $\mathcal{T}_i$, we can understand $G - V_i$ completely. In particular, we may compute the optimum weight of an independent set in G_i, so that the maximum among the computed values is a $(1-\varepsilon)$-approximation of the maximum weight of an independent set in G.

At this point every graph G_i has treewidth at most $3k$, but this may quickly cease to be the case once some updates arrive. Therefore, for every $i \in \{0, 1, \ldots, k-1\}$ we create a child data structure $\mathbb{D}_i$ that is responsible for handling the graph G_i. In the data structure $\mathbb{D}_i$, vertices of G_i are partitioned into an initially empty *stash* Z, which contains all vertices involved in any updates made so far, and the remaining vertices. We maintain the following invariant ($\bigstar$): Every connected component of $G_i - Z$ has neighborhood of size $\mathcal{O}(k)$. Data structure $\mathbb{D}_i$ maintains the *compressed graph* H_i, which is a minor of G_i obtained as follows: (i) contract every connected component of $G_i - Z$ to a single vertex, and (ii) identify all vertices corresponding to contracted components with the same neighborhood into single vertices. Since Z is initially empty, H_i is initially a single-vertex graph. When an update affects G_i, say an insertion of an edge uv, we add to Z both u and v together with the $\mathcal{O}(k \log n)$ vertices contained in all the ancestor bags[18] of the top-most bags of $\mathcal{T}_i$ that contain u and v; this way we preserve invariant ($\bigstar$). Every addition of a vertex to Z results in "uncompressing" a part of H_i, which can be done efficiently and adds only $\mathcal{O}(k)$ new vertices to H_i.

The data structure $\mathbb{D}_i$ maintains also an approximation of the maximum weight of an independent set in G_i. This is done by maintaining an approximate optimum for an auxiliary maximization problem

[18] The actual presentation differs here in that we work with the notion of an *elimination forest* instead of a tree decomposition. However, this would be the tree decomposition equivalent of this step.

on the compressed graph H_i, which we find convenient to phrase in the language of valued 2CSPs, and which can be considered a "contracted" variant of Maximum Weight Independent Set. More precisely, every vertex of H_i that is also a vertex of G_i has a domain consisting of two values – taken or not taken – while every vertex that resulted from a contraction of some connected components of $G_i - Z$ has a larger domain, corresponding to possible interactions between those components and the rest of the graph. The revenues provided by the contracted vertices for different elements of their domains reflect the maximum weights of independent sets that can be achieved within contracted connected components for different interactions with the rest of the graph. And these maximum weights can be read from the dynamic programming tables in $\mathcal{T}_i$ computed upon the initialization of $\mathbb{D}^{\text{main}}$, because vertices of $G_i - Z$ have not yet been touched by updates. The approximate optimum to the constructed auxiliary instance of 2CSP is computed *from scratch* upon every update in H_i, in time $f(\varepsilon) \cdot |V(H_i)|$ using Baker's technique in H_i.

Thus, every update to G is relayed by $\mathbb{D}^{\text{main}}$ to k data structures $\mathbb{D}_i$, and within each $\mathbb{D}_i$ we apply $k^{\mathcal{O}(1)} \log n$ updates to H_i, each consisting of running Baker's scheme in time $f(\varepsilon) \cdot |V(H_i)|$. The crucial idea is to *reset* the data structure $\mathbb{D}^{\text{main}}$ every $\sqrt{n}$ updates, by reconstructing it from scratch in time $f(\varepsilon) \cdot n$, so that graphs H_i never grow too large. After every reconstruction of $\mathbb{D}^{\text{main}}$, all graphs H_i consist of single vertices, so throughout the next $\sqrt{n}$ updates they can grow to size at most $k^{\mathcal{O}(1)}\sqrt{n}\log n$, because every H_i grows by at most $k^{\mathcal{O}(1)} \log n$ vertices at each update. This means that running Baker's scheme upon every update to any H_i will take worst-case time $f(\varepsilon) \cdot \widetilde{\mathcal{O}}(\sqrt{n})$, while the amortized time complexity of resetting the data structure $\mathbb{D}^{\text{main}}$ is $f(\varepsilon) \cdot \widetilde{\mathcal{O}}(\sqrt{n})$ as well.

Our data structure for the Maximum Weight Independent Set problem implements the natural generalization of the idea presented above to more than two levels. Specifically, we use $L = \Theta\left(\varepsilon \cdot \frac{\log\log n}{\log\log\log n}\right)$ levels, where every data structure $\mathbb{D}$ at level i gets reset every $n^{1-\frac{i}{L}}$ updates relayed to $\mathbb{D}$. One technical detail that requires attention are the domains in the auxiliary 2CSP instances: Every compression step increases the maximum domain size from, say, Δ to $\Delta^{\mathcal{O}(k)}$, so one needs to carefully choose the number of layers L so that it is super-constant in n, but small enough so that the domain sizes are kept subpolynomial in n.

The data structure for the Minimum Weight Dominating Set problem is similar in spirit. As a matter of fact, the standard way of applying Baker's technique to this problem, by allowing double-paying in an ε-fraction of the layers, seems difficult to translate to the dynamic setting. Instead, we design a different way of applying Baker's technique to Minimum Weight Dominating Set, which interestingly relies on *under-approximating* the minimum weight of a dominating set, rather than over-approximating. There are several technical issues that arise when applying the approach presented above in the setting of dominating sets: In particular when a vertex u is touched by an update within any constituent data structure $\mathbb{D}$, we need to update the suitable information about both u and the neighbors of u in the graph. This is why in the result for Minimum Weight Dominating Set we resort to the regime of graphs with bounded maximum degree.

Finally, let us remark that in the approach sketched above, the key properties kept by each of the constituent data structures are the invariant (★) and the invariant that the removal of the stash breaks the graph into connected components of treewidth $\mathcal{O}(k)$. This exactly means that each of those components will be an $\mathcal{O}(k)$-*protrusion*. Protrusion-based arguments are by now a standard methodology in the design of parameterized and kernelization algorithms on planar and topologically-restricted graphs, see e.g. [BFL+16, FLST20] or [FLSZ19, Chapter 15]. In this chapter we show how these ideas can be helpful also in the setting of dynamic approximation algorithms.

5.2 Additional preliminaries

In this section we will list some additional preliminary definitions and results that will be useful in the proof of Theorem 1.3.7.

Multigraphs. In the case of Dominating Set and its generalizations, it will be convenient to work with multigraphs $G = (V, E)$. We assume that in a multigraph, there may be multiple edges connecting the same pair of vertices (also called *parallel edges*), but there are no self-loops. We distinguish different edges connecting the same pair of vertices – for example, we can assume that the edges of the graph have pairwise different labels.

For a vertex $v \in V(G)$, let $\delta(v)$ denote the set of edges with one endpoint in v. We define the degree of v as $\deg(v) = |\delta(v)|$. For two disjoint sets of vertices $A, B \subseteq V(G)$, we use the notation $E(A, B)$ to denote the set of edges with one endpoint in A and the other in B. Given a set of vertices S, to *collapse*

S in $V(G)$ is to identify all vertices of S to a single vertex v_S, remove all edges with both endpoints in S, and for all other edges with an endpoint in S, reroute the endpoint to v_S. Note that this process might produce a multigraph from a simple graph.

Apex graphs. We will use the result of Demaine and Hajiaghayi [DH04] that apex-minor-free classes of graphs have *linearly locally bounded treewidth*, which improved upon the earlier work of Eppstein [Epp00].

Theorem 5.2.1 ([DH04]). *For every apex-minor-free class $\mathcal{C}$ of graphs there exists a constant $\kappa_{\mathcal{C}} > 0$ such that for every integer $r \geqslant 1$ and graph $G \in \mathcal{C}$ of radius at most r, we have $\mathrm{tw}(G) \leqslant \kappa_{\mathcal{C}} \cdot r$.*

Apex-minor-free classes of graphs (and more generally, minor-free classes) have a bounded ratio of the number of edges to the number of vertices [Kos84]; that is, for any apex-minor-free class of graphs $\mathcal{C}$, there exists a constant $\rho_{\mathcal{C}} > 0$ so that for every graph $G \in \mathcal{C}$, it holds that $|E(G)| \leqslant \rho_{\mathcal{C}} \cdot |V(G)|$.

Additionally, the following facts from the theory of sparse graphs will prove useful for us. Since every apex-minor-free class of graphs (and more generally, every minor-closed class excluding at least one graph) has bounded expansion [NdM08], it has bounded *neighborhood complexity*:

Theorem 5.2.2 ([RVS19]). *There exists a constant $\nu_{\mathcal{C}} > 0$ such that for every $G \in \mathcal{C}$ and nonempty $X \subseteq V(G)$, the number of different neighborhoods of vertices of G on X is bounded by $\nu_{\mathcal{C}} \cdot |X|$. That is,*

$$|\{N(v) \cap X \mid v \in V(G)\}| \leqslant \nu_{\mathcal{C}} \cdot |X|.$$

Therefore, we have that:

Corollary 5.2.3. *If $\mathcal{C}$ is apex-minor-free, $G \in \mathcal{C}$ and $X \subseteq V(G)$ is nonempty, then*

$$|\{N(C) \mid C \in \mathrm{cc}(G \setminus X)\}| \leqslant \nu_{\mathcal{C}} \cdot |X|.$$

Proof. Produce a graph $H \in \mathcal{C}$ by contracting each connected component of $G \setminus X$ to a single vertex and apply Theorem 5.2.2 to H and X. □

Generalizing independent sets. In this chapter, we shall work with a generalization of the Max Weight Independent Set problem that is most conveniently phrased in the language of weighted CSPs. An instance I of Max Weight Nullary 2CSP consists of:

- a graph G, called the *Gaifman graph*;
- for every vertex $u \in V(G)$, a finite *domain* D_u and a *revenue function* $\mathsf{rev}_u \colon D_u \to \mathbb{R}_{\geqslant 0}$; and
- for every edge $uv \in E(G)$, a *constraint* $C_{uv} \subseteq D_u \times D_v$.

We require that each domain D_u contains a special value 0 whose revenue is 0, that is, $\mathsf{rev}_u(0) = 0$. Moreover, the value 0 is always allowed in constraints: For every edge uv, we have $\{0\} \times D_v \subseteq C_{uv}$ and $D_u \times \{0\} \subseteq C_{uv}$. We can assume that at least one domain is of size at least two, as otherwise the problem is trivial. We will concisely denote that $I = (G, D, \mathsf{rev}, C)$, where D is the set of all domains, rev is the set of revenue functions and C is the set of all constraints.

A *solution* to a Max Weight Nullary 2CSP instance as above is a mapping ϕ that with each vertex u of G associates a value $\phi(u) \in D_u$ so that $(\phi(u), \phi(v)) \in C_{uv}$ for every edge uv of G. Note that such a solution always exists, as one can map every vertex u to $0 \in D_u$. The *revenue* of a solution ϕ is defined as

$$\mathsf{rev}(\phi) \coloneqq \sum_{u \in V(G)} \mathsf{rev}_u(\phi(u)).$$

In Max Weight Nullary 2CSP one is asked to find a solution with the maximum possible revenue.

For our purposes, the domains D_u can be assumed to be small (i.e., bounded in size by $f(\varepsilon) \cdot n^{o(1)}$ for some function f). With this assumption in mind, in our time complexity analysis we will omit the time required to store and manipulate the domains and the constraints of the CSP.

We may model the Max Weight Independent Set problem in the language of Max Weight Nullary 2CSP. Let G be a graph and $\mathbf{w} \colon V(G) \to \mathbb{R}_{\geqslant 0}$ be a weight function on the vertices of G. Then construct a Max Weight Nullary 2CSP instance with Gaifman graph G as follows:

- For every vertex u of G, we set $D_u = \{0, 1\}$, $\mathsf{rev}_u(0) = 0$, and $\mathsf{rev}_u(1) = \mathbf{w}(u)$.

- For every edge uv of G, we set $C_{uv} = \{(0,0),(0,1),(1,0)\}$.

It is straightforward to see that the maximum revenue of a solution to the instance described above is equal to the maximum weight of an independent set in G.

For convenience, given an instance $I = (G, D, \mathsf{rev}, C)$ of Max Weight Nullary 2CSP and a vertex subset $Y \subseteq V(G)$, we denote by $I[Y]$ the instance *induced* by Y. This is the instance obtained from I by removing from G all vertices not in Y, and all constraints incident to a vertex outside of Y. Further, denote $I \setminus Y = I[V(G) \setminus Y]$. Note that any solution to an induced instance can be lifted to a solution to the original instance of the same revenue by mapping all the removed vertices to 0. Finally, we denote $V(I) \coloneqq V(G)$ and $E(I) \coloneqq E(G)$.

Generalizing dominating sets. Similarly to the case of the Max Weight Independent Set, we will work with a generalization of the Min Weight Dominating Set problem defined as follows. An instance of Min Weight Generalized Domination consists of:

- a multigraph G, called the *Gaifman graph*; and
- for every vertex u, a finite *domain* D_u, a *cost function* $\mathsf{cost}_u \colon D_u \to \mathbb{R}_{\geqslant 0} \cup \{+\infty\}$, a *supply function* $\mathsf{supply}_u \colon D_u \to 2^{\delta(u)}$ from the domain D_u to the set of all subsets of edges with one endpoint in u, and a *demand function* $\mathsf{demand}_u \colon D_u \to 2^{\delta(u)}$.

We additionally require that every domain D_u contains a state s_u with $\mathsf{supply}_u(s_u) = \delta(u)$ and finite cost.

A *solution* to an instance as above is a mapping ϕ that with each vertex u of G associates a value $\phi(u) \in D_u$, so that the following property holds. For every edge $e \in E(G)$ with endpoints u and v, if $e \in \mathsf{demand}_u(\phi(u))$, then $e \in \mathsf{supply}_v(\phi(v))$; and conversely, if $e \in \mathsf{demand}_v(\phi(v))$, then $e \in \mathsf{supply}_u(\phi(u))$.[19]

The cost of such a solution is defined as

$$\mathsf{cost}(\phi) = \sum_{u \in V(G)} \mathsf{cost}_u(\phi(u)).$$

The Min Weight Generalized Domination problem is to find a solution to the given instance with the minimum possible cost. Observe that every instance of Min Weight Generalized Domination has a finite-cost solution as a solution ϕ mapping every vertex u to the state s_u is valid.

The Min Weight Dominating Set problem can be modeled using Min Weight Generalized Domination as follows. Given a simple graph G and a weight function $\mathbf{w} \colon V(G) \to \mathbb{R}_{\geqslant 0}$, we create an instance $I = (G, D, \mathsf{cost}, \mathsf{supply}, \mathsf{demand})$, where for every vertex u of G we define the domain $D_u \colon N[u]$ as well as the following cost, supply, and demand functions:

$$\mathsf{cost}_u(v) = \begin{cases} \mathbf{w}(u) & \text{if } u = v, \\ 0 & \text{otherwise;} \end{cases}$$

$$\mathsf{supply}_u(v) = \begin{cases} \delta(u) & \text{if } u = v, \\ \emptyset & \text{otherwise;} \end{cases}$$

$$\mathsf{demand}_u(v) = \begin{cases} \emptyset & \text{if } u = v, \\ \{uv\} & \text{otherwise.} \end{cases}$$

It is easy to see that the minimum weight of a solution in the instance of Min Weight Generalized Domination defined above coincides with the minimum weight of a dominating set in G.

Abusing the notation, we will say that $I \in \mathcal{C}$ if $G \in \mathcal{C}$. Also, we use $V(I)$ and $E(I)$ to mean the set of vertices and the set of edges of G.

Next, we will say that a vertex $u \in V(I)$ is:

- (s,d)-*meager* for $s, d \geqslant 1$ if $|\deg(u)| \leqslant s$ and $|D_u| \leqslant d$;
- *state-monotonous* if for every ordered pair of states $x_1, x_2 \in D_u$, there exists a state x, called the *combination* of x_1 with x_2, such that

$$\begin{aligned} \mathsf{cost}_u(x) &\leqslant \mathsf{cost}_u(x_1) + \mathsf{cost}_u(x_2), \\ \mathsf{supply}_u(x) &= \mathsf{supply}_u(x_1) \cup \mathsf{supply}_u(x_2), \\ \mathsf{demand}_u(x) &\subseteq \mathsf{demand}_u(x_1). \end{aligned}$$

[19] We remark that it might be the case that in a valid solution, e belongs to both $\mathsf{demand}_u(\phi(u))$ and $\mathsf{demand}_v(\phi(v))$; in this case, we require both $\mathsf{supply}_u(\phi(u))$ and $\mathsf{supply}_v(\phi(v))$ to contain e.

Note that the definition of the combination of states above is not commutative due to the fact that the constraint on $\mathsf{demand}_u(x)$ only depends on x_1. Thus, the combination of x_1 with x_2 might be different from the combination of x_2 with x_1.

We say that an instance of Min Weight Generalized Domination is (s,d)-*decent* if every vertex of the instance is (s,d)-meager and state-monotonous. An easy verification of the definition shows that every instance of Min Weight Generalized Domination created from Min Weight Dominating Set on a graph of maximum degree Δ is $(\Delta, \Delta+1)$-decent as all vertices of the resulting instance are state-monotonous and $(\Delta, \Delta+1)$-meager. Our data structure will only operate on (s,d)-decent instances, for some small, bounded values of s and d.

Next, for $A \subseteq V(I)$ and a valuation $\phi \in \prod_{u \in A} D_u$ of vertices in A, we say that the valuation is *locally correct on A to I* if, for every edge e with both endpoints u, v in A, if $e \in \mathsf{demand}_u(\phi(u))$, then $e \in \mathsf{supply}_v(\phi(v))$. Note that if ϕ is a correct solution to I, then $\phi|_A$ is a locally correct solution on A to I.

Given an instance I of Min Weight Generalized Domination and a vertex $u \in V(I)$, to *relieve* u in I is to produce an instance I' by setting $\mathsf{demand}_u(x) \leftarrow \emptyset$ for all $x \in D_u$. In other words, relieving marks u as a vertex that does not need to be dominated by a solution to I'. Note that if there exists a solution to I, then a solution to I' also exists and the minimum cost of such a solution is less than or equal to the minimum cost of a solution to I.

Finally, given a set $A \subseteq V(G)$, we define the *A-cleared subinstance of I*, denoted $\mathsf{Clear}(I; A)$, as the instance produced from I by:

- removing all the edges whose both endpoints are in A, erasing them also from the corresponding demand and supply sets;
- relieving each vertex of A in the resulting instance.

The following fact is straightforward.

Observation 5.2.4. *If ϕ is a correct solution to I, then it is also a correct solution to $\mathsf{Clear}(I; A)$.*

5.3 Maximum Weight Independent Set

In this section we give a data structure that implements the first query, $\mathsf{QueryMWIS}()$, of Theorem 1.3.7. As mentioned, we will actually solve a more general problem of Max Weight Nullary 2CSP (which we will call 2CSP from now on for brevity). That is, we will maintain a dynamic instance of 2CSP undergoing the following types of updates:

- $\mathsf{AddVertex}(u, D_u, \mathsf{rev}_u)$: Adds an isolated vertex u together with the domain D_u and the revenue function $\mathsf{rev}_u \colon D_u \to \mathbb{R}_{\geqslant 0}$;
- $\mathsf{AddEdge}(u, v, C_{uv})$: Adds an edge uv together with the constraint $C_{uv} \subseteq D_u \times D_v$;
- $\mathsf{RemoveEdge}(u, v)$: Removes an edge uv;
- $\mathsf{UpdateRevenue}(u, \mathsf{rev}_u)$: Changes the revenue function of u to $\mathsf{rev}_u \colon D_u \to \mathbb{R}_{\geqslant 0}$.

After each update, the data structure should maintain an approximate optimum revenue to the currently stored the instance I: a nonnegative real p satisfying $(1-\varepsilon)\mathsf{OPT} \leqslant p \leqslant \mathsf{OPT}$, where OPT denotes the optimum revenue for I. For technical reasons, we do not support removing vertices from the instance; however, observe that the removal of a vertex can be emulated by removing all edges incident to the vertex and replacing its revenue with the constant-zero function.

Note that the original dynamic instance of 2CSP constructed from Max Weight Independent Set has a fixed n-element vertex set, so it will never be updated by $\mathsf{AddVertex}$. However, the designed data structure will create auxiliary dynamic instances of 2CSP that *will* be maintained using all four types of updates.

Henceforth, we assume that a given apex-minor-free class of graphs $\mathcal{C}$ is fixed; that is, we consider all constants depending only on $\mathcal{C}$ to be absolute constants. Moreover, throughout this section we denote by $\varepsilon > 0$ the parameter ε fixed in the initialization of the data structure.

5.3.1 Introductory results

We begin by summarizing the Baker's technique applied to 2CSP. In short, we show that if the Gaifman graph has bounded treewidth, then we can solve 2CSP exactly using a standard dynamic programming over the tree decomposition. On the other hand, if the Gaifman graph belongs to $\mathcal{C}$, then we can solve the instance approximately by a classical application of the Baker's technique [Bak94].

First, the following lemma is obtained by dynamic programming on a tree decomposition.

Lemma 5.3.1. *Let $I = (G, D, \mathsf{rev}, C)$ be an instance of* 2CSP *on n vertices and $\Delta \geqslant 2$ be the maximum size of a domain of any vertex. Then, we can solve I in time $n \cdot \Delta^{\mathcal{O}(\mathrm{tw}(G))}$.*

Proof. First, we can use an algorithm computing a tree decomposition that is a 2-approximation of an optimal one in time $n \cdot 2^{\mathcal{O}(\mathrm{tw}(G))}$ by Korhonen [Kor21]. Once we have that approximation, we use standard dynamic programming over it that could be seen as a generalization of a similar algorithm solving maximum independent set over graphs with bounded treewidth. For each bag of that decomposition it suffices to have dynamic programming states indexed by all possible valuations of variables from that bag and the transitions follow easily. That dynamic programming takes time $n \cdot \Delta^{\mathcal{O}(\mathrm{tw}(G))}$, so as $\Delta \geqslant 2$, in total this algorithm takes time $n \cdot 2^{\mathcal{O}(\mathrm{tw}(G))} + n \cdot \Delta^{\mathcal{O}(\mathrm{tw}(G))} = n \cdot \Delta^{\mathcal{O}(\mathrm{tw}(G))}$. □

Next, the following claim shows how subinstances of small treewidth are constructed by Baker's technique:

Lemma 5.3.2. *Let $G \in \mathcal{C}$ be a connected graph and fix $s \in V(G)$. Then:*

- *For integers $\ell \geqslant 0$, $k \geqslant 1$, we have $\mathrm{tw}(G[\{v \in V(G) \mid \ell \leqslant \mathsf{dist}(s, v) < \ell + k\}]) \leqslant \mathcal{O}(k)$;*
- *For integers $k > i \geqslant 0$, let $V_i = \{v \in V(G) \mid \mathsf{dist}(s, v) \equiv i \mod k\}$. Then $\mathrm{tw}(G \setminus V_i) \leqslant \mathcal{O}(k)$.*

Proof. For the first point, let $G \in \mathcal{C}$ be connected and fix $s \in V(G)$, $k \geqslant 1$ and $\ell \geqslant 0$. Let G' be a minor of G created by:

- discarding from G' all vertices at distance at least $\ell + k$ from s;
- if $\ell > 0$, contracting all vertices at distance strictly less than ℓ to s.

It can be readily verified that the radius of G' is bounded from above by k since s is at distance at most k from every vertex of G'. Thus by Theorem 5.2.1, we have $\mathrm{tw}(G') \leqslant \mathcal{O}(k)$. Since $G[A] = G'[A]$, we conclude that $\mathrm{tw}(G[A]) \leqslant \mathrm{tw}(G')$.

For the second point, choose integers i, k with $0 \leqslant i < k$. Observe that $G \setminus V_i$ is a disjoint union of the subgraphs induced by at most $k - 1$ consecutive BFS layers (i.e., for every pair of vertices in a single component, their distance from s in G differs by at most $k-1$). Hence, the first point of the lemma applies, implying that each of these subgraphs has treewidth at most $\mathcal{O}(k)$. Therefore, $\mathrm{tw}(G \setminus V_i) \leqslant \mathcal{O}(k)$. □

With these helper lemmas in hand, we can prove that:

Lemma 5.3.3. *Let $\varepsilon' > 0$ be a positive real number and $I = (G, D, \mathsf{rev}, C)$, $G \in \mathcal{C}$, be an instance of* 2CSP *with n vertices, where the maximum size of a domain is $\Delta \geqslant 2$. One can compute a real p such that $(1 - \varepsilon')\mathsf{OPT} \leqslant p \leqslant \mathsf{OPT}$ in time $n \cdot \Delta^{\mathcal{O}(1/\varepsilon')}$, where OPT is the maximum revenue of a solution to I.*

Proof. We solve Max Weight Nullary 2CSP by a classical use of the Baker's technique. Let $k = \lceil \frac{1}{\varepsilon'} \rceil$. We assume that G is connected (because we can solve each connected component separately), choose an arbitrary root r, partition the graph into the BFS layers and group them into groups mod k. That is, let $V_i = \{v \mid \mathsf{dist}_G(r, v) \equiv i \mod k\}$ for $i = 0, 1, \ldots, k-1$. Then let $I_i = I \setminus V_i$. By Lemma 5.3.2, the Gaifman graph of each I_i has treewidth at most $\mathcal{O}(k)$. Now, using Lemma 5.3.1, we compute an optimum solution to each I_i; call it ϕ_i and set $p_i = \mathsf{rev}(\phi_i)$. We claim that it suffices to set $p := \max_i p_i$.

For each $i \in \{0, \ldots, k-1\}$, we extend ϕ_i to a valid solution ϕ_i' of I by placing value 0 on each variable from V_i. If ϕ is an optimum solution to I, so that $\mathsf{OPT} = \mathsf{rev}(\phi)$, then $\phi|_{V(G) \setminus V_i}$ is a valid solution to I_i, hence $\mathsf{rev}(\phi|_{V(G)\setminus V_i}) \leqslant \mathsf{rev}(\phi_i)$. As $V_0, V_1, \ldots, V_{k-1}$ form a partition of $V(G)$, we have that $\mathsf{rev}(\phi|_{V_0}) + \ldots + \mathsf{rev}(\phi|_{V_{k-1}}) = \mathsf{rev}(\phi)$, so there exists some i such that $\mathsf{rev}(\phi|_{V_i}) \leqslant \frac{\mathsf{rev}(\phi)}{k} \leqslant \varepsilon' \mathsf{OPT}$. Therefore, we have that $p \geqslant \mathsf{rev}(\phi_i) \geqslant \mathsf{rev}(\phi|_{V(G)\setminus V_i}) = \mathsf{rev}(\phi) - \mathsf{rev}(\phi|_{V_i}) \geqslant (1 - \varepsilon')\mathsf{OPT}$. As each ϕ_i' is a valid solution to I and we have that $\mathsf{rev}(\phi_i) = \mathsf{rev}(\phi_i')$, we obviously have that $p \leqslant \mathsf{OPT}$, which completes the proof of the claim.

Solving each I_i requires time $n \cdot \Delta^{\mathcal{O}(k)}$. As we solve k auxiliary instances, the total time complexity is $n \cdot k \cdot \Delta^{\mathcal{O}(k)} = n \cdot \Delta^{\mathcal{O}(1/\varepsilon')}$. □

We then recall a useful technical tool allowing us to construct elimination forests graphs of small width and height.

Lemma 2.3.4 ([KNPS24]). *Let G be a graph on n vertices given together with a tree decomposition of width w with $\mathcal{O}(n)$ nodes. Then, in time $\mathcal{O}(wn \log n)$, one can compute an elimination forest F of G with the following properties:*

1. *F has height $\mathcal{O}(w \cdot \log n)$;*
2. *for each $u \in V(F)$, we have $|\mathsf{Reach}_F(u)| = \mathcal{O}(w)$;*
3. *for each $u \in V(F)$, the graph $G[\mathsf{desc}_F[u]]$ is connected.*

The following observation shows how property (3) of Lemma 2.3.4 can be exploited in algorithms working with elimination forests.

Lemma 5.3.4. *Assume F is an elimination forest of G such that for each $u \in V(F)$, the graph $G[\mathsf{desc}_F[u]]$ is connected. If $u, v \in V(F)$ are nonroot vertices of F and $\mathsf{parent}_F(u) \neq \mathsf{parent}_F(v)$, then it holds that $\mathsf{Reach}_F(u) \neq \mathsf{Reach}_F(v)$.*

Proof. It is always the case that $\mathsf{parent}_F(u) \in \mathsf{Reach}_F(u)$ as otherwise $\mathsf{desc}_F[\mathsf{parent}_F(u)]$ would not be connected. Also, $\mathsf{Reach}_F(u) \subseteq \mathsf{anc}_F[\mathsf{parent}_F(u)]$, hence $\mathsf{parent}_F(u)$ is the (unique) deepest vertex of $\mathsf{Reach}_F(u)$. This finishes the proof. □

5.3.2 Compressed instances

We now define the crucial notion of a *compressed instance* of 2CSP. Intuitively, compression of instances provides us with a means of producing significantly smaller instances of 2CSP with the same maximum revenue. We begin with the definition of a compressed graph:

Definition 11. *Let G be a graph and Y be a subset of $V(H)$. By $G\{Y\}$ we denote the* compressed graph *for G and Y, defined as follows:*

$$V(G\{Y\}) = Y \cup \{N(C) \mid C \in \mathrm{cc}(G \setminus Y)\} \subseteq Y \cup 2^Y,$$
$$E(G\{Y\}) = E(G[Y]) \cup \{vS \mid v \in S,\, S \in V(G\{Y\}) \setminus Y\}.$$

In other words, we group the connected components of $G \setminus Y$ by their neighborhoods and for each such group we create one vertex connected to all vertices from the neighborhood of this group.

Next we say what it means to compress an instance.

Definition 12. *Let $I = (G, D, \mathsf{rev}, C)$ be a* 2CSP *instance and Y a subset of $V(G)$. We define the compressed instance $I\{Y\} = (G\{Y\}, D', \mathsf{rev}', C')$ as follows.*

- (Domains.) *If $v \in Y$, then $D'_v = D_v$. If $S \in V(G\{Y\}) \setminus Y$, then $D'_S = \{0\} \cup \prod_{v \in S} D_v$.*
- (Revenues.) *If $v \in Y$, then $\mathsf{rev}'_v = \mathsf{rev}_v$. Now assume $S \in V(G\{Y\}) \setminus Y$. For convenience, let $S = \{u_1, \ldots, u_s\}$ and let R be the union of all connected components of $G \setminus Y$ whose neighborhood is S. Let $I_S = I[R \cup S]$. Then $\mathsf{rev}'_S(0) = 0$, and $\mathsf{rev}'_S(d_1, \ldots, d_s)$ for $d_i \in D_{u_i}$ is the maximum revenue of $\mathsf{rev}(\phi|_R)$, where ϕ ranges over all valid solutions to I_S such that $\phi(u_i) = d_i$ for each $1 \leqslant i \leqslant s$. If no such solutions exist, then we set $\mathsf{rev}'_S(d_1, \ldots, d_s) = 0$.*
- (Constraints.) *If $u, v \in Y$, then $C'_{uv} = C_{uv}$. If $S \in V(G\{Y\}) \setminus Y$ and $u \in S$, then*

  $$C'_{uS} = \{(d, e) \mid d \in D'_u \text{ and } e \in D'_S,\, d = 0 \vee e = 0 \vee e(u) = d\}.$$

  In other words, the constraint C'_{uS} permits the valuations ϕ of $G\{Y\}$ for which $\phi(u) = 0$ or $\phi(S) = 0$, or for which $\phi(S) \in \prod_{v \in S} D_v$ and $\phi(S)(u) = \phi(u)$.

It can be easily verified that for every I and every Y, $I\{Y\}$ is a correct instance of Max Weight Nullary 2CSP. Also, note that in the case of defining $\mathsf{rev}'_S(d_1, \ldots, d_s)$, no feasible ϕ exists if and only if setting $\phi(u_i) = d_i$ for each $1 \leqslant i \leqslant s$ already violates some constraint, as we can always put ϕ as a zero function on R.

In other words, given a 2CSP instance I and a set $Y \subseteq V(I)$, we define the compression $I\{Y\}$ by contracting the connected components of $G \setminus Y$ to single vertices, and identifying the contracted vertices with the same neighborhood S in Y to the same vertex x. Then, for each possible valuation $d \in \prod_{v \in S} D_v$ of vertices in S, we encode in x the maximum revenue of a partial solution for the connected components assigned to x that is consistent with d. Note that after a valuation of vertices in S is fixed, the instances of 2CSP induced by each connected component assigned to x are pairwise independent: there are no constraints between the connected components of $G \setminus Y$. Therefore:

Observation 5.3.5. *Let $I = (G, D, \mathsf{rev}, C)$ be a* 2CSP *instance and let $Y \subseteq V(G)$. Let $S \in V(G\{Y\}) \setminus Y$, $S = \{u_1, \ldots, u_s\}$, and $C_1, C_2, \ldots, C_\ell$ be the connected components of $G \setminus Y$ whose neighborhood is S. For each $1 \leqslant i \leqslant \ell$, let $I_i = I[C_i \cup S]$. Let also $d_1 \in D_{u_1}, \ldots, d_s \in D_{u_s}$. Then, $\mathsf{rev}'_S(d_1, \ldots, d_s) = \sum_{i=1}^{\ell} \mathsf{rev}(\phi_i|_{C_i})$, where ϕ_i is a solution to I_i maximizing $\mathsf{rev}(\phi_i|_{C_i})$ under the assumption that $\phi_i(u_j) = d_j$ for all $1 \leqslant j \leqslant s$, or $\mathsf{rev}'_S(d_1, \ldots, d_s) = 0$ if setting $\phi(u_j) = d_j$ for all $1 \leqslant j \leqslant s$ violates some constraint.*

Having defined compressed graphs and instances, we now present a series of their properties. Namely: $\mathcal{C}$ is closed under graph compressions and compressed instances have the same optimum revenues as the original instances.

Lemma 5.3.6. *Let G be a graph and Y be a subset of $V(H)$. Then $G \in \mathcal{C}$ implies $G\{Y\} \in \mathcal{C}$.*

Proof. $H\{Y\}$ can be obtained from H through a series of contractions (where we contract each connected component of $H \setminus Y$ into a single vertex) and vertex removals (we remove all but one vertices from each group that has a particular neighborhood in Y). Hence, $H\{Y\}$ is a minor of H. Since $\mathcal{C}$ is assumed to be minor-closed, the claim follows. □

Lemma 5.3.7. *Let $I = (G, D, \mathsf{rev}, C)$ be a* 2CSP *instance, $Y \subseteq V(G)$ and $I\{Y\} = (G\{Y\}, D', \mathsf{rev}', C')$ be the compressed instance. Then, the optimum solutions of I and $I\{Y\}$ have equal revenues.*

Proof. Let OPT and OPT' denote the revenues of optimum solutions to I and $I\{Y\}$.

Let ϕ be any solution to I. We shall define ϕ' which will be a valid solution to $I\{Y\}$. We define ϕ' to agree with ϕ on Y. Then $\mathsf{rev}'(\phi'|_Y) = \mathsf{rev}(\phi|_Y)$. Now choose $S \in \{N(E) \mid E \in \mathrm{cc}(H \setminus Y)\}$. Let $S = \{u_1, \ldots, u_s\}$ and $\phi(u_1) = d_1, \ldots, \phi(u_s) = d_s$. We set $\phi'(S) \coloneqq (d_1, \ldots, d_s)$. It can be readily verified that ϕ' complies with all the constraints in C', so ϕ' is valid. Now, if R is the union of all the connected components of $H \setminus Y$ whose neighborhood is equal to S, then $\mathsf{rev}'(\phi'(S)) \geqslant \mathsf{rev}(\phi|_R)$ from the definition of rev'. By adding such inequalities for all possible pairs of S and R and the equality $\mathsf{rev}'(\phi'|_Y) = \mathsf{rev}(\phi|_Y)$, we conclude that $\mathsf{rev}'(\phi') \geqslant \mathsf{rev}(\phi)$, hence $\mathsf{OPT}' \geqslant \mathsf{OPT}$.

On the other hand, let ψ' be any solution to $I\{Y\}$. We shall define ψ – a valid solution to I. We define ψ to agree with ψ' on Y, hence $\mathsf{rev}(\psi|_Y) = \mathsf{rev}'(\psi'|_Y)$. Now choose $S \in V(H\{Y\}) \setminus Y$ and assume that $S = \{u_1, \ldots, u_s\}$. Let R be the union of all the connected components of $H \setminus Y$ whose neighborhood is equal to S. If $\mathsf{rev}'(\psi'(S)) = 0$, then we set $\psi(v) \coloneqq 0$ for all $v \in R$. Otherwise, $\mathsf{rev}'(\psi'(S)) > 0$, so $\psi'(S) \in \prod_{v \in S} D_v$. In this case, from the definition of $C'_{u_i S}$, it has to be that $\psi'(S)(u_1) = \psi'(u_1) = \psi(u_1), \ldots, \psi'(S)(u_s) = \psi'(u_s) = \psi(u_s)$, and let $I_S = I[R \cup S]$. Since $\mathsf{rev}'(\psi'(S)) > 0$, there exists a solution ζ to I_S such that $\zeta(u_i) = \psi'(u_i) = \psi(u_i)$ for all $1 \leqslant i \leqslant s$. Let ζ be any such maximum-revenue solution, and set $\psi|_R \coloneqq \zeta|_R$. Then $\mathsf{rev}(\psi|_R) = \mathsf{rev}'(\psi'(S))$ by the definition of rev'. It is also easy to verify that ψ is valid. It follows that $\mathsf{rev}(\psi) = \mathsf{rev}'(\psi')$, so $\mathsf{OPT} \geqslant \mathsf{OPT}'$. □

5.3.3 Dynamic maintenance of compressed instances

We proceed to show how the notion of compressed instances can be used in the setting of the dynamic maintenance of 2CSPs. Consider an instance $I = (G, D, \mathsf{rev}, C)$ of 2CSP, where $G \in \mathcal{C}$ and $\mathrm{tw}(G) \leqslant w$ for some $w \in \mathbb{N}$. The instance I undergoes a sequence of updates, so that at every point of time we have that $G \in \mathcal{C}$ (but it might not necessarily be the case that $\mathrm{tw}(G) \leqslant w$). Our goal is to maintain a much smaller 2CSP instance $I^\star$ with the same value of optimum solution as I, so that at all points of time, the size of $I^\star$ is roughly proportional to the number of updates I has undergone so far. This is formalized by the following lemma.

Lemma 5.3.8. *Let $w, n, \Delta \in \mathbb{N}$, $\Delta \geqslant 2$. There is a data structure that supports the following operations:*

- ***Initialize** the data structure with an n-vertex* 2CSP *instance $I = (G, D, \mathsf{rev}, C)$, where $G \in \mathcal{C}$, $\mathrm{tw}(G) \leqslant w$, and all vertices in I have domains not larger than Δ.*
- ***Update** the instance I using one of the following update types:* AddVertex, AddEdge, RemoveEdge, UpdateRevenue. *It is guaranteed that after the update, we have that $G \in \mathcal{C}$ and all vertices in I have domains not larger than Δ.*

The initialization of the data structure is performed in time $\Delta^{\mathcal{O}(w)} \cdot n \log n$. Afterwards, the data structure additionally maintains a 2CSP *instance $I^\star = (G^\star, D^\star, \mathsf{rev}^\star, C^\star)$, updated by* AddVertex, AddEdge, RemoveEdge *and* UpdateRevenue, *with the following properties:*

(a) $G^\star \in \mathcal{C}$;

(b) the maximum revenue of a solution to $I^\star$ is equal to that of I;

(c) each vertex of $I^\star$ has domain bounded in size by $\Delta^{\mathcal{O}(w)}$;

(d) after a sequence of $t \geqslant 0$ updates to I, we have $|V(I^\star)| \leqslant t \cdot w^{\mathcal{O}(1)} \log n$. Moreover, on each update to I, the instance $I^\star$ can be updated in time $\Delta^{\mathcal{O}(w)} \log n$ and this causes at most $w^{\mathcal{O}(1)} \log n$ updates to $I^\star$.

Proof. On initialization, we compute a tree decomposition of G of width at most $\mathcal{O}(w)$ in time $2^{\mathcal{O}(w)} \cdot n$, for instance using the fixed-parameter algorithm of Korhonen [Kor21]. We then transform this tree decomposition into an elimination forest F of G with $V(F) = V(G)$ such that: (i) F has height $\mathcal{O}(w \log n)$; (ii) for each $u \in V(F)$, we have $|\mathsf{Reach}_F(u)| = \mathcal{O}(w)$; and (iii) for each $u \in V(F)$, the graph $G[\mathsf{desc}_F[u]]$ is connected. This can be done in time $\mathcal{O}(w \cdot n \log n)$ by Lemma 2.3.4.

Moreover, we precompute multiple dynamic programming tables on F:

- For each $u \in V(F)$ and for each possible valuation $\phi \in \prod_{s \in \mathsf{Reach}_F(u)} D_s$ of vertices from $\mathsf{Reach}_F(u)$, compute $T[u][\phi]$: the maximum possible revenue $\mathsf{rev}(\psi|_{\mathsf{desc}_F[u]})$ over all solutions ψ to $I[\mathsf{desc}_F[u] \cup \mathsf{Reach}_F(u)]$ agreeing with ϕ on $\mathsf{Reach}_F(u)$, or 0 if no such solution exists. Note that $|\mathsf{Reach}_F(u)| = \mathcal{O}(w)$, hence there are at most $\Delta^{\mathcal{O}(w)}$ possible valuations of ϕ on $\mathsf{Reach}_F(u)$ for a given u. Also note that for a vertex u with children $c_1, \ldots, c_s$ we have that $\mathsf{Reach}_F(c_i) \subseteq \mathsf{Reach}_F(u) \cup \{u\}$. Hence, we can compute all values $T[\cdot][\cdot]$ using a simple bottom-up dynamic programming in time $n \cdot \Delta^{\mathcal{O}(w)}$.
- For each $u \in V(F)$, compute the set $\mathcal{N}_u = \{\mathsf{Reach}_F(c) \mid c \text{ is a child of } u \text{ in F}\}$. This can be performed easily in time $\mathcal{O}(nw)$. Note that $R \subseteq \mathsf{Reach}_F(u) \cup \{u\}$ for all $R \in \mathcal{N}_u$.
- For each $u \in V(F)$, $R \in \mathcal{N}_u$ and valuation $\phi \in \prod_{s \in R} D_s$ of vertices from R, compute $W[u][R][\phi]$: the sum of $T[c][\phi]$ such that c is a child of u and $\mathsf{Reach}_F(c) = R$. This table can be computed in time $n \cdot \Delta^{\mathcal{O}(w)}$ as well: for each nonroot vertex c and each valuation ϕ of $\mathsf{Reach}_F(c)$, the value $T[c][\phi]$ is added to exactly one entry of W.

We remark that the dynamic programming tables above suffice to compute the maximum-revenue solution for I at the time of the initialization: It is simply the sum over $T[r][\emptyset]$ ranging over all roots r of trees of the elimination forest F.

Also initialize sets $A = B = Z = \emptyset$ and $I^\star = I\{\emptyset\}$. We remark that $I\{\emptyset\}$ is a one-vertex instance of 2CSP with the same revenue of the optimum solution as I.

Let $I^{\text{init}} = (G^{\text{init}}, D^{\text{init}}, \mathsf{rev}^{\text{init}}, C^{\text{init}})$ be the initial instance of 2CSP. In the sequel, by $I^{\text{cur}} = (G^{\text{cur}}, D^{\text{cur}}, \mathsf{rev}^{\text{cur}}, C^{\text{cur}})$ we denote the current snapshot of the instance in the data structure; initially, $I^{\text{cur}} := I^{\text{init}}$. Note that $V(I^{\text{init}}) \subseteq V(I^{\text{cur}})$ at all times since vertices cannot be explicitly removed from I. Instead, we emulate the removals of vertices from 2CSP by removing all the edges incident to the vertex and setting the revenue of the vertex to the zero function. If two instances of 2CSP are isomorphic after removing from both of them the sets of isolated vertices with zero revenues, we say that these instances are *equivalent*.

Throughout the life of the data structure, we maintain the following invariants:

- $A = V(I^{\text{cur}}) \setminus V(I^{\text{init}})$ is the set of vertices added to I since the instantiation of the structure.
- $B \subseteq V(I^{\text{cur}})$ is the set of vertices that were part of any update to I so far (i.e., $v \in B$ if v was added to I, the revenue of v was changed, or an edge incident to v was added or removed).
- $Z = A \cup \mathsf{anc}_F[B \setminus A]$, i.e., Z contains A and all the ancestors in F of all vertices of $B \setminus A$.
- $I^\star$ is equivalent to $I^{\text{cur}}\{Z\}$.

Note that it follows that $A \subseteq B$, $V(I^{\text{cur}}) = V(I^{\text{init}}) \cup A$ and $B \setminus A \subseteq V(I^{\text{init}})$. Also, the invariants are satisfied at the time of initialization. Moreover, the required properties (a), (b) and (c) follow from the invariant: Since $I^\star$ is equivalent to $I^{\text{cur}}\{Z\}$, property (a) follows from Lemma 5.3.6, property (b) follows from Lemma 5.3.7, and property (c) follows from the definition of a compressed instance and the fact that the domain of each vertex of I has size bounded by Δ.

We envision Z as consisting of two parts – an "unordered cloud" of newly added vertices (which is A) and a well-structured part that is the smallest prefix of F containing all vertices from $V(I^{\text{init}})$ that were part of any update. Note that for any vertex $v \notin Z$, we have $v \notin B$. Therefore, any $v \notin Z$ is not adjacent to any vertex of A, and is adjacent to some vertex of I^{cur} if and only if it is adjacent to this vertex in I^{init}. In other words, the neighborhood of v has stayed unchanged; that is, $v \notin Z \Rightarrow N_{G^{\text{cur}}}(v) = N_{G^{\text{init}}}(v)$. The

main idea of our approach is that we treat Z as a possibly complicated part, whereas the behavior of the vertices outside of Z can be well-understood through the preprocessed dynamic programming tables. Thus, we can effectively compress the parts of the graph outside of Z and construct a concise instance of 2CSP without changing the revenue of the optimum solution. We will prove that the size of the compressed instance depends mainly on the size of Z – intuitively, we do not compress Z in any way, but the parts of the graph outside of Z will be compressed heavily. Fortunately, the size of Z cannot increase too much with a single update:

Claim 5.3.9. *The set Z expands by at most $\mathcal{O}(w \log n)$ vertices with each update to I.*

Proof of the claim. Naturally, all sets A, B, Z can only expand after each update.

If the update to I is of type $\mathsf{AddVertex}(u, D_u, \mathsf{rev}_u)$, then sets A and B grow by one vertex u, and $\mathsf{anc}_F[B \setminus A]$ does not change. Hence Z grows by exactly one vertex.

For our convenience, let P_u for a vertex $u \in V(G^{\mathrm{cur}})$ be the empty set if $u \in A$ and $\mathsf{anc}_F[u]$ otherwise. If the update is of type $\mathsf{AddEdge}(u, v, c)$ or $\mathsf{RemoveEdge}(u, v)$, then A does not change, B expands by u and v (if these vertices were not part of B yet), and $\mathsf{anc}_F[B \setminus A]$ grows by vertices that are contained within $P_u \cup P_v$. Since F is of height $\mathcal{O}(w \log n)$, it follows that Z grows by $\mathcal{O}(w \log n)$ vertices.

Finally, if the update is of type $\mathsf{UpdateRevenue}(v, r_v)$, then A does not change, B additionally includes v (if not in B yet), and $\mathsf{anc}_F[B \setminus A]$ grows by a set of vertices that is contained within P_v, which again is of size $\mathcal{O}(w \log n)$. ◁

Let us now understand the structure of the compressed instance $I^{\mathrm{cur}}\{Z\}$. Recall that for each $u \in V(F)$, it holds that $\mathsf{desc}_F[u]$ induces a connected graph in G^{init}. Therefore, if $u \notin Z$, then $\mathsf{desc}_F[u]$ induces a connected subgraph of G^{cur} as well. If additionally $\mathsf{parent}_F(u) \in Z$, then $N_{G^{\mathrm{cur}}}(\mathsf{desc}_F[u]) = N_{G^{\mathrm{init}}}(\mathsf{desc}_F[u]) = \mathsf{Reach}_F(u) \subseteq \mathsf{anc}_F(u) \setminus \{u\} \subseteq Z$, so we conclude that $\mathsf{desc}_F[u]$ is actually a connected component of $G^{\mathrm{cur}} \setminus Z$. Hence, there exists a natural bijection between the connected components of $G^{\mathrm{cur}} \setminus Z$ and the vertices u such that $u \notin Z$, but $\mathsf{parent}_F(u) \in Z$ (or $u \notin Z$ is the root of some tree in F). We will call all such vertices u *appendices* of Z in F.

Observe that the revenues of the vertices in the compressed instance can be inferred from the entries of the precomputed dynamic programming table $T[\cdot][\cdot]$. In the instance $I^{\mathrm{cur}}\{Z\}$, consider $S \in V(G\{Z\}) \setminus Z$ and let $\phi \in \prod_{s \in S} D_s$ be a valuation of vertices in S. Then, $\mathsf{rev}^{\mathrm{cur}}(S, \phi)$ is by Observation 5.3.5 exactly the sum of $T[u][\phi]$ over all appendices u of Z in F such that $\mathsf{Reach}_F(u) = S$.

It turns out that it is possible to maintain $G^{\mathrm{cur}}\{Z\}$ efficiently given the precomputed tables T and W. To this end, we claim that $G\{Z\}$ can be updated efficiently when an appendix of Z is added to Z.

Claim 5.3.10. *Let two sets $Z_1, Z_2 \subseteq V(G^{\mathrm{cur}})$ be such that $A \subseteq Z_1 \subseteq Z_2 \subseteq V(G^{\mathrm{cur}})$, $|Z_2| = |Z_1| + 1$ and $Z_1 \setminus A$ and $Z_2 \setminus A$ are prefixes of F with $Z_2 - Z_1 \subseteq V(F)$. Then, an instance equivalent to $I^{\mathrm{cur}}\{Z_2\}$ can be obtained from an instance equivalent to $I^{\mathrm{cur}}\{Z_1\}$ through a sequence of $w^{\mathcal{O}(1)}$ updates. Moreover, this sequence can be computed in time $\Delta^{\mathcal{O}(w)}$.*

Proof of the claim. Let $z \in V(F)$ be such that $Z_2 = Z_1 \cup \{z\}$. Let $\mathfrak{C}_1$ be the set of connected components of $G^{\mathrm{cur}} \setminus Z_1$, $\mathfrak{C}_2$ be the set of connected components of $G^{\mathrm{cur}} \setminus Z_2$ and $c_1, \ldots, c_t$ be the set of children of z in F. We have that $\mathfrak{C}_1 \setminus \mathfrak{C}_2 = \{G^{\mathrm{cur}}[\mathsf{desc}_F[z]]\}$ and $\mathfrak{C}_2 \setminus \mathfrak{C}_1 = \{G^{\mathrm{cur}}[\mathsf{desc}_F[c_1]], \ldots, G^{\mathrm{cur}}[\mathsf{desc}_F[c_t]]\}$.

Let $S = \mathsf{Reach}_F(z)$. The compressed instance $I^{\mathrm{cur}}\{Z_1\}$ contains a vertex v_S representing the union of all connected components of $G^{\mathrm{cur}} \setminus Z_1$ whose neighborhoods are exactly S; and $G^{\mathrm{cur}}[\mathsf{desc}_F[z]]$ is one of such components. To obtain $I^{\mathrm{cur}}\{Z_2\}$ from $I^{\mathrm{cur}}\{Z_1\}$, we need to:

1. Remove the contribution of $G^{\mathrm{cur}}[\mathsf{desc}_F[z]] \in \mathfrak{C}_1 \setminus \mathfrak{C}_2$ from the compressed instance. If $\mathsf{desc}_F[z]$ is the only connected component of $G^{\mathrm{cur}} \setminus Z_1$ with neighborhood S, then the vertex v_S should be removed from the instance; this is emulated by removing all edges incident to v_S and replacing the revenue of v_S with the zero function. Otherwise, the revenue of v_S is updated by subtracting, for each valuation $\phi \in \prod_{s \in S} D_s$ of S, the entry $T[v][\phi]$ from the revenue of vertex v_S in state ϕ. In both cases, the compressed vertex v_S has degree $|\mathsf{Reach}_F(z)| \leqslant \mathcal{O}(w)$, so in either case we apply at most $\mathcal{O}(w)$ updates to the compressed instance.

2. Add the vertex z to the compressed instance. Since the set of neighbors of z in Z_1 is exactly $\mathsf{Reach}_F(z)$, this requires one vertex addition and $\mathcal{O}(w)$ constraint additions.

3. Include the contribution of the connected components $G^{\mathrm{cur}}[\mathsf{desc}_F[c_1]], \ldots, G^{\mathrm{cur}}[\mathsf{desc}_F[c_t]] \in \mathfrak{C}_2 \setminus \mathfrak{C}_1$ in the compressed instance. Even though t might possibly be large, the number of different neighborhoods of the new connected components is bounded – this follows from $N_{G^{\mathrm{cur}}}(\mathsf{desc}_F[c_i]) = \mathsf{Reach}_F(c_i) \subseteq$

$\mathsf{Reach}_F(z) \cup \{z\}$. Since $|\mathsf{Reach}_F(z)| \leqslant \mathcal{O}(w)$, by Corollary 5.2.3 there exist at most $\mathcal{O}(w)$ distinct neighborhoods of the new components; in other words, $|\mathcal{N}_z| \leqslant \mathcal{O}(w)$. By Lemma 5.3.4, all these neighborhoods actually correspond to new vertices in the compressed instance. Thus, for each $S' \in \mathcal{N}_z$ (equivalently, for each $S' \subseteq \mathsf{Reach}_F(z) \cup \{z\}$ with at least one component $\mathsf{desc}_F[c_i]$ with the neighborhood equal to S'), we add to the compressed instance a new vertex $v_{S'}$. For each valuation $\phi \in \prod_{s \in S'} D_s$ of vertices in S', the revenue of $v_{S'}$ in state ϕ is the sum over all $T[c_i][\phi]$ for all $1 \leqslant i \leqslant t$ such that $\mathsf{Reach}_F(c_i) = S'$; this is exactly the preprocessed value $W[z][S'][\phi]$. Each of the new $\mathcal{O}(w)$ vertices has degree at most $\mathcal{O}(w)$, so the number of updates to the compressed instance is $\mathcal{O}(w^2)$. For each fresh vertex $v_{S'}$, we iterate over all $\Delta^{\mathcal{O}(w)}$ valuations of S' to produce the revenues for each state of $v_{S'}$. This can be done in total time $\mathcal{O}(w) \cdot \Delta^{\mathcal{O}(w)} = \Delta^{\mathcal{O}(w)}$. ◁

Now the proof of the lemma follows easily from Claim 5.3.9 and Claim 5.3.10. If the update to I^{cur} is of type $\mathsf{AddVertex}(r_u)$, then Z grows by one isolated vertex and we just pass that update to $I^\star$. Otherwise, the set A does not change and Z grows by $\mathcal{O}(w \log n)$ vertices from F that can be easily determined. We process the additions to Z vertex by vertex, applying Claim 5.3.10 on each addition. Note that the vertices should be added to Z in the order of the increasing distance from the root of F, so as to ensure that $Z \setminus A$ remains a prefix of F at all times. Finally, after all the required vertices are included in Z, we relay the queried update of I^{cur} to $I^\star$. Since the vertices involved in the update (the vertex whose revenue is changed or both of the endpoints of an updated edge) are now in Z, this update can be passed verbatim to $I^\star$. It is now easy to verify that all the stated invariants are preserved by the update.

We invoke Claim 5.3.10 at most $\mathcal{O}(w \log n)$ times, so the total number of updates performed on $I^\star$ is $(w \log n) \cdot w^{\mathcal{O}(1)} = w^{\mathcal{O}(1)} \log n$. Also, the total update time is $\Delta^{\mathcal{O}(w)} \log n$. This satisfies the required property (d) of the described data structure and concludes the proof. □

5.3.4 Full algorithm

We now describe the implementation of the data structure in detail. Recall that our aim is to maintain an n-vertex dynamic instance $I^{\mathrm{main}} = (G^{\mathrm{main}}, D^{\mathrm{main}}, \mathsf{rev}^{\mathrm{main}}, C^{\mathrm{main}})$ of 2CSP in a data structure, updated by $\mathsf{AddEdge}$, $\mathsf{RemoveEdge}$, and $\mathsf{UpdateRevenue}$, that can be queried for an approximate optimum revenue in the current snapshot of the instance: For a parameter $\varepsilon > 0$ fixed at the initialization, the data structure should, when queried, return a nonnegative real p such that $(1-\varepsilon)\mathsf{OPT} \leqslant p \leqslant \mathsf{OPT}$. The initialization of the data structure should take time $f(\varepsilon) \cdot n^{1+o(1)}$ and each update should take amortized time $f(\varepsilon) \cdot n^{o(1)}$, for some function f that is doubly-exponential in $\mathcal{O}(1/\varepsilon^2)$.

We fix an integer $L \in \mathbb{N}$, whose value will be determined later, and set $k := \lceil L/\varepsilon \rceil$. We construct a recursive, L-level data structure. That is, we will maintain a collection of auxiliary data structures, each maintaining an instance of 2CSP and an approximate optimum revenue to the maintained instance. Each auxiliary data structure will be assigned to one of the levels $1, \ldots, L$. At level L, we have a single auxiliary data structure $\mathbb{D}^{\mathrm{main}}$ maintaining I^{main}. Next, consider an auxiliary data structure $\mathbb{D}$ at level $q \in \{1, \ldots, L\}$, and assume $\mathbb{D}$ maintains an instance $I = (G, D, \mathsf{rev}, C)$ of 2CSP. If $q \geqslant 2$, then $\mathbb{D}$ maintains a collection of k data structures $\mathbb{D}_0, \ldots, \mathbb{D}_{k-1}$ at level $q-1$, called *children* of $\mathbb{D}$, with each child maintaining an instance derived from I. If $q \leqslant L-1$, then $\mathbb{D}$ is maintained by exactly one data structure at level $q+1$, called the *parent* of $\mathbb{D}$. Note that this way, the entire collection of auxiliary data structures forms a rooted tree of height L and branching k, where the root is $\mathbb{D}^{\mathrm{main}}$ and the leaves are the data structures at level 1. In particular, the total number of maintained data structures is $\mathcal{O}(k^L)$.

Moreover, each auxiliary data structure $\mathbb{D}$ at level q preserves the following invariants:

(I1) $G \in \mathcal{C}$ and $|V(G)| \leqslant n^{q/L}$;

(I2) for each $v \in V(G)$, we have $|D_v| \leqslant g(q)$ for some function g to be specified in Section 5.3.5;

(I3) $\mathbb{D}$ maintains a nonnegative real p satisfying $(1 - \varepsilon \cdot \frac{q}{L})\mathsf{OPT} \leqslant p \leqslant \mathsf{OPT}$, where OPT is the maximum revenue of a solution to I.

Note that (I1) and (I2) are satisfied by $\mathbb{D}^{\mathrm{main}}$ with $g(L) = 2$. By (I3), upon query $\mathsf{QueryMWIS}()$, the data structure can just return the real p stored in $\mathbb{D}^{\mathrm{main}}$.

Now, we explain the implementation of each data structure. First, let $\mathbb{D}$ be a data structure at level 1 maintaining an instance I of 2CSP. On each update to I, we update I by brute force: We recompute the approximate solution from scratch using Lemma 5.3.3 with ε' supplied to it equal to $\frac{\varepsilon}{L}$. Thus, processing each update to I takes time $|V(I)| \cdot g(1)^{\mathcal{O}(L/\varepsilon)} \leqslant n^{1/L} \cdot g(1)^{\mathcal{O}(L/\varepsilon)}$. Thus each level-1 data structure satisfies invariant (I3).

From that point on, let us fix some $2 \leqslant q \leqslant L$ and describe the implementation of a data structure $\mathbb{D}$ of level q, which maintains a Max Weight Nullary 2CSP instance $I = (G, D, \mathsf{rev}, C)$, assuming invariants (I1) and (I2). Let $I^{\mathrm{cur}} = (G^{\mathrm{cur}}, D^{\mathrm{cur}}, \mathsf{rev}^{\mathrm{cur}}, C^{\mathrm{cur}})$ denote the current snapshot of I. The lifetime of $\mathbb{D}$ is partitioned into *epochs*: sequences of τ_q updates to I, with τ_q to be specified later. The first epoch begins when $\mathbb{D}$ is initialized; and a new epoch begins each time $\mathbb{D}$ processes τ_q updates to I. At the start of each epoch, we let $I^{\mathrm{old}} := I^{\mathrm{cur}}$ and we apply the Baker's scheme to I^{old}. That is, let $I^{\mathrm{old}} = (G^{\mathrm{old}}, D^{\mathrm{old}}, \mathsf{rev}^{\mathrm{old}}, C^{\mathrm{old}})$. We produce a partition of $V(I^{\mathrm{old}})$ into layers $V_0, \ldots, V_{k-1}$ as follows. Assume that G^{old} is connected; otherwise apply the scheme to each connected component of G^{old}, and let the ith layer V_i be the union of the ith layers for each connected component of G^{old}. Now choose an arbitrary vertex s as a root, run the breadth-first search (BFS) on G^{old} from s, partition the graph into the BFS layers, and group them based on modulo k, getting sets $V_i = \{v \mid \mathsf{dist}_{G^{\mathrm{old}}}(s, v) \equiv i \mod k\}$ for $i = 0, 1, \ldots, k-1$.

Now, given sets $V_0, \ldots, V_{k-1}$, we define k dynamic instances of 2CSP: $I_i^{\mathrm{cur}} = I^{\mathrm{cur}} \setminus V_i$, called *universes*. At the start of the epoch, $I_i^{\mathrm{cur}} = I^{\mathrm{old}} \setminus V_i$. Note that by Lemma 5.3.2, we have that $\mathrm{tw}(G_i^{\mathrm{old}}) \leqslant \mathcal{O}(k)$. Thus, for each $0 \leqslant i \leqslant k-1$, we can instantiate a data structure of Lemma 5.3.8 on each I_i^{cur}, with $w = \mathcal{O}(k) = \mathcal{O}(L/\varepsilon)$ and $\Delta = g(q)$. Since $|V(I^{\mathrm{old}})| \leqslant n^{q/L}$, the initialization of each structure takes time $g(q)^{\mathcal{O}(L/\varepsilon)} \cdot n^{q/L} \log n$. After the initialization, the ith data structure maintains a compressed instance $I_i^\star$ of 2CSP with the same revenue of the optimum solution as I_i^{cur}, each initially containing one vertex. Therefore, to finish the initialization, we recursively spawn a collection of k children auxiliary data structures $\mathbb{D}_0, \ldots, \mathbb{D}_{k-1}$ at level $q-1$. Each $\mathbb{D}_i$ stores the compressed instance $I_i^\star$ and maintains a $(1 - \varepsilon\frac{q-1}{L})$-approximate optimum revenue to $I_i^\star$ (which is also the $(1 - \varepsilon\frac{q-1}{L})$-approximate optimum revenue to I_i^{cur}). Note that each of the children data structures recursively spawn k additional children data structures at level $q-2$, etc., until constructing k^{q-1} data structures at level 1 in total. Each of these data structures is initialized with a one-vertex 2CSP instance.

Now, assuming we can maintain a good approximation p_i of the maximum-revenue solution to each I_i^{cur}, we can also maintain such a fairly good approximation to I by simply keeping the maximum p_i:

Lemma 5.3.11. *Let OPT_i be the optimum revenue of a solution in the instance I_i^{cur}, and let p_i be such that $(1 - \varepsilon\frac{q-1}{L})\mathsf{OPT}_i \leqslant p_i \leqslant \mathsf{OPT}_i$. Let $p = \max(p_0, \ldots, p_{k-1})$. Then $(1 - \varepsilon\frac{q}{L})\mathsf{OPT} \leqslant p \leqslant \mathsf{OPT}$, where OPT is the revenue of the optimum solution to I^{cur}.*

Proof. Recall that G^{old} is the Gaifman graph of I^{old} and G^{cur} is the Gaifman graph of I^{cur}. Also, let A be the set of vertices that were added to I^{cur} since the beginning of the current epoch. Then, we have that $V(G^{\mathrm{cur}}) = V(G^{\mathrm{old}}) \cup A$. Let G_i^{cur} be the Gaifman graph of the universe I_i^{cur}. Then $V_i \subseteq V(G^{\mathrm{cur}})$ and $V(G_i^{\mathrm{cur}}) = V(G^{\mathrm{cur}}) \setminus V_i$. Let $\phi, \phi_0, \ldots, \phi_{k-1}$ be optimum solutions to the instances $I^{\mathrm{cur}}, I_0^{\mathrm{cur}}, \ldots, I_{k-1}^{\mathrm{cur}}$. As $V_0, \ldots, V_{k-1}$ are disjoint, we have that $\mathsf{rev}(\phi|_{V_0}) + \ldots + \mathsf{rev}(\phi|_{V_{k-1}}) \leqslant \mathsf{rev}(\phi)$. Therefore, there exists $0 \leqslant i \leqslant k-1$ such that $\mathsf{rev}(\phi|_{V_i}) \leqslant \frac{\mathsf{rev}(\phi)}{k} \leqslant \mathsf{rev}(\phi) \cdot \frac{\varepsilon}{L}$.

As $\phi|_{V(G_i^{\mathrm{cur}})}$ is a valid solution to I_i^{cur}, we have that $\mathsf{rev}(\phi|_{V(G_i^{\mathrm{cur}})}) \leqslant \mathsf{rev}(\phi_i) = \mathsf{OPT}_i$. In turn, we have $(1 - \frac{\varepsilon}{L})\mathsf{rev}(\phi) \leqslant \mathsf{rev}(\phi) - \mathsf{rev}(\phi|_{V_i}) = \mathsf{rev}(\phi|_{V(G^{\mathrm{cur}}) \setminus V_i}) = \mathsf{rev}(\phi|_{V(G_i^{\mathrm{cur}})}) \leqslant \mathsf{OPT}_i$. By multiplying both sides by $1 - \varepsilon\frac{q-1}{L}$, we get $(1 - \frac{\varepsilon}{L})(1 - \varepsilon\frac{q-1}{L})\mathsf{OPT} \leqslant (1 - \varepsilon\frac{q-1}{L})\mathsf{OPT}_i \leqslant p_i \leqslant p$, which implies that $p \geqslant \mathsf{OPT}(1 - \frac{\varepsilon}{L})(1 - \varepsilon\frac{q-1}{L}) \geqslant \mathsf{OPT}(1 - \varepsilon\frac{q}{L})$, as required. On the other hand, as any ϕ_j can be extended to the solution of the full instance by putting zeros on V_j, we obviously get $p_j \leqslant \mathsf{OPT}_j \leqslant \mathsf{OPT}$ for each $j = 0, \ldots, k-1$, which in turn implies that $p \leqslant \mathsf{OPT}$, as desired. □

Each update to I^{cur} is processed by $\mathbb{D}$ as follows: For each $i \in \{0, \ldots, k-1\}$, if the update involves a vertex of V_i, then I_i^{cur} remains unchanged by the update and no further action is required. Otherwise, the update is relayed to I_i^{cur}. The data structure from Lemma 5.3.8 produces a sequence of at most $k^{\mathcal{O}(1)} \log n$ updates to $I_i^\star$, which are then relayed to $\mathbb{D}_i$. After all children data structures process the updates, $\mathbb{D}$ recomputes its approximate solution to I^{cur} by querying each $\mathbb{D}_i$ for the approximation of the maximum revenue of a solution to $I_i^\star$ and returning the maximum value. Note that during one epoch, the size of each $I_i^\star$ does not grow above $\tau_q \cdot k^{\mathcal{O}(1)} \log n$ (Lemma 5.3.8(d)). By choosing τ_q so that this value is significantly less than $n^{(q-1)/L}$, we ensure the satisfaction of invariant (I1) by the children data structures (the fact that the Gaifman graph of $I_i^\star$ belongs to $\mathcal{C}$ follows from Lemma 5.3.8(a)). The invariant (I2) is satisfied due to Lemma 5.3.8(c), provided we set g so that $g(q-1) \geqslant g(q)^{\mathcal{O}(k)}$. Also, since the revenue of an optimum solution to $I_i^\star$ is equal to that of I_i^{cur} (Lemma 5.3.8(b)), each child data structure $\mathbb{D}_i$ maintains a nonnegative real p_i satisfying the preconditions of Lemma 5.3.11. Therefore, the value $p := \max(p_0, \ldots, p_{k-1})$ is an $(1 - \varepsilon\frac{q}{L})$-approximation of the optimum revenue in I^{cur}, which proves the satisfaction of invariant (I3) by $\mathbb{D}$.

Finally, when an epoch in $\mathbb{D}$ ends, $\mathbb{D}$ is reinitialized with $I^{\text{old}} := I^{\text{cur}}$ and a new epoch starts. We remark that this causes the destruction and the recursive reinitialization of the children data structures $\mathbb{D}_i$.

5.3.5 Setting the parameters and time complexity analysis

In this section, we are going to discuss the parameters whose specification was postponed: L, function g, and epoch lengths. Finally, we analyze the amortized time complexity of the updates.

At first, we are going to bound the domain sizes.

Lemma 5.3.12. *One can set function g so that invariant (I2) is satisfied for all the constructed data structures and $g(q) \in 2^{k^{\mathcal{O}(L)}}$ for all $q \in \{1, \ldots, L\}$.*

Proof. Recall that invariant (I2) states that the domain sizes in instances stored by data structures at level q are bounded by $g(q)$. For $q = L$ it suffices to set $g(L) = 2$. Next, for an instance within some data structure at level $q > 1$ we create some number of universes, and instances created for those universes have the same domains as the original one, say of size at most Δ. However, when compressing an instance and recursing to the deeper level, the domain sizes increase to at most Δ^{Ck} for some constant C. Indeed, recall that for a compressed vertex S we had $D'_S = \{0\} \cup \prod_{v \in S} D_v$, and due to the structure of F and Z we know that such S is always of size $\mathcal{O}(k)$. It follows that we may set $g(q-1) = g(q)^{Ck}$. A straightforward induction now shows that $g(q) = 2^{(Ck)^{L-q}}$, so in particular $g(q) \leqslant 2^{k^{\mathcal{O}(L)}}$ for all $q \in \{1, \ldots, L\}$. □

We know that each instance at level q spawns k universes and each universe spawns one instance at level $q-1$; unless $q = 1$, in which case the instance in question is a leaf. Hence, there are at most k^L instances at level 1. We can proceed further with this reasoning to show the following.

Lemma 5.3.13. *Each update to $\mathbb{D}^{\text{main}}$ causes at most $(k \log n)^{\mathcal{O}(L)}$ updates throughout all data structures.*

Proof. Recall that by Lemma 5.3.8(d), each update to a universe at level $q > 1$ causes at most $(k \log n)^{\mathcal{O}(1)}$ updates propagated to instances at level $q-1$. As each instance at level $q > 1$ has k universes associated with it, each update to an instance at level q causes at most $k \cdot (k \log n)^{\mathcal{O}(1)} = (k \log n)^{\mathcal{O}(1)}$ updates propagated to level $q-1$. Hence, there are at most $(k \log n)^{\mathcal{O}(L)}$ updates throughout all data structures per update to $\mathbb{D}^{\text{main}}$. □

Lemma 5.3.14. *The total time of updating all data structures for a single update to $\mathbb{D}^{\text{main}}$, excluding the time of all reinitializations, can be bounded by $n^{\frac{1}{L}} \cdot 2^{(\frac{L}{\varepsilon})^{\mathcal{O}(L)} + \mathcal{O}(L \log \log n)}$.*

Proof. Let us focus first on the time required to recompute the solutions to the instances at level 1 as explained in Lemma 5.3.3, where ε', n and Δ from the statement of this lemma are equal to $\frac{\varepsilon}{L}, n^{\frac{1}{L}}$ and $g(1) \in 2^{k^{\mathcal{O}(L)}}$, respectively. Each such recomputation uses time $n^{\frac{1}{L}} \cdot \left(2^{k^{\mathcal{O}(L)}}\right)^{\mathcal{O}(\lceil \frac{L}{\varepsilon} \rceil)} = n^{\frac{1}{L}} \cdot 2^{k^{\mathcal{O}(L)} \cdot \lceil \frac{L}{\varepsilon} \rceil}$. As there are at most $(k \log n)^{\mathcal{O}(L)}$ updates by Lemma 5.3.13, the total time used for all such recomputations can be bounded by $(k \log n)^{\mathcal{O}(L)} \cdot n^{\frac{1}{L}} \cdot 2^{k^{\mathcal{O}(L)} \cdot \lceil \frac{L}{\varepsilon} \rceil}$.

All overheads coming from processing a single update to some $\mathbb{D}$ like indexing the tables or navigating in F are of the form $(k \log n)^{\mathcal{O}(1)}$. Hence, based on that and Lemma 5.3.13, excluding the time required for all hypothetical reinitializations, the total time needed for updating all the necessary information is $(k \log n)^{\mathcal{O}(L)} + (k \log n)^{\mathcal{O}(L)} \cdot n^{\frac{1}{L}} \cdot 2^{k^{\mathcal{O}(L)} \cdot \lceil \frac{L}{\varepsilon} \rceil}$. Recall that we actually set $k = \lceil \frac{L}{\varepsilon} \rceil$, so we can do the following simplifications: $(k \log n)^{\mathcal{O}(L)} = (\lceil \frac{L}{\varepsilon} \rceil \log n)^{\mathcal{O}(L)} = 2^{\mathcal{O}(L(\log \log n + \log \frac{L}{\varepsilon}))}$, hence $(k \log n)^{\mathcal{O}(L)} \cdot n^{\frac{1}{L}} \cdot 2^{k^{\mathcal{O}(L)} \cdot \lceil \frac{L}{\varepsilon} \rceil} = 2^{\mathcal{O}(L(\log \log n + \log \frac{L}{\varepsilon}))} \cdot n^{\frac{1}{L}} \cdot 2^{(\frac{L}{\varepsilon})^{\mathcal{O}(L)}} = n^{\frac{1}{L}} \cdot 2^{(\frac{L}{\varepsilon})^{\mathcal{O}(L)} + \mathcal{O}(L \log \log n)}$. □

As the next step, we are going to set epoch lengths and bound the amortized time of all reinitializations per single update to $\mathbb{D}^{\text{main}}$. Let τ_q denote the epoch length for data structures on level q for some $2 \leqslant q \leqslant L$. Recall that the epoch length for a data structure is measured in the number of updates to this particular instance (as opposed to $\mathbb{D}^{\text{main}}$). Also recall Item (d) from Lemma 5.3.8, which asserts that each update to $\mathbb{D}$ at level q generates $(k \log n)^{\mathcal{O}(1)}$ updates to its children structures. Let us be more specific and let c be such a constant that this number is at most $(k \log n)^c$. Then, we set $\tau_q = n^{\frac{q-1}{L}} / (k \log n)^c$. For such a choice, it is indeed the case that the children data structures are passed at most $n^{\frac{q-1}{L}}$ updates before they are rebuilt, hence we maintain invariant (I1): instances at level $q-1$ are of size at most $n^{\frac{q-1}{L}}$ at all times.

Thus, we can bound the amortized time complexity of initializations and reinitializations.

Lemma 5.3.15. *The amortized time of all initializations and reinitializations per single update to* $\mathbb{D}^{\mathrm{main}}$ *is* $n^{\frac{1}{L}} \cdot 2^{(\frac{L}{\varepsilon})^{\mathcal{O}(L)}+\mathcal{O}(L \log\log n)}$.

Proof. We view that a lifespan of a particular data structure $\mathbb{D}$ corresponds to one epoch of the parent data structure (unless it is $\mathbb{D}^{\mathrm{main}}$). If the parent data structure of $\mathbb{D}$ gets rebuilt, we trash $\mathbb{D}$. Let t be the number of started epochs of $\mathbb{D}$ throughout its whole lifetime. We note that when it is initialized, it is of a form $I\{\emptyset\}$ and consists of a single vertex (unless it is the initialization of $\mathbb{D}^{\mathrm{main}}$, which takes $n \cdot 2^{\mathcal{O}(k)}$ time). Hence, the first initialization takes constant time. As guaranteed by Lemma 5.3.8 and Lemma 5.3.12, each reinitialization of $\mathbb{D} = (G, D, \mathsf{rev}, C)$ takes $(|V(G)| + |E(G)|) \log |V(G)| \cdot g(q)^{\mathcal{O}(k)} = (|V(G)| + |E(G)|) \log |V(G)| \cdot 2^{k^{\mathcal{O}(L)}}$ time. Thanks to our invariants, we are guaranteed that $|V(G)| \leqslant n^{\frac{q}{L}}$ and $|E(G)| = \mathcal{O}(|V(G)|)$, hence the reinitialization time can be bounded as $n^{\frac{q}{L}} \log n \cdot 2^{k^{\mathcal{O}(L)}}$.

Note that if t epochs were started, then the lifetime of $\mathbb{D}$ contained $t-1$ full epochs. The number of reinitializations will also be equal to $t-1$. Hence, if u denotes the number of updates to $\mathbb{D}$ so far, then the time required for all reinitializations of $\mathbb{D}$ can be bounded as $(t-1) \cdot n^{\frac{q}{L}} \log n \cdot 2^{k^{\mathcal{O}(L)}} \leqslant (t-1) \cdot \tau_q \cdot n^{\frac{1}{L}} \cdot (k \log n)^{c+1} \cdot 2^{k^{\mathcal{O}(L)}} \leqslant u \cdot n^{\frac{1}{L}} \cdot (\log n)^{\mathcal{O}(1)} \cdot 2^{k^{\mathcal{O}(L)}}$. Hence, the amortized time required for all reinitializations of $\mathbb{D}$ can be bounded as $n^{\frac{1}{L}} \cdot (\log n)^{\mathcal{O}(1)} \cdot 2^{k^{\mathcal{O}(L)}}$ per an update to $\mathbb{D}$. As each update to $\mathbb{D}^{\mathrm{main}}$ causes $(k \log n)^{\mathcal{O}(L)}$ updates to all structures in total and there are at most k^L data structures, the total amortized time required for all reinitializations per a single update to $\mathbb{D}^{\mathrm{main}}$ is $n^{\frac{1}{L}} \cdot (\log n)^{\mathcal{O}(L)} \cdot 2^{k^{\mathcal{O}(L)}} \cdot k^{\mathcal{O}(L)} = n^{\frac{1}{L}} \cdot (\log n)^{\mathcal{O}(L)} \cdot 2^{(\frac{L}{\varepsilon})^{\mathcal{O}(L)}} = n^{\frac{1}{L}} \cdot 2^{(\frac{L}{\varepsilon})^{\mathcal{O}(L)}+\mathcal{O}(L \log\log n)}$. □

With these lemmas, we can present the final complexity analysis:

Lemma 5.3.16. *L may be set so that the amortized time of an update to* $\mathbb{D}^{\mathrm{main}}$ *is* $n^{\mathcal{O}(\frac{\log\log\log n}{\log\log n \cdot \varepsilon})} = n^{o(\frac{1}{\varepsilon})}$.

Proof. As both Lemmas 5.3.14 and 5.3.15 guarantee $n^{\frac{1}{L}} \cdot 2^{(\frac{L}{\varepsilon})^{\mathcal{O}(L)}+\mathcal{O}(L \log\log n)}$ time for the corresponding computations, we get that the amortized time of an update to $\mathbb{D}^{\mathrm{main}}$ is equal to $n^{\frac{1}{L}} \cdot 2^{(\frac{L}{\varepsilon})^{\mathcal{O}(L)}+\mathcal{O}(L \log\log n)}$ as well.

Let us set $L = \frac{\log\log n}{\log\log\log n} \cdot \varepsilon \cdot \delta$ for some absolute constant $\delta \in (0,1)$ independent on ε. For such a choice we have that $\mathcal{O}(L \log\log n) \leqslant \left(\frac{L}{\varepsilon}\right)^{\mathcal{O}(L)}$, so the final complexity can be bounded by

$$\begin{aligned} n^{\frac{1}{L}} \cdot 2^{\left(\frac{L}{\varepsilon}\right)^{\mathcal{O}(L)}} &= n^{\frac{\log\log\log n}{\log\log n \cdot \varepsilon \cdot \delta}} \cdot 2^{\left(\frac{\log\log n}{\log\log\log n} \cdot \delta\right)^{\mathcal{O}\left(\frac{\log\log n}{\log\log\log n} \cdot \varepsilon \cdot \delta\right)}} \\ &= 2^{\frac{\log n \cdot \log\log\log n}{\log\log n \cdot \varepsilon \cdot \delta}} \cdot 2^{2^{\mathcal{O}(\log\log n \cdot \varepsilon \cdot \delta)}} \\ &= 2^{\frac{\log n \cdot \log\log\log n}{\log\log n \cdot \varepsilon \cdot \delta}} \cdot 2^{(\log n)^{\mathcal{O}(\varepsilon \cdot \delta)}}. \end{aligned}$$

If δ is sufficiently small, then $\mathcal{O}(\varepsilon \cdot \delta)$ can be bounded from above by $\frac{1}{2}$. Then, the second factor will be dominated by the first one and the final complexity is then bounded by $n^{\mathcal{O}(\frac{\log\log\log n}{\log\log n \cdot \varepsilon})} = n^{o(\frac{1}{\varepsilon})}$. □

However, the time complexity of $f(\varepsilon) \cdot n^{o(1)}$ would be preferable over $n^{o(\frac{1}{\varepsilon})}$, so as the last step, let us explain how to arrive at it. We will distinguish two cases, based on whether $n > 2^{2^{\frac{1}{\varepsilon^2}}}$.

Case 1: $n > 2^{2^{\frac{1}{\varepsilon^2}}}$.

Then we have $\log\log n > \frac{1}{\varepsilon^2} \Rightarrow \varepsilon > \sqrt{\log\log n}$, so $n^{\mathcal{O}(\frac{\log\log\log n}{\log\log n \cdot \varepsilon})} = n^{\mathcal{O}\left(\frac{\log\log\log n}{\sqrt{\log\log n}}\right)} = n^{o(1)}$.

Case 2: $n \leqslant 2^{2^{\frac{1}{\varepsilon^2}}}$.

In this case, instead of using our final algorithm, after every single update we can simply use a brute-force method provided by Lemma 5.3.3 and approximately solve the instance in time $n \cdot 2^{\mathcal{O}(\frac{1}{\varepsilon})} = 2^{2^{\mathcal{O}(\frac{1}{\varepsilon^2})}}$.

In any case, the time complexity of an update can be bounded as $2^{2^{\mathcal{O}(\frac{1}{\varepsilon^2})}} \cdot n^{\mathcal{O}\left(\frac{\log\log\log n}{\sqrt{\log\log n}}\right)} = f(\varepsilon) \cdot n^{o(1)}$. Then, the initialization of the data structure on an n-vertex graph $G \in \mathcal{C}$ can be performed in time $n^{1+o(1)}$: We create an instance of the data structure for an edgeless n-vertex graph and add edges to it one by one. Since $|E(G)| \leqslant \mathcal{O}(n)$ for any $G \in \mathcal{C}$, the bound on the initialization time follows. This concludes the proof of Theorem 1.3.7 in the setting of Maximum Weight Independent Set.

5.4 Minimum Weight Dominating Set

In this section, we describe a dynamic approximation scheme for Minimum Weight Dominating Set. As in the case of the Maximum Weight Independent Set, we assume that a given apex-minor-free class of graphs $\mathcal{C}$ is fixed. Moreover, throughout this section we denote by $\varepsilon > 0$ the parameter ε fixed in the initialization of the data structure.

Consider the dynamic setting of Min Weight Generalized Domination. We assume that a data structure for Generalized Domination is initialized with integers $s, d \geqslant 1$ and a dynamic (s,d)-decent instance of Min Weight Generalized Domination. We consider the following types of updates to instances of Min Weight Generalized Domination:

- $\mathsf{AddVertex}(u, D_u, \mathsf{cost}_u)$: Adds an isolated vertex, say u, to the instance, together with a domain D_u of size at most d and a cost function $\mathsf{cost}_u : D_u \to \mathbb{R}_{\geqslant 0} \cup \{+\infty\}$. For each state $s \in D_u$, we initialize $\mathsf{supply}_u(s) = \emptyset$ and $\mathsf{demand}_u(s) = \emptyset$.
- $\mathsf{AddEdge}(u, v, D_u^{\mathsf{supply}}, D_u^{\mathsf{demand}}, D_v^{\mathsf{supply}}, D_v^{\mathsf{demand}})$: Adds an edge e with endpoints u and v; for each endpoint $w \in \{u, v\}$, the edge e is added to each set $\mathsf{supply}_w(x)$ for $x \in D_w^{\mathsf{supply}}$ and to each set $\mathsf{demand}_w(x)$ for $x \in D_w^{\mathsf{demand}}$.
- $\mathsf{RemoveEdge}(e)$: Removes an edge e from the graph, removing it also from the corresponding demand and supply sets.
- $\mathsf{UpdateCost}(u, \mathsf{cost}'_u)$: Replaces cost_u for a vertex $u \in V(G)$ with a new function $\mathsf{cost}'_u : D_u \to \mathbb{R}_{\geqslant 0} \cup \{+\infty\}$.

As in the case with the independent set, we do not support vertex removals.

We now adapt the statement of Theorem 1.3.7 to the language of instances of Min Weight Generalized Domination.

Lemma 5.4.1. *Let $s, d \in \mathbb{N}$ be absolute constants and choose $\delta > 0$. Then there exists a data structure storing a dynamic (s,d)-decent instance $I = (G, D, \mathsf{cost}, \mathsf{supply}, \mathsf{demand})$ of* Min Weight Generalized Domination *under the assumption that $G \in \mathcal{C}$ holds at every point of time. The data structure maintains a nonnegative real p satisfying $(1-\delta)\mathsf{OPT} \leqslant p \leqslant \mathsf{OPT}$, where OPT is the minimum possible cost of a solution to I. The data structure is initialized in time $f(\delta) \cdot n^{1+o(1)}$, and each update takes time $f(\delta) \cdot n^{o(1)}$, where $f(\delta)$ is doubly-exponential in $\mathcal{O}(1/\delta^2)$.*

Counterintuitively, instead of trying approximate Min Weight Generalized Domination by finding an approximate solution that would have a slightly higher cost than optimal, the data structure of Lemma 5.4.1 will maintain a good *lower bound* on the minimum possible cost of an instance. This is, however, still enough to show that the query $\mathsf{QueryMWDS}$ of Theorem 1.3.7 can be answered correctly: Assume, for some absolute constant $\Delta \in \mathbb{N}$ that we are given a dynamic graph G with a bound Δ on the maximum degree of a vertex in G and $\varepsilon > 0$, and we wish to maintain a nonnegative real p' such that $\mathsf{OPT} \leqslant p' \leqslant (1+\varepsilon)\mathsf{OPT}$, where OPT is the minimum weight of a dominating set of G. Set $\delta := \frac{\varepsilon}{1+\varepsilon}$ and instantiate the data structure of Lemma 5.4.1, initializing it with the $(\Delta, \Delta+1)$-decent instance I of Min Weight Generalized Domination constructed from G and the parameter δ. Each update to G is translated to the appropriate update to I and forwarded to the constructed data structure. Thus, the data structure is created in time $f(\delta) \cdot n^{1+o(1)}$ and each update to G takes time $f(\delta) \cdot n^{o(1)}$. Since $1/\delta = 1 + 1/\varepsilon$ and f is doubly-exponential in $\mathcal{O}(1/\delta^2)$, f is also doubly-exponential in $\mathcal{O}(1/\varepsilon^2)$.

The data structure from Lemma 5.4.1 maintains a nonnegative real p such that $(1-\delta)\mathsf{OPT} \leqslant p \leqslant \mathsf{OPT}$. Setting $p' := \frac{p}{1-\delta}$, we obtain that $\mathsf{OPT} \leqslant p' \leqslant (1-\delta)^{-1}\mathsf{OPT} = (1+\varepsilon)\mathsf{OPT}$. Thus our implementation can return p' as the over-approximation of the minimum weight of a dominating set of G that is at most a factor of ε away from the optimum weight.

5.4.1 Static variant of Generalized Domination

Here, we show how to apply the Baker's technique to decent instances of Min Weight Generalized Domination.

Lemma 5.4.2. *Let $s, d, w \geqslant 1$. Given on input an (s,d)-decent instance of* Min Weight Generalized Domination *$I = (G, D, \mathsf{cost}, \mathsf{supply}, \mathsf{demand})$ with $n \geqslant 1$ vertices with the property that $\mathrm{tw}(G) \leqslant w$, we can compute the minimum cost of a solution to I in worst-case time $\mathcal{O}(ns) \cdot d^{\mathcal{O}(w)}$.*

Proof of Lemma 5.4.2. Compute a tree decomposition T of G of width $\mathcal{O}(w)$ in time $n \cdot 2^{\mathcal{O}(w)}$, e.g., using the algorithm of Korhonen [Kor21]. Then the problem can be modeled by a straightforward dynamic programming scheme on tree decompositions; here, we only give the states of the dynamic programming process.

For every node $t \in V(T)$ of the tree decomposition with a bag $\mathsf{bag}(t) \subseteq V(G)$, let $X_t \coloneqq \prod_{v \in \mathsf{bag}(t)} D_v$. For each $t \in V(T)$ and $x \in X_t$, we compute the value $P[t][x]$: the minimum cost of a locally correct valuation ϕ of the vertices of G in the subtree of t in T with the property that for every $v \in \mathsf{bag}(t)$, we have that $\phi(v) = x(v)$.

For each node $t \in V(T)$, there exist at most $d^{|\mathsf{bag}(t)|} \leqslant d^{\mathcal{O}(w)}$ different dynamic programming states. It is then straightforward to implement the entire dynamic programming scheme in time $d^{\mathcal{O}(w)} \cdot \mathcal{O}(sn)$; here, the additional factor $\mathcal{O}(s)$ in the time complexity comes from the need of the verification of the demands and the supplies for each edge of the graph. □

Lemma 5.4.3. *Let $s, d, k \geqslant 1$. Consider an (s,d)-decent instance of* Min Weight Generalized Domination *$I = (G, D, \mathsf{cost}, \mathsf{supply}, \mathsf{demand})$ of optimum cost* OPT*. Let $V_0, \ldots, V_{k-1} \subseteq V(G)$ be so that the sets $N_G[V_0], N_G[V_1], \ldots, N_G[V_{k-1}]$ are pairwise disjoint. For each $j \in \{0, \ldots, k-1\}$, consider the instance $I_j \coloneqq \mathsf{Clear}(I; V_j)$. Then:*

- *for every $j \in \{0, \ldots, k-1\}$, I_j is (s,d)-decent and has a solution of cost at most* OPT*;*
- *there exists $j \in \{0, \ldots, k-1\}$ such that the minimum-cost solution to I_j has cost at least $(1 - \frac{1}{k})$*OPT*.*

Proof. By the properties of V_j-cleared instances, we immediately have that each I_j has a solution of cost at most OPT. It remains to show that for some $j \in \{0, \ldots, k-1\}$, some subinstance I_j has the minimum cost of a solution lower-bounded by $(1 - \frac{1}{k})$OPT.

Let ϕ be some optimum solution to I. By the pigeonhole principle and the fact that the closed neighborhoods of all sets V_j are pairwise disjoint, there exists some $j \in \{0, \ldots, k-1\}$ such that $\mathsf{cost}(\phi|_{N[V_j]}) \leqslant \frac{1}{k}$OPT. We claim that OPT_j, the minimum cost of a solution of I_j, is at least $(1 - \frac{1}{k})$OPT.

Suppose we have a solution ψ_j to I_j of cost strictly less than $(1 - \frac{1}{k})$OPT. Recall that $V(I_j) = V(I)$ and that each $\psi_j(v)$ for each $v \in V(I)$ is also an element of the domain of v in I. With that in mind, construct a valuation ϕ' of $V(I)$ as follows:

- if $v \in V_j$, set $\phi'(v)$ to the combination of the state $\phi(v)$ with $\psi_j(v)$;
- if $v \in N(V_j)$, set $\phi'(v)$ to the combination of the state $\psi_j(v)$ with $\phi(v)$;
- otherwise, set $\phi'(v) \coloneqq \psi_j(v)$.

We now show that the construction of ϕ' contradicts our assumptions:

Claim 5.4.4. *ϕ' is a valid solution to I of cost strictly less than* OPT*.*

Proof of the claim. Suppose ϕ' is not a valid solution to I. Then there exists an edge e with endpoints u, v such that $e \in \mathsf{demand}_v(\phi'(v))$, but $e \notin \mathsf{supply}_u(\phi'(u))$. By the definition, the edge cannot have one endpoint in V_j and the other outside of $N[V_j]$. We now consider cases, depending on the location of u and v with respect to V_j:

- Assume $v \in V_j$. By the state-monotonicity of v, we have $\mathsf{demand}_v(\phi'(v)) \subseteq \mathsf{demand}_v(\phi(v))$, so $e \in \mathsf{demand}_v(\phi(v))$. Then also $u \in N[V_j]$. By the state-monotonicity of u, we have that $\mathsf{supply}_u(\phi'(u)) \supseteq \mathsf{supply}_u(\phi(u))$, so $e \notin \mathsf{supply}_u(\phi(u))$. This is, however, a contradiction: e witnesses that ϕ was not a valid solution to I in the first place.
- Assume $v \notin V_j$. Then $e \in \mathsf{demand}_v(\psi_j(v))$: either $v \in N(V_j)$ and then this follows from the state-monotonicity of v (so $\mathsf{demand}_v(\phi'(v)) \subseteq \mathsf{demand}_v(\psi_j(v))$), or $v \notin N[V_j]$ and then simply $\phi'(v) = \psi_j(v)$. Also, from the state-monotonicity of $u \in V(G)$ we have that $\mathsf{supply}_u(\phi'(u)) \supseteq \mathsf{supply}_u(\psi_j(u))$, so $e \notin \mathsf{demand}_u(\psi_j(u))$. Since $v \notin V_j$, we have $e \in E(I_j)$, so this yields a contradiction: e witnesses that ψ_j was not a valid solution to I_j in the first place.

Hence ϕ' is a correct solution to I. By the state-monotonicity of vertices of I, and by the facts that $\mathsf{cost}(\phi|_{N[V_j]}) \leqslant \frac{1}{k}$OPT and ψ_j has cost less than $(1 - \frac{1}{k})$OPT, we conclude that ϕ' has cost strictly smaller than OPT. ◁

Claim 5.4.4 yields a contradiction. Thus I_j has no solution of cost smaller than $(1 - \frac{1}{k})$OPT. □

Lemma 5.4.5. *Let $s, d \geqslant 1$, $\delta > 0$, and consider an (s,d)-decent instance of $I \in \mathcal{C}$ of* Min Weight Generalized Domination *with $n \geqslant 1$ vertices and optimum solution cost OPT. Then we can compute a nonnegative real p such that $(1-\delta)\mathsf{OPT} \leqslant p \leqslant \mathsf{OPT}$ in time $\mathcal{O}(ns) \cdot d^{\mathcal{O}(1/\delta)}$.*

Proof. Assume without loss of generality that $\delta \in (0,1)$, and set $k := \lceil \frac{1}{\delta} \rceil$. Assume that the instance I is connected; otherwise, solve for each connected component separately and return the sum of the results. Fix a vertex $r \in V(I)$ and create in time $\mathcal{O}(n)$ the partition $V(I) = A_0 \cup \cdots \cup A_{4k-1}$, where A_i contains vertices v such that $\mathsf{dist}(r,v) \equiv i \mod 4k$. Then, for each $j \in \{0, \ldots, k-1\}$, let $V_j = A_{4j+1} \cup A_{4j+2}$ and $I_j = \mathsf{Clear}(I; V_j)$. Note that $N[V_j] \subseteq A_{4j} \cup A_{4j+1} \cup A_{4j+2} \cup A_{4j+3}$, so the closed neighborhoods of the sets V_j are pairwise vertex-disjoint and Lemma 5.4.3 applies. That is, if OPT is the minimum-cost solution to I, then each I_j is (s,d)-decent and has a solution of cost at most OPT, and there exists a subinstance with minimum cost of a solution at least $(1-\frac{1}{k})\mathsf{OPT} \geqslant (1-\delta)\mathsf{OPT}$.

Observe that for each $j \in \{0, \ldots, k-1\}$, the Gaifman graph of I_j has bounded treewidth: Consider a connected component C of the Gaifman graph. The component cannot simultaneously contain vertices of A_{4j+1} and A_{4j+2}, so the vertex set of C is contained within some $4k$ consecutive BFS layers. In other words, for some $\ell \geqslant 0$, we have that $C \subseteq \{v \in V(G) \mid \ell \leqslant \mathsf{dist}(s,v) \leqslant \ell + 4k - 1\}$. Thus by Lemma 5.3.2, $G[C]$ has treewidth at most $\mathcal{O}(k)$, so also the Gaifman graph of I_j has treewidth at most $\mathcal{O}(k) = \mathcal{O}(\frac{1}{\delta})$.

Let OPT be the minimum-cost solution to I. By Lemma 5.4.3, each I_j is (s,d)-decent, the treewidth of the Gaifman graph of I_j is at most $\mathcal{O}(\frac{1}{\delta})$, all instances have solutions of cost at most OPT, and there exists an instance with the minimum cost of a solution at least $(1-\frac{1}{k}\mathsf{OPT}) \geqslant (1-\delta)\mathsf{OPT}$. Now, we solve each instance I_j optimally using the algorithm from Lemma 5.4.2. Let p_j be the value returned by I_j and let $p = \max(p_0, \ldots, p_{k-1})$; then $p \in [(1-\delta)\mathsf{OPT}, \mathsf{OPT}]$, so p is as required. It can be easily verified that the algorithm runs in time $n \cdot s \cdot d^{\mathcal{O}(1/\delta)}$. □

5.4.2 Compression for Generalized Domination

We proceed to show how to translate the process of instance compression to the setting of Min Weight Generalized Domination.

We begin by presenting how to encode interactions between a set of vertices and its neighborhood in a concise way. Consider an instance $I = (G, D, \mathsf{cost}, \mathsf{supply}, \mathsf{demand})$ and a set of vertices $R \subseteq V(I)$. Let also S be the neighborhood of R, i.e., $S = N_G(R)$. For every valuation $\phi \in \prod_{u \in R} D_u$ of R, we define the *interaction* of R with S:

$$\mathsf{interaction}_{R,S}(\phi) \in \left(2^{\{\mathsf{Supply},\mathsf{Demand}\}}\right)^{E(R,S)},$$

so that $\mathsf{interaction}_{R,S}(\phi)$ is a function x mapping each edge connecting R with S to a subset of $\{\mathsf{Supply}, \mathsf{Demand}\}$ as follows: Let $e \in \delta(S)$, $e = uv$, where $u \in R$ and $v \in S$. Then we have $\mathsf{Supply} \in x(e)$ if and only if $e \in \mathsf{supply}_u(\phi_u))$, and $\mathsf{Demand} \in x(e)$ if and only if $e \in \mathsf{demand}_u(\phi(u))$.

In other words, we record, for every edge e connecting a vertex $u \in R$ with a neighbor $v \in S$, whether u provides supply on e (so that in any solution extending ϕ to I, v can safely demand that supply) and whether u demands supply on e (so that in a solution extending ϕ to I, v has to provide that supply). Intuitively, $\mathsf{interaction}_{R,S}(\phi)$ stores the entire interaction of R with the neighborhood in a concise way. The following lemma formalizes this intuition:

Lemma 5.4.6. *Let $R \subseteq V(I)$ and $S = N(R)$. Assume ϕ is a correct solution to I and ψ is a locally correct solution on R so that $\mathsf{interaction}_{R,S}(\phi|_R) = \mathsf{interaction}_{R,S}(\psi)$. Then, the following valuation ϕ' of $V(I)$:*

$$\phi'(u) = \begin{cases} \psi(u) & \text{if } u \in R, \\ \phi(u) & \text{if } u \notin R \end{cases}$$

is also a correct solution to I.

Proof. Assume otherwise; in this case, there exists an edge $e \in E(G)$ between u and v so that $e \in \mathsf{demand}_v(\phi'(v))$, yet $e \notin \mathsf{supply}_u(\phi'(u))$. It cannot be that $u, v \in R$ since $\phi'|_R = \psi$ is locally correct on R; and also it cannot be that $u, v \notin R$ as $\phi'|_{V(I)\setminus R} = \phi|_{V(I)\setminus R}$ and ϕ is a correct solution to I. Hence, exactly one of the endpoints of e is in R.

If $u \in R$, then $v \in S$. Since $\phi'(u) = \psi(u)$, we have that $e \notin \mathsf{supply}_u(\psi(u))$ and therefore $\mathsf{Supply} \notin \mathsf{interaction}_{R,S}(\psi)(e)$. So also $\mathsf{Supply} \notin \mathsf{interaction}_{R,S}(\phi|_R)(e)$, implying $e \notin \mathsf{supply}_u(\phi(u))$. But $\phi'(v) = \phi(v)$, so $e \in \mathsf{demand}_v(\phi'(v))$ implies $e \in \mathsf{demand}_v(\phi(v))$. This is a contradiction since e witnesses that ϕ was not a correct solution to I in the first place.

The other case is that $v \in R$ and $u \in S$. From $\phi'(v) = \psi(v)$ we have $e \in \mathsf{demand}_v(\psi(v))$, so $\mathsf{Demand} \in \mathsf{interaction}_{R,S}(\psi)(e)$, so $\mathsf{Demand} \in \mathsf{interaction}_{R,S}(\phi|_R)(e)$ and $e \in \mathsf{demand}_v(\phi(v))$. Also, since $\phi'(u) = \phi(u)$, then we have $e \notin \mathsf{supply}_u(\phi(u))$ – a contradiction as e witnesses that ϕ was not a correct solution to I. □

We are now ready to give the definition of a compressed instance.

Definition 13. *Let $I = (G, D, \mathsf{cost}, \mathsf{supply}, \mathsf{demand})$ be an (s,d)-decent instance of* Min Weight Generalized Domination *and $Y \subseteq V(I)$. Then, we say that a* compressed instance *is an instance $I\{Y\} = (G', D', \mathsf{cost}', \mathsf{supply}', \mathsf{demand}')$ created from I by the following series of steps:*

1. *Enumerate all connected components $C_1, \ldots, C_\ell \in \mathrm{cc}(G \setminus Y)$. Assume that, for some $\ell_1 \in \{0, \ldots, \ell\}$, the neighborhood of each component $C_1, \ldots, C_{\ell_1}$ intersects Y, and the neighborhood of each component $C_{\ell_1+1}, \ldots, C_\ell$ is disjoint from Y (that is, each $C_{\ell_1+1}, \ldots, C_\ell$ is a connected component of G not adjacent to Y).*

2. *For each $i \in \{1, \ldots, \ell_1\}$, let $S_i = N(C_i) \subseteq Y$. Collapse C_i to a single vertex u_i with the domain*

$$D_{u_i} := \left(2^{\{\mathsf{Supply},\mathsf{Demand}\}}\right)^{E(C_i,S_i)}.$$

   For each $x \in D_{u_i}$, we set $\mathsf{cost}_{u_i}(x)$ to the minimum cost of a valuation $\phi \in \prod_{u \in C_i} D_u$, locally correct on C_i, so that $x = \mathsf{interaction}_{C_i,S_i}(\phi)$. If no such valuation exists, we set $\mathsf{cost}_{u_i}(x) = +\infty$.

   Finally, set the supply and the demand of any given state x as follows:

$$\mathsf{supply}'_{u_i}(x) = \{e \in E(C_i, S_i) \mid \mathsf{Supply} \in x(e)\},$$
$$\mathsf{demand}'_{u_i}(x) = \{e \in E(C_i, S_i) \mid \mathsf{Demand} \in x(e)\}.$$

3. *Collapse $R := C_{\ell_1+1} \cup \cdots \cup C_\ell$ to a single isolated vertex $u_\ominus$ with the one-state domain $D_{u_\ominus} = \{\ominus\}$; the cost $\mathsf{cost}_{u_\ominus}(\ominus)$ is defined as the minimum-cost valuation of R that is locally correct on R (note that a valuation locally correct on R with finite cost always exists).*

Intuitively, each component $C_i \in \mathrm{cc}(G \setminus Y)$ with nonempty neighborhood S on Y is collapsed into a single vertex u_i; note that, in contrast to the compression for Independent Set, different components with the same neighborhood S are collapsed to separate vertices. The collapsed vertex encodes in its state all possible interactions of C_i with S_i. Note that by Lemma 5.4.6, it is enough to record, for each possible interaction, the minimum-cost valuation of C_i with this interaction on S_i. On the other hand, the components not adjacent to Y do not interact with Y in any way, so the union R of these components can be replaced with a single vertex representing the minimum-cost locally correct valuation of R.

Observe that the construction above is correct in the sense that in $I\{Y\}$, every vertex u has a state s_u with finite cost and full supply $\mathsf{supply}_u(s_u) = \delta(u)$: For $u \in Y \cup \{\ominus\}$ this is obvious. Then, for a vertex u_i with $i \in \{1, \ldots, \ell_1\}$, there exists a finite-cost valuation ϕ_i, locally correct on C_i, mapping each vertex $u \in C_i$ to s_u. Moreover, for this valuation, we have $\mathsf{Supply} \in \mathsf{interaction}_{C_i,S_i}(\phi_i)(e)$ for all $e \in E(C_i, S_i)$.

We now prove a series of properties of compressed instances. We begin by proving that compression does not change the minimum cost of an instance:

Lemma 5.4.7. *For every instance I of* Min Weight Generalized Domination *and $Y \subseteq V(I)$, the instances I and $I\{Y\}$ have the same minimum cost of a solution.*

Proof. Let OPT be a minimum cost of a solution to $I = (G, D, \mathsf{cost}, \mathsf{supply}, \mathsf{demand})$ and OPT' be a minimum cost of a solution to $I\{Y\} = (G', D', \mathsf{cost}', \mathsf{supply}', \mathsf{demand}')$. Also let $\alpha : V(I) \to V(I\{Y\})$ be a function mapping vertices $v \in V(I)$ onto elements of $V(I\{Y\})$ to which v was collapsed during the compression. Note that $\alpha(v) = v$ for $v \in Y$.

Let ϕ be any minimum-cost solution to I. We construct a correct solution ϕ' to $I\{Y\}$ as follows.

- If $u \in Y$, then set $\phi'(u) := \phi(u)$.
- If $u \notin Y$ and $u \neq u_\ominus$, then let $C = \alpha^{-1}(u) \in \mathrm{cc}(G \setminus Y)$, $S = N_G(C) \subseteq Y$, and set $\phi'(u) := \mathsf{interaction}_{C,S}(\phi|_C)$.
- Set $\phi'(u_\ominus) = \ominus$.

The correctness of ϕ' follows easily from the definition. Note that for every $u \in Y$, we naturally have $\mathsf{cost}'_u(\psi(u)) = \mathsf{cost}_u(\phi(u))$. For $u \notin Y$ with $u \neq u_\ominus$, let $C = \alpha^{-1}(u) \in \mathrm{cc}(G \setminus Y)$ and $S = N_G(C)$. Since $\phi|_C$ is a locally correct valuation of C with interaction $\phi'(u)$ and $\mathsf{cost}'_u(\phi'(u))$ is the minimum cost of such a valuation, we have $\mathsf{cost}'_u(\psi(u)) \leqslant \mathsf{cost}_u(\phi|_C)$. By the same token, $\phi|_{\alpha^{-1}(u_\ominus)}$ is a locally correct valuation of $\alpha^{-1}(u_\ominus)$ and $\mathsf{cost}'_{u_\ominus}(\ominus)$ is the minimum cost of such a valuation; thus, $\mathsf{cost}'_{u_\ominus}(\ominus) \leqslant \mathsf{cost}_u(\phi|_{\alpha^{-1}(u_\ominus)})$. Hence the cost of ϕ' in $I\{Y\}$ is at most equal to the cost of ϕ in I. Thus $\mathsf{OPT}' \leqslant \mathsf{OPT}$.

Now let ψ' be any minimum-cost solution to $I\{Y\}$. Recover a valid solution ψ to I as follows.

- If $u \in Y$, then set $\psi(x) \coloneqq \psi'(x)$.
- If $u \notin Y$ and $u \neq u_\ominus$, then let $C = \alpha^{-1}(u) \in \mathrm{cc}(G \setminus Y)$ and $S = N(C) \subseteq Y$. Let $\zeta_C \in \prod_{v \in C} D_v$ be the minimum-cost locally correct valuation of C with $\mathsf{interaction}_{C,S}(\zeta_C) = \psi'(u)$, and set $\psi(x) \coloneqq \zeta_C(x)$ for $x \in C$.
- Similarly, if $u = u_\ominus$, then let $R = \alpha^{-1}(u_\ominus)$, and let $\zeta_R \in \prod_{v \in R}$ be the minimum-cost locally correct valuation of R; set $\psi(x) \coloneqq \zeta_R(x)$ for $x \in R$.

It can be verified through a straightforward case study that ϕ is a valid solution to I, and ϕ has the same total cost as ψ in $I\{Y\}$. Therefore $\mathsf{OPT} \leqslant \mathsf{OPT}'$. □

We proceed to show that, under suitable conditions, compressed instances are decent and belong to $\mathcal{C}$ if the original instance belonged to $\mathcal{C}$:

Lemma 5.4.8. *If $I \in \mathcal{C}$ and $Y \subseteq V(I)$, then $I\{Y\} \in \mathcal{C}$.*

Proof. As in Lemma 5.3.6, we can show that the Gaifman graph of $I\{Y\}$ is a minor of the Gaifman graph of I. Thus, $I\{Y\} \in \mathcal{C}$. □

Lemma 5.4.9. *Let $s, d, t \geqslant 1$. Consider an (s,d)-decent instance $I = (G, D, \mathsf{cost}, \mathsf{supply}, \mathsf{demand})$ and $Y \subseteq V(I)$. If for every connected component $C \in \mathrm{cc}(G \setminus Y)$, we have $|N(C)| \leqslant t$, then $I\{Y\} = (G', D', \mathsf{cost}', \mathsf{supply}', \mathsf{demand}')$ is $(st, d + 4^{st})$-decent.*

Proof. First observe that the set of edges incident to vertices of Y does not change under the compression; thus (s,d)-meager vertices of Y remain (s,d)-meager. For the same reason, each state-monotonous vertex of Y remains such.

Now consider a vertex $u \in I\{Y\} \setminus (Y \cup \{u_\ominus\})$ created as a result of a collapse of a component $C \in \mathrm{cc}(G \setminus Y)$ with neighborhood $S \subseteq Y$. By the definition of u, we see that $|D_u| = 4^{|E(C,S)|}$. Observe that $|E(C,S)| \leqslant |S| \cdot s$ as every vertex of S is (s,d)-meager and thus is incident to at most s edges. Also, $|S| \leqslant t$ from the assumption. Hence, $|D_u| \leqslant 4^{st}$. Also, it is easy to verify that the degree of u in $I\{Y\}$ is equal to $|E(C,S)| \leqslant st$.

It remains to verify that u is state-monotonous. Recall that $D'_u = \left(2^{\{\mathsf{Supply},\mathsf{Demand}\}}\right)^{|E(C,S)|}$. Consider two states $x_1, x_2 \in D'_u$, aiming to prove that there exists a combination x of x_1 with x_2. If $\mathsf{cost}'_u(x_1) + \mathsf{cost}'_u(x_2) = +\infty$, then it is enough to put as x the constant function always returning $\{\mathsf{Supply}\}$; i.e., the state x for which $\mathsf{supply}'_u(x) = \delta_{G'}(u)$ and $\mathsf{demand}'_u(x) = \emptyset$. Otherwise, there exist locally correct valuations ϕ_1, ϕ_2 of C, of cost exactly $\mathsf{cost}'_u(x_1)$ and $\mathsf{cost}'_u(x_2)$, respectively, so that $\mathsf{interaction}_{C,S}(\phi_1) = x_1$ and $\mathsf{interaction}_{C,S}(\phi_2) = x_2$. Construct a new valuation ϕ of C as follows: For each $v \in C$, let $\phi(v)$ be the combination of $\phi(x_1)$ with $\phi(x_2)$, and let $x \coloneqq \mathsf{interaction}_{C,S}(\phi)$. The following claims are straightforward and follow from the verification with the definitions.

Claim 5.4.10. *ϕ is locally correct on C and has cost at most $\mathsf{cost}'_u(x_1) + \mathsf{cost}'_u(x_2)$.*

Claim 5.4.11. *Let $e \in E(C,S)$. Then $\mathsf{Supply} \in x(e)$ if and only if $\mathsf{Supply} \in x_1(e)$ or $\mathsf{Supply} \in x_2(e)$. Moreover, if $\mathsf{Demand} \in x(e)$, then $\mathsf{Demand} \in x_1(e)$.*

It follows that x is a combination of x_1 with x_2. Since x_1, x_2 were chosen arbitrarily, we conclude that u is state-monotonous.

Finally, the vertex $u_\ominus$ is isolated and contains only one state in its domain; hence it is state-monotonous and $(1,1)$-decent. □

5.4.3 Dynamic maintenance of compressions

We now introduce the analog of Lemma 5.3.8 in the setting of Generalized Domination. Note that the proof of the following lemma follows closely that of Lemma 5.3.8, however we provide it here in full for completeness.

Lemma 5.4.12. *Let $w, n, s, d \in \mathbb{N}$. One can construct a data structure that supports the following operations:*

- ***Initialize** the data structure with an n-vertex (s,d)-decent instance of* Min Weight Generalized Domination *$I = (G, D, \mathsf{cost}, \mathsf{supply}, \mathsf{demand})$, where $G \in \mathcal{C}$ and $\mathrm{tw}(G) \leqslant w$.*
- ***Update** the instance I using one of the following update types:* AddVertex, AddEdge, RemoveEdge, UpdateCost. *It is guaranteed that after the update, we have that $G \in \mathcal{C}$ and the instance I after the update is (s,d)-decent.*

The initialization of the data structure is performed in time $2^{\mathcal{O}(sw)} \cdot d \cdot n \log n$. Afterwards, the data structure additionally maintains an instance $I^\star = (G^\star, D^\star, \mathsf{cost}^\star, \mathsf{supply}^\star, \mathsf{demand}^\star)$ of Min Weight Generalized Domination *with the following properties:*

(a) $G^\star \in \mathcal{C}$;

(b) the minimum cost of a solution to $I^\star$ is equal to that of I;

(c) $I^\star$ is $(\mathcal{O}(sw), d + 4^{\mathcal{O}(sw)})$-decent;

(d) after a sequence of $t \geqslant 0$ updates to I, we have $|V(I^\star)| \leqslant t \cdot s \cdot w^{\mathcal{O}(1)} \log n$. Moreover, on each update to I, the instance $I^\star$ can be updated in time $2^{\mathcal{O}(sw)} \cdot \log n$ and causes at most $s \cdot w^{\mathcal{O}(1)} \log n$ updates to $I^\star$.

Proof. As in the case of the independent set, preprocess G in time $2^{\mathcal{O}(w)} \cdot n \log n$ and produce an elimination forest F of G with $V(F) = V(G)$ with the following properties:

- F has height $\mathcal{O}(w \log n)$;
- for each $u \in V(F)$, we have $|\mathsf{Reach}_F(u)| = \mathcal{O}(w)$;
- for each $u \in V(F)$, the graph $G[\mathsf{desc}_F[u]]$ is connected.

For convenience, for each $u \in V(G)$, let $C_u = \mathsf{desc}_F[u]$, $S_u = \mathsf{Reach}_F(u) = N(C_u)$ and $X_u = \left(2^{\{\mathsf{Supply},\mathsf{Demand}\}}\right)^{E(C_u,S_u)}$. Observe that each element of $\mathsf{interaction}_{C_u,S_u}$ is an element of X_u. Note that $|S_u| \leqslant \mathcal{O}(w)$ and I is (s,d)-decent, so $|E(C_u,S_u)| \leqslant \mathcal{O}(sw)$ and therefore $|X_u| \leqslant 2^{\mathcal{O}(sw)}$.

Observe also that the nodes in the elimination forest have bounded degrees:

Claim 5.4.13. *Every node $u \in V(F)$ has at most $\mathcal{O}(sw)$ children in F.*

Proof of the claim. Enumerate the children of u in F: $c_1, \ldots, c_t$. For each c_i, we have $\mathsf{Reach}_F(c_i) \subseteq \mathsf{Reach}_F(u) \cup \{u\}$. Each such c_i is thus a neighbor of some vertex in $\mathsf{Reach}_F(u) \cup \{u\}$. The claim follows since each vertex of G has degree at most s and that $|\mathsf{Reach}_F(u) \cup \{u\}| \leqslant \mathcal{O}(w)$. ◁

On initialization of the data structure, compute the following dynamic programming table:

- For each $u \in V(F)$ and every interaction $x \in X_u$, compute $T[u][x]$: the minimum possible cost of a locally correct partial solution $\phi \in \prod_{x \in C_u} D_x$ such that $\mathsf{interaction}_{C_u,S_u}(\phi) = x$; or $+\infty$ if no such partial solution exists.

The table $T[\cdot][\cdot]$ has at most $n \cdot 2^{\mathcal{O}(sw)}$ states and can be computed in time $nd \cdot 2^{\mathcal{O}(sw)}$. Hence, the total preprocessing time is upper-bounded by $2^{\mathcal{O}(sw)} \cdot nd \log n$. Also observe that the table $T[\cdot][\cdot]$ suffices to compute the minimum-cost solution for I at the time of initialization: the cost is equal to the sum over $T[r][\emptyset]$, ranging over all roots r of trees of F.

As before, in the beginning we set $A = B = Z = \emptyset$.
Let $I^{\mathrm{init}} = (G^{\mathrm{init}}, D^{\mathrm{init}}, \mathsf{cost}^{\mathrm{init}}, \mathsf{supply}^{\mathrm{init}}, \mathsf{demand}^{\mathrm{init}})$ be the initial instance of 2CSP. In the sequel, by $I^{\mathrm{cur}} = (G^{\mathrm{cur}}, D^{\mathrm{cur}}, \mathsf{cost}^{\mathrm{cur}}, \mathsf{supply}^{\mathrm{cur}}, \mathsf{demand}^{\mathrm{cur}})$ we denote the current snapshot of the instance in the data structure; initially, $I^{\mathrm{cur}} := I^{\mathrm{init}}$.

Recall from the setting of the Independent Set that throughout the life of the data structure, we maintain the following invariants:

- $A = V(I^{\mathrm{cur}}) \setminus V(I^{\mathrm{init}})$ is the set of vertices added to I since the instantiation of the data structure.
- $B \subseteq V(I^{\mathrm{cur}})$ is the set of vertices that were part of any update to I so far (i.e., $v \in B$ if v was added to I, the cost of v was changed, or an edge incident to v was added or removed).
- $Z = A \cup \mathsf{anc}_F[B \setminus A]$. In other words, Z contains all vertices of A and the ancestors in F of all vertices of $B \setminus A$.
- $I^\star$ is equivalent to $I^{\mathrm{cur}}\{Z\}$.

Now, the costs of the vertices in the compressed instance can be inferred from the entries of $T[\cdot][\cdot]$:

- The connected components of $G^{\mathrm{cur}} \setminus Z$ adjacent to Z are exactly the components of the form $\mathsf{desc}_F[y]$, where $y \notin Z$ is not a root of any tree in F and $\mathsf{parent}_F(y) \in Z$. In the compression, for each y, we introduce one vertex $u_y \in I\{Z\} \setminus Z$ with the domain $D_{u_y} = X_y$ and the cost function $\mathsf{cost}_{u_y} : D_{u_y} \to \mathbb{R}_{\geqslant 0} \cup \{+\infty\}$, where $\mathsf{cost}_{u_y}(x) = T[y][x]$ for $x \in D_{u_y}$.
- The connected components of $G^{\mathrm{cur}} \setminus Z$ nonadjacent to Z are exactly the components of the form $\mathsf{desc}_F[r]$, where $r \notin Z$ is a root of a tree in F. All these connected components are collapsed to a single vertex u_ε with a single state ε, of cost that is the sum over $T[r][\emptyset]$ for all roots $r \notin Z$ of trees in F.

Recall the following claim from the setting of the Independent Set. Note that since the sets A, B, Z are constructed as in that setting, the claim also holds here.

Claim 5.4.14. *On each update to I, the set Z grows by at most $\mathcal{O}(w \log n)$ vertices.*

Two instances I_1, I_2 of Min Weight Generalized Domination are *equivalent* if the instances are isomorphic after the removal of all isolated vertices with the constant-zero cost function. We now sketch a single update to Z – an addition of an appendix of Z.

Claim 5.4.15. *Let two sets $Z_1, Z_2 \subseteq V(G^{\mathrm{cur}})$ be such that $A \subseteq Z_1 \subseteq Z_2 \subseteq V(G^{\mathrm{cur}})$, $|Z_2| = |Z_1| + 1$ and $Z_1 \setminus A$ and $Z_2 \setminus A$ are prefixes of F with $Z_2 - Z_1 \subseteq V(F)$. Then, an instance equivalent to $I^{\mathrm{cur}}\{Z_2\}$ can be obtained from an instance equivalent to $I^{\mathrm{cur}}\{Z_1\}$ through a sequence of $sw^{\mathcal{O}(1)}$ updates. Moreover, this sequence can be computed in time $2^{\mathcal{O}(sw)}$.*

Proof of the claim. Let $z \in V(F)$ be such that $Z_2 = Z_1 \cup \{z\}$. Let $\mathfrak{C}_1$ be the set of connected components of $G^{\mathrm{cur}} \setminus Z_1$, $\mathfrak{C}_2$ be the set of connected components of $G^{\mathrm{cur}} \setminus Z_2$ and $c_1, \ldots, c_t$ be the set of children of z in F. We have that $\mathfrak{C}_1 \setminus \mathfrak{C}_2 = \{G^{\mathrm{cur}}[\mathsf{desc}_F[z]]\}$ and $\mathfrak{C}_2 \setminus \mathfrak{C}_1 = \{G^{\mathrm{cur}}[\mathsf{desc}_F[c_1]], \ldots, G^{\mathrm{cur}}[\mathsf{desc}_F[c_t]]\}$.

Let $S = \mathsf{Reach}_F(z)$. The compressed instance $I^{\mathrm{cur}}\{Z_1\}$ contains a vertex v_S representing the union of all connected components of $G^{\mathrm{cur}} \setminus Z_1$ whose neighborhoods are exactly S; and $G^{\mathrm{cur}}[\mathsf{desc}_F[z]]$ is one of such components. To obtain $I^{\mathrm{cur}}\{Z_2\}$ from $I^{\mathrm{cur}}\{Z_1\}$, we need to:

1. Remove the contribution of $G^{\mathrm{cur}}[\mathsf{desc}_F[z]] \in \mathfrak{C}_1 \setminus \mathfrak{C}_2$ from the compressed instance. If z is nonroot, then the vertex u_z corresponding to the collapse of $\mathsf{desc}_F[z]$ should be removed from the instance; this is emulated by removing all edges incident to u_z and replacing the cost of u_z with the zero function; this can be done using $\mathcal{O}(w)$ updates to the compressed instance. Otherwise, $\mathsf{desc}_F[z]$ is collapsed to the special vertex u_ε (possibly with other connected components nonadjacent to Z). Then, u_ε has only one state ε, and its cost is adjusted by simply subtracting $T[z][\emptyset]$.
2. Add the vertex z to the compressed instance. Since the set of neighbors of z in Z_1 is exactly $\mathsf{Reach}_F(z)$, this requires one vertex addition and $\mathcal{O}(w)$ edge additions.
3. Include the contribution of the connected components $G^{\mathrm{cur}}[\mathsf{desc}_F[c_1]], \ldots, G^{\mathrm{cur}}[\mathsf{desc}_F[c_t]] \in \mathfrak{C}_2 \setminus \mathfrak{C}_1$ in the compressed instance. Since $t \leqslant \mathcal{O}(sw)$ by Claim 5.4.13, we simply iterate all components and for each, we add the corresponding vertex u_{c_i} to the instance (with the costs of the states sourced from T), along with at most $\mathcal{O}(w)$ edges incident to each vertex. Thus, the number of updates to the compressed instance is $\mathcal{O}(sw^2)$. For each fresh vertex, we iterate over all $|X_{c_i}| = 2^{\mathcal{O}(sw)}$ states u_{c_i} can attain, so the total time of the update is $\mathcal{O}(sw^2) \cdot 2^{\mathcal{O}(sw)} = 2^{\mathcal{O}(sw)}$. ◁

As before, the implementation of the data structure directly follows from Claims 5.4.14 and 5.4.15. Since on each update to I^{cur}, the set Z is updated $\mathcal{O}(w \log n)$ times, the total number of updates performed on $I^\star$ is $\mathcal{O}(w \log n) \cdot sw^{\mathcal{O}(1)} = sw^{\mathcal{O}(1)} \log n$. Therefore, the total update time is $2^{\mathcal{O}(sw)} \log n$. □

5.4.4 Full algorithm

We are now ready to give the exposition of the data structure proving Lemma 5.4.1. The rest of this section describes the data structure $\mathbb{D}^{\text{main}}$ maintaining a (s,d)-decent dynamic instance of Min Weight Generalized Domination $I^{\text{main}} = (G^{\text{main}}, D^{\text{main}}, \mathsf{cost}^{\text{main}}, \mathsf{supply}^{\text{main}}, \mathsf{demand}^{\text{main}})$. Similarly to the case of Max Weight Nullary 2CSP, we fix an integer $L \in \mathbb{N}$ and set $k := \lceil L/\delta \rceil$. We construct a recursive, L-level data structure comprised of auxiliary data structures maintaining subinstances of Min Weight Generalized Domination; the only data structure at level L maintains I^{main}, and each data structure $\mathbb{D}$ at level $q \in \{2,3,\ldots,L\}$ stores a collection of k children data structures at level $q-1$.

A data structure $\mathbb{D}$ at level q maintaining an instance $I = (G, D, \mathsf{cost}, \mathsf{supply}, \mathsf{demand})$ preserves the following invariants:

(I1) $G \in \mathcal{C}$ and $|V(G)| \leqslant n^{q/L}$;

(I2) I is $(\widehat{s}(q), \widehat{d}(q))$-decent, for some functions $\widehat{s}$, $\widehat{d}$ to be exactly specified later;

(I3) $\mathbb{D}$ maintains a nonnegative real p satisfying $(1 - \delta \cdot \frac{q}{L})\mathsf{OPT} \leqslant p \leqslant \mathsf{OPT}$, where OPT is the minimum cost of a solution to I.

Each data structure at level 1 maintains an instance I of size at most $n^{1/L}$ and computes an approximate lower bound on the cost of the solution within the $\frac{\delta}{L}$ fraction of OPT, by rerunning the Baker's technique on each update. By Lemma 5.4.5, each update to such a data structure takes time $n^{1/L} \cdot \widehat{s}(1) \cdot \widehat{d}(1)^{\mathcal{O}(L/\delta)}$.

Now consider a data structure $\mathbb{D}$ at level $q \geqslant 2$ maintaining a $(\widehat{s}(q), \widehat{d}(q))$-decent instance $I = (G, D, \mathsf{cost}, \mathsf{supply}, \mathsf{demand})$ of Min Weight Generalized Domination. We define a variable $I^{\text{cur}} = (G^{\text{cur}}, D^{\text{cur}}, \mathsf{cost}^{\text{cur}}, \mathsf{supply}^{\text{cur}}, \mathsf{demand}^{\text{cur}})$ tracking the current snapshot of I. The lifetime of $\mathbb{D}$ is partitioned into *epochs*: sequences of τ_q updates to I, with τ_q to be specified later. The first epoch begins when $\mathbb{D}$ is initialized; and a new epoch begins each time $\mathbb{D}$ processes τ_q updates to I.

Let $I^{\text{old}} = (G^{\text{old}}, D^{\text{old}}, \mathsf{cost}^{\text{old}}, \mathsf{supply}^{\text{old}}, \mathsf{demand}^{\text{old}})$ be the snapshot of I at the start of each epoch; that is, at the start of an epoch, we set $I^{\text{old}} := I^{\text{cur}}$. We partition $V(G^{\text{old}})$ into layers $L_0, \ldots, L_{4k-1}$ as follows: If G^{old} is connected, choose a vertex $r \in V(G)$ and assigning a vertex $v \in V(G)$ to L_j if and only if $\mathsf{dist}_{G^{\text{old}}}(r,v) \equiv j \mod 4k$. If G^{old} is disconnected, produce a partition into layers for each connected components, and let L_j be the union over jth layers for each connected component.

We now define pairwise disjoint dynamic sets $V_0, \ldots, V_{k-1}$. For each $i \in \{0,\ldots,k-1\}$, let $V_i^{\text{old}} := L_{4i+1} \cup L_{4i+2}$ be the initial contents of V_i. Thanks to this choice, all V_i-cleared subinstances of I have small treewidth:

Lemma 5.4.16. *For each $i \in \{0,\ldots,k-1\}$, the treewidth of the Gaifman graph of $\mathsf{Clear}(I^{\text{old}}; V_i)$ is bounded by $\mathcal{O}(k)$.*

Proof. Let C be a connected component of the Gaifman graph of $\mathsf{Clear}(I^{\text{old}}; V_i)$. Note that C cannot simultaneously contain vertices at distance $4k\ell + (4i+1)$ and $4k\ell + (4i+2)$, for any $\ell \in \mathbb{N}$. Therefore, C is contained within the set of vertices of $\mathsf{Clear}(I^{\text{init}}; V_i)$ at distance at least ℓ and at most $\ell + 4k - 1$, for some $\ell \in \mathbb{N}$. By Lemma 5.3.2, C has treewidth bounded by $\mathcal{O}(k)$. Therefore, the entire Gaifman graph of $\mathsf{Clear}(I^{\text{old}}; V_i)$ has treewidth bounded by $\mathcal{O}(k)$. □

Note that the sets V_i are, contrary to the case of Max Weight Nullary 2CSP, dynamic – when processing some updates, we may decide to shrink some of these sets. Given that, let $i \in \{0,\ldots,k-1\}$, let V_i^{cur} track the current contents of V_i. We maintain the following invariants about each set V_i^{cur}:

- For each $i \in \{0,\ldots,k-1\}$, we have that $V_i^{\text{cur}} \subseteq V_i^{\text{old}}$.
- The sets $N_{G^{\text{cur}}}[V_0^{\text{cur}}], \ldots, N_{G^{\text{cur}}}[V_{k-1}^{\text{cur}}]$ are pairwise disjoint.

By the construction of V_i, the invariants are satisfied at the time of the initialization: We have $G^{\text{cur}} = G^{\text{init}}$, so for each $i \in \{0,\ldots,k-1\}$ it holds that $N_{G^{\text{cur}}}[V_i^{\text{cur}}] \subseteq L_{4i} \cup L_{4i+1} \cup L_{4i+2} \cup L_{4i+3}$. Note that from the invariant it follows that the sets V_i^{cur} are pairwise disjoint.

Next, we maintain k universes $I_0^{\text{cur}}, \ldots, I_{k-1}^{\text{cur}}$, where $I_i^{\text{cur}} = \mathsf{Clear}(I^{\text{cur}}; V_i^{\text{cur}})$. For each universe, we initialize a data structure from Lemma 5.4.12, maintaining an instance $I_i^\star$ that is a compression of I_i^{cur} with the same minimum cost of a solution. Finally, we initialize k child data structures $\mathbb{D}_0, \ldots, \mathbb{D}_{k-1}$ at level $q-1$, where each $\mathbb{D}_i$ stores the compressed instance $I_i^\star$ and maintains a $(1 - \delta\frac{q-1}{L})$-approximation to the minimum cost of a solution to $I_i^\star$. As previously, this initialization is recursive: Each child data

structure maintains another collection of k child data structures at level $q-2$ each, etc., until construction k^{q-1} data structures at level 1 in total.

Assume each child data structure $\mathbb{D}_i$ maintains a good lower bound p_i on the minimum cost of a solution to I_i^{cur}; then we can also preserve a good lower bound on the minimum cost of a solution to I by keeping the maximum of the lower bounds:

Lemma 5.4.17. *Let OPT_i be the minimum cost of a solution to the instance I_i^{cur}, and let p_i be so that $(1-\delta\frac{q-1}{L})\mathsf{OPT}_i \leqslant p_i \leqslant \mathsf{OPT}_i$. Let $p = \max(p_0,\ldots,p_{k-1})$. Then $(1-\delta\frac{q}{L})\mathsf{OPT} \leqslant p \leqslant \mathsf{OPT}$, where OPT is the minimum cost of a solution to I.*

Proof. Recall that the closed neighborhoods of the sets $V_0^{\mathrm{cur}},\ldots,V_{k-1}^{\mathrm{cur}}$ are pairwise disjoint. Thus, Lemma 5.4.3 applies to the cleared subinstances $I_0^{\mathrm{cur}},\ldots,I_{k-1}^{\mathrm{cur}}$ and we get that:

- for every $j \in \{0,\ldots,k-1\}$, we have that $\mathsf{OPT}_j \leqslant \mathsf{OPT}$;
- there exists $j \in \{0,\ldots,k-1\}$ such that $\mathsf{OPT}_j \geqslant (1-\frac{1}{k})\mathsf{OPT}$.

Thus, by our assumptions, for every $j \in \{0,\ldots,k-1\}$ we have $p_j \leqslant \mathsf{OPT}$, so also $p \leqslant \mathsf{OPT}$. Moreover, there exists $j \in \{0,\ldots,k-1\}$ with $\mathsf{OPT}_j \geqslant (1-\frac{1}{k})\mathsf{OPT}$. For this value of j, we have $p_j \geqslant (1-\delta\frac{q-1}{L})\mathsf{OPT}_j$, so in total, $p_j \geqslant (1-\delta\frac{q-1}{L})(1-\delta\frac{1}{L})\mathsf{OPT} \geqslant (1-\delta\frac{q}{L})\mathsf{OPT}$. As $p \geqslant p_j$, we conclude that $p \geqslant (1-\delta\frac{q}{L})\mathsf{OPT}$. □

We now show how the data structure is updated. First, we describe the removal of a vertex $v \in V_i^{\mathrm{cur}}$ from V_i^{cur}. In this setting, the cleared subinstance I_i^{cur} must be updated: Each of the edges of I^{cur} with one endpoint in v and the other endpoint in $V_i^{\mathrm{cur}} \setminus \{v\}$ must be introduced to I_i^{cur}. Moreover, since v ceases to be relieved in I_i^{cur}, the function demand_v in I_i^{cur} must be amended; this is done by removing and reinserting all already existing edges incident to v with the updated demand sets. Since I^{cur} is $(\widehat{s}(q),\widehat{d}(q))$-decent, the entire process involves $\mathcal{O}(\widehat{s}(q))$ updates to I_i^{cur}. Each of these updates, in turn, causes the data structure from Lemma 5.4.12 to apply a number of updates to $I_i^\star$, which are then relayed to $\mathbb{D}_i$.

Now, we implement an update to I^{cur} (i.e, AddVertex, AddEdge, RemoveEdge, or UpdateCost). For every vertex v involved in the update (a fresh vertex, an endpoint of an added or removed edge, or a vertex with its cost updated), whenever v belongs to some set V_i^{cur}, we first remove v from V_i^{cur} as described above. Afterwards, we apply the update to each subinstance I_i^{cur}; again, the data structure from Lemma 5.4.12 then issues a number of updates to $I_i^\star$, which we relay to $\mathbb{D}_i$. Finally, $\mathbb{D}$ recomputes a lower bound on the minimum cost of a solution to I^{cur} by querying each $\mathbb{D}_i$ for the lower bound p_i on the minimum cost of a solution to $I_i^\star$ and returning the maximum value.

It remains to verify the satisfaction of the invariants. First, we verify the invariants regarding the sets $V_0^{\mathrm{cur}},\ldots,V_{k-1}^{\mathrm{cur}}$. The only nontrivial property is the preservation of the disjointness of the closed neighborhoods of the sets after the addition of an edge to I^{cur}:

Lemma 5.4.18. *Let G be a graph and $V_0,\ldots,V_{k-1} \subseteq V(G)$ be so that $N_G[V_0],\ldots,N_G[V_{k-1}]$ are pairwise disjoint. Let also $u,v \in V(G)$, and consider a graph G' equal to G with an edge uv added, and $V_i' = V_i \setminus \{u,v\}$ for $i \in \{0,\ldots,k-1\}$. Then $N_{G'}[V_0'],\ldots,N_{G'}[V_{k-1}']$ are pairwise disjoint.*

Proof. Let $i \in \{0,\ldots,k-1\}$ and suppose there exists a vertex $w \in N_{G'}[V_i'] \setminus N_G[V_i]$. Equivalently, $N_{G'}[w]$ intersects V_i', but $N_G[w]$ is disjoint from V_i. If $w \notin \{u,v\}$, then $N_{G'}[w] = N_G[w]$ and $V_i' \subseteq V_i$ – a contradiction. Now assume without loss of generality that $w = u$. Then $N_{G'}[w] \subseteq N_G[w] \cup \{v\}$, but $V_i' \subseteq V_i \setminus \{v\}$ – again a contradiction. We conclude that $N_{G'}[V_i'] \subseteq N_G[V_i]$ for every $0 \leqslant i < k$, so if the sets $N_G[V_0],\ldots,N_G[V_{k-1}]$ were pairwise disjoint, then so are $N_{G'}[V_0'],\ldots,N_{G'}[V_{k-1}']$. □

Since each update to I^{cur} causes at most $\mathcal{O}(\widehat{s}(q))$ updates to each I_i^{cur}, it follows that during one epoch, the size of each $I_i^\star$ does not grow above $\tau_q \cdot (\widehat{s}(q)\cdot k)^{\mathcal{O}(1)} \log n$ (Lemma 5.4.12(d)). Later, we will choose τ_q so that this value is significantly less than $n^{(q-1)/L}$ so as to ensure the satisfaction of invariant (I1) by the children data structures; the fact that the Gaifman graph of $I_i^\star$ belongs to $\mathcal{C}$ follows from Lemma 5.4.12(a). The invariant (I2) is satisfied due to Lemma 5.4.12(c), provided we set $\widehat{s}$ and $\widehat{d}$ according to the bounds provided by Lemma 5.4.12. Also, since the revenue of an optimum solution to $I_i^\star$ is equal to that of I_i^{cur} (Lemma 5.4.12(b)), each child data structure $\mathbb{D}_i$ maintains a nonnegative real p_i satisfying the preconditions of Lemma 5.4.17. Therefore, the value $p := \max(p_0,\ldots,p_{k-1})$ is an $(1-\delta\frac{q}{L})$-approximation of the optimum revenue in I^{cur}, which proves the satisfaction of invariant (I3) by $\mathbb{D}$. Finally, as previously, when an epoch in $\mathbb{D}$ ends, $\mathbb{D}$ is reinitialized with $I^{\mathrm{old}} := I^{\mathrm{cur}}$ and a new epoch starts, causing the destruction and the recursive reinitialization of the children data structures $\mathbb{D}_i$.

5.4.5 Setting the parameters and time complexity analysis

Here, we perform the complexity analysis of the data structure for Min Weight Generalized Domination. That is, we prove the following statement:

Lemma 5.4.19. *Suppose that the values s, d are absolute constants. Then the parameters L, $\widehat{s}(\cdot)$, $\widehat{d}(\cdot)$ and τ. can be chosen so that the data structure for* Min Weight Generalized Domination *performs each update in time $f(\delta) \cdot n^{o(1)}$, where $f(\delta)$ is doubly-exponential in $\mathcal{O}(1/\delta^2)$.*

The proof is an adaptation of Section 5.3.5 to the setting of Min Weight Generalized Domination with slightly different bounds in several parts of the analysis.

Proof. Recall that all constants depending on $\mathcal{C}$ are absolute constants.

First, we bound the values of $\widehat{s}(\cdot)$ and $\widehat{d}(\cdot)$.

Claim 5.4.20. *One can set functions $\widehat{s}$ and $\widehat{d}$ so that the invariant (I2) is satisfied for all constructed data structures, and moreover $\widehat{s}(q) \in k^{\mathcal{O}(L)}$ and $\widehat{d}(q) \in 2^{k^{\mathcal{O}(L)}}$ for all $q \in \{1, \ldots, L\}$.*

Proof of the claim. Recall that (c) states that the data structures at level q maintain $(\widehat{s}(q), \widehat{d}(q))$-decent instances of Min Weight Generalized Domination. By the assumptions, it is enough to set $\widehat{s}(L) = s$ and $\widehat{d}(L) = d$, i.e., $\widehat{s}(L)$ and $\widehat{d}(L)$ are absolute constants. Assuming that a given data structure $\mathbb{D}$ at level $q > 1$ maintains a $(\widehat{s}(q), \widehat{d}(q))$-decent instance I of Min Weight Generalized Domination, $\mathbb{D}$ stores a collection of children data structures at level $q-1$, each maintaining a compression of I produced by Lemma 5.4.12; from Lemma 5.4.12(c) we infer that each such compression is $(\mathcal{O}(\widehat{s}(q)w), \widehat{d}(q) + 4^{\mathcal{O}(\widehat{s}(q)w)})$-decent, where $w \in \mathcal{O}(k)$. Choose an absolute constant $C > 0$ so that for every $s', d' \geqslant 1$, a compression of a (s', d')-decent instance is $(C \cdot s'k, d' + 4^{C \cdot s'k})$-decent. Then it is enough to set $\widehat{s}(q-1) = \widehat{s}(q) \cdot Ck$ and $\widehat{d}(q-1) = \widehat{d}(q) + 4^{C \cdot \widehat{s}(q)k}$. A straightforward induction implies that $\widehat{s}(q) = s \cdot (Ck)^{L-q}$. Therefore, assuming $k \geqslant 2$, we have $\widehat{s}(q) \in k^{\mathcal{O}(L)}$ and $\widehat{d}(q) \in 2^{k^{\mathcal{O}(L)}}$ for all $q \in \{1, \ldots, L\}$. ◁

Each auxiliary data structure at level q spawns k auxiliary data structures at level $q-1$ for $q \geqslant 2$; and each auxiliary data structure at level $q = 1$ is a leaf, i.e., it does not spawn any children data structures. Hence, there are at most k^L auxiliary data structures at level 1. Therefore:

Claim 5.4.21. *Each update to $\mathbb{D}^{\text{main}}$ causes at most $k^{\mathcal{O}(L^2)}(\log n)^{\mathcal{O}(L)}$ updates throughout all data structures.*

Proof of the claim. By Lemma 5.4.12(d), each update to a universe I_i^{cur} causes at most $\widehat{s}(q) w^{\mathcal{O}(1)} \log n$ updates propagated to instances at level $q-1$. Since each instance at level $q > 1$ has k associated universes I_i^{cur} and each update to I^{cur} causes at most $\mathcal{O}(\widehat{s}(q))$ updates to each universe I_i^{cur}, we conclude that a single update to an instance maintained by a data structure at level q causes at most

$$\widehat{s}(q)^2 w^{\mathcal{O}(1)} \log n \tag{5.1}$$

updates to be propagated to the children data structures at level $q-1$. Recall that $w \in \mathcal{O}(k)$ and that by Claim 5.4.20 we have $\widehat{s}(q) \in k^{\mathcal{O}(L)}$. Therefore, (5.1) is bounded by

$$k^{\mathcal{O}(L)} \log n.$$

By straightforward induction, a single update to $\mathbb{D}^{\text{main}}$ (the data structure at level L) causes at most

$$\mathcal{O}\left(\left(k^{\mathcal{O}(L)} \log n\right)^L\right) \leqslant k^{\mathcal{O}(L^2)} (\log n)^{\mathcal{O}(L)}.$$

updates to all data structures. ◁

Claim 5.4.22. *The total time required to update all data structures for a single update to $\mathbb{D}^{\text{main}}$, excluding the time of all required reinitializations, is bounded by $n^{1/L} \cdot 2^{(\frac{L}{\delta})^{\mathcal{O}(L)} + \mathcal{O}(L \log\log n)}$.*

Proof of the claim. Focus first on the time required to recompute the solutions to the instances at level 1. Each data structure at level 1 recomputes the solution from scratch using Lemma 5.4.5 with an instance with at most $n^{1/L}$ vertices, the bound on the maximum degree $s = \widehat{s}(1)$, the bound on the domain size $d = \widehat{d}(1)$ and the approximation guarantee $\frac{\delta}{L}$. Therefore, each update at level 1 is processed in

time $n^{1/L} \cdot \widehat{s}(1) \cdot \widehat{d}(1)^{\mathcal{O}(L/\delta)} \leqslant n^{1/L} \cdot 2^{k^{\mathcal{O}(L)} \cdot \frac{L}{\delta}}$. At a data structure at level $q > 1$, it can be verified that each update is processed in time $2^{\mathcal{O}(\widehat{s}(q)k)} \cdot (\log n)^{\mathcal{O}(1)}$ (excluding the time spent at data structures at lower levels). Since $2^{\mathcal{O}(\widehat{s}(q)k)} \cdot (\log n)^{\mathcal{O}(1)} \leqslant 2^{k^{\mathcal{O}(L)}} \cdot (\log n)^{\mathcal{O}(1)}$, we conclude that for any fixed auxiliary data structure, every update is processed in time at most $n^{1/L} \cdot (\log n)^{\mathcal{O}(1)} \cdot 2^{k^{\mathcal{O}(L)} \cdot \frac{L}{\delta}}$.

Using the bound from Claim 5.4.21 on the total number of updates to all data structures, we infer that, for a single update to $\mathbb{D}^{\mathrm{main}}$, the total recomputation time in the data structures is bounded by $k^{\mathcal{O}(L^2)}(\log n)^{\mathcal{O}(L)} n^{1/L} (\log n)^{\mathcal{O}(1)} 2^{k^{\mathcal{O}(L)} \cdot \frac{L}{\delta}}$. Recall that we set $k = \lceil \frac{L}{\delta} \rceil$, so we have that $k^{\mathcal{O}(L^2)} = 2^{\mathcal{O}(L^2 \log \frac{L}{\delta})}$, $(\log n)^{\mathcal{O}(L)} = 2^{\mathcal{O}(L \log\log n)}$, and $2^{k^{\mathcal{O}(L)} \cdot \frac{L}{\delta}} = 2^{(\frac{L}{\delta})^{\mathcal{O}(L)}}$. Therefore, we can bound the total update time, excluding the reinitializations, by

$$n^{1/L} \cdot 2^{\mathcal{O}(L^2 \log \frac{L}{\delta} + L \log\log n + (\frac{L}{\delta})^{\mathcal{O}(L)})} \leqslant n^{1/L} \cdot 2^{(\frac{L}{\delta})^{\mathcal{O}(L)} + \mathcal{O}(L \log\log n)}.$$

◁

Now we set the epoch lengths and bound the amortized time of all reinitializations per a single update to $\mathbb{D}^{\mathrm{main}}$. Let τ_q denote the epoch length for data structures on level q for some $2 \leqslant q \leqslant L$. Recall that the epoch length for a data structure is measured in the number of updates to this particular instance (as opposed to $\mathbb{D}^{\mathrm{main}}$). Recall also from Item (d) of Lemma 5.4.12 that each update to a data structure $\mathbb{D}$ at level q generates $(\widehat{s}(1)k \log n)^{\mathcal{O}(1)} \leqslant k^{\mathcal{O}(L)} (\log n)^{\mathcal{O}(1)}$ updates to the children structures. Fix now a constant $c > 0$ such that this number of updates is actually bounded by $(k^L \log n)^c$, and set $\tau_q := n^{(q-1)/L}/(k^L \log n)^c$. Then, Lemma 5.4.12(d) implies invariant (I1): Children data structures at level $q-1$ are guaranteed to maintain instances of size $n^{(q-1)/L}$.

We now bound the amortized time complexity of initializations and reinitializations across all data structures.

Claim 5.4.23. *The amortized time of all initializations and reinitializations per a single update to* $\mathbb{D}^{\mathrm{main}}$ *is bounded by* $n^{1/L} \cdot 2^{(\frac{L}{\delta})^{\mathcal{O}(L)} + \mathcal{O}(L \log\log n)}$.

Proof of the claim. Consider an initialization or a reinitialization of a data structure $\mathbb{D}$, maintaining an instance I of Min Weight Generalized Domination, at some level $q \geqslant 2$. The (re)initialization takes time $|V(I)| \log |V(I)| \cdot 2^{k^{\mathcal{O}(L)}}$ by Lemma 5.4.12 and Claim 5.4.20. Thanks to the invariants, we have $|V(G)| \leqslant n^{q/L}$ and $|E(G)| = O(|V(G)|)$, hence the (re)initialization time is bounded by $n^{q/L} \log n \cdot 2^{k^{\mathcal{O}(L)}}$. Note that the (re)initialization of $\mathbb{D}$ causes all children data structures to be destructed and reset to constant-sized instances; however, this can be done in amortized constant time per instance.

Consider the state of $\mathbb{D}$ after some u updates; let $t = \lceil u/\tau_q \rceil$ be the index of the current epoch of $\mathbb{D}$, so that the lifetime of $\mathbb{D}$ is assumed to contain $t-1$ full epochs so far. Then, the total time of all reinitializations of $\mathbb{D}$ so far is bounded by $(t-1) \cdot n^{q/L} \log n \cdot 2^{k^{\mathcal{O}(L)}} \leqslant (t-1) \cdot \tau_q \cdot n^{1/L} \cdot (k^L \log n)^{c+1} \cdot 2^{k^{\mathcal{O}(L)}} \leqslant u \cdot n^{1/L} (\log n)^{\mathcal{O}(1)} \cdot 2^{k^{\mathcal{O}(L)}}$. Hence, the amortized time for the reinitializations of $\mathbb{D}$ can be bounded by $n^{1/L} \cdot (\log n)^{\mathcal{O}(1)} \cdot 2^{k^{\mathcal{O}(L)}}$ per an update to $\mathbb{D}$.

As each update to $\mathbb{D}^{\mathrm{main}}$ causes $k^{\mathcal{O}(L^2)} (\log n)^{\mathcal{O}(L)}$ updates to all auxiliary data structures in total and there are at most k^L auxiliary data structures, we conclude that the total amortized time required for all reinitializations per a single update to $\mathbb{D}^{\mathrm{main}}$ is bounded by

$$k^{\mathcal{O}(L^2)} \cdot n^{1/L} \cdot (\log n)^{\mathcal{O}(L)} \cdot 2^{k^{\mathcal{O}(L)}} \leqslant n^{1/L} (\log n)^{\mathcal{O}(L)} \cdot 2^{(\frac{L}{\delta})^{\mathcal{O}(L)}} = n^{1/L} \cdot 2^{(\frac{L}{\delta})^{\mathcal{O}(L)} + \mathcal{O}(L \log\log n)}.$$

◁

Note that Claims 5.4.22 and 5.4.23 provide the same time complexity bounds as Lemmas 5.3.14 and 5.3.15 in the setting of Max Weight Nullary 2CSP. Therefore, the following final time complexity analysis is shown by the same argument as Lemma 5.3.16, hence we omit its proof.

Claim 5.4.24. *L may be set so that the amortized time of an update to* $\mathbb{D}^{\mathrm{main}}$ *is* $n^{\mathcal{O}(\frac{\log\log\log n}{\log\log n \cdot \delta})} = n^{o(\frac{1}{\delta})}$.

Finally, as before, we use the following trick to separate δ from n in the final time complexity. If $\log\log n > \frac{1}{\delta^2}$, then $\delta > \sqrt{\log\log n}$, hence $n^{\mathcal{O}(\frac{\log\log\log n}{\log\log n \cdot \delta})} = n^{\mathcal{O}\left(\frac{\log\log\log n}{\sqrt{\log\log n}}\right)} = n^{o(1)}$. In the opposite case, we have $n \leqslant 2^{2^{1/\delta^2}}$; in this case, on every update, we rerun the static algorithm from Lemma 5.4.5 in time $nsd^{\mathcal{O}(1/\delta)} \leqslant 2^{2^{\mathcal{O}(1/\delta^2)}}$. In any case, the time complexity of an update is bounded by $f(\delta) \cdot n^{\mathcal{O}\left(\frac{\log\log\log n}{\sqrt{\log\log n}}\right)}$, where $f(\delta) \in 2^{2^{\mathcal{O}(1/\delta^2)}}$. This finishes the proof. □

5.5 Conclusions

We presented dynamic approximation schemes for Maximum Weight Independent Set and Minimum Weight Dominating Set in apex-minor-free classes of graphs. For both problems, we designed data structures with amortized subpolynomial update time that maintain, for a fixed $\varepsilon > 0$, a $(1-\varepsilon)$-approximation of the maximum weight of an independent set in a dynamic graph; and a $(1+\varepsilon)$-approximation of the minimum weight of a dominating set in a dynamic graph under an additional assumption that at all times, the maximum degree of the graph is bounded by a fixed constant. We remark that the results of this chapter can be adapted in a straightforward way to also solve the more general variants of Maximum Weight Independent Set and Minimum Weight Dominating Set introduced in Section 5.2 – namely, Max Weight Nullary 2CSP and Min Weight Generalized Domination. That is:

- For every fixed apex-minor-free class of graphs $\mathcal{C}$, parameter $\varepsilon > 0$, and upper bound K on the size of a domain of a vertex, we can maintain a fully dynamic nullary 2CSP I with a fixed set of n vertices, each of which has a (possibly different) domain of size at most K. We require that after each update, the Gaifman graph of I should belong to $\mathcal{C}$. Then after each update, the data structure can output a nonnegative real p satisfying $(1-\varepsilon)\mathsf{OPT}_I \leqslant p \leqslant \mathsf{OPT}_I$, where OPT_I is the maximum revenue of a solution to I.
- For every fixed apex-minor-free class of graphs $\mathcal{C}$, parameter $\varepsilon > 0$, and integers s, d, we can maintain a fully dynamic n-vertex (s,d)-decent instance I of Generalized Domination. We require that after each update, the Gaifman graph of I should belong to $\mathcal{C}$. Then after each update, the data structure can output a nonnegative real p satisfying $\mathsf{OPT}_I \leqslant p \leqslant (1+\varepsilon)\mathsf{OPT}_I$, where OPT_I is the minimum cost of a solution to I.

Tracing the time complexity analyses in Section 5.3.5 and Section 5.4.5, we can show that both data structures process each update in amortized $n^{o(1)}$ time.

To the best of our knowledge, Theorem 1.3.7 presents the first nontrivial data structure for approximation schemes on planar graphs in a fully dynamic setting. Hence, we hope that it may open multiple new avenues for future research. Here are three questions that immediately come to mind:

- Theorem 1.3.7 applies only to apex-minor-free classes; can it be extended to H-minor-free classes for any fixed H? The main obstacle here is that we are not aware of any approach that yields a (static) EPTAS for the considered problems in H-minor-free graphs with a near-linear running time dependency on graph size; and this would be implied by such an extension.
- Theorem 1.3.7 tackles only the Maximum Weight Independent Set and Minimum Weight Dominating Set problems; can it be extended to other problems amenable to Baker's technique, for instance the first-order expressible optimization problems considered in [DGKS06, Dvo22]? As the reader will see, the proof of Theorem 1.3.7 is considerably more difficult and delicate than that of the basic Baker's technique. At this point, even extending the result for Minimum Weight Dominating Set beyond the regime of bounded-degree graphs seems unclear.
- The $n^{o(1)}$ factor in the bound on the update time of our data structure is actually $n^{\mathcal{O}\left(\frac{\log\log\log n}{\sqrt{\log\log n}}\right)}$. This means that the amortized update time is indeed subpolynomial, but barely. It would be interesting to obtain better bounds – ideally polylogarithmic in n, but even an update time of the form $2^{\sqrt{\log n}(\log\log n)^{\mathcal{O}(1)}}$, similar to the bounds in Chapters 3 and 4, would be welcome.

  Interestingly, in the proof of Theorem 1.3.7 we do not use the data structure for dynamic treewidth, even though we extensively use various structural and algorithmic properties of tree decompositions in our dynamic variant of Baker's technique. It would be interesting to see if the approach presented here can be combined with the advances of Chapter 3, possibly yielding improved guarantees on the update time of the data structure.
- *Deamortization* of the presented data structures – that is, designing data structures for the problems considered in this chapter with worst-case guarantees – is another interesting question. This is subject of a work in progress.

Part II

Twin-width

Chapter 6

Compact oracle for d-twin-ordered matrices

In this chapter we design a compact representation for d-twin-ordered matrices that simultaneously occupies $\mathcal{O}_d(n)$ bits and offers query time $\mathcal{O}_d(\log \log n)$. We recall this result below.

Theorem 1.3.8 ([PSZ22]). *Let $d \in \mathbb{N}$ be a fixed constant. Then for a given binary $n \times n$ matrix M that is d-twin-ordered one can construct a data structure that occupies $\mathcal{O}_d(n)$ bits and can be queried for entries of M in worst-case time $\mathcal{O}(\log \log n)$ per query. The construction time is $\mathcal{O}_d(n \log n \log \log n)$ in the Word RAM model, assuming M is given by specifying $\ell = \mathcal{O}_d(n)$ rectangles $R_1, \ldots, R_\ell$ that form a partition of symbols 1 in M.*

Note that by Lemma 2.5.2, every d-twin-ordered $n \times n$ matrix can be represented as a union of $\mathcal{O}_d(n)$ rectangles that form a partition of symbols 1 in the matrix.

The proof of Theorem 1.3.8 proceeds roughly as follows. Consider a parameter m that divides n and a partition of the given matrix M into $(n/m)^2$ *zones* – square submatrices – each of which is induced by m consecutive rows and m consecutive columns. Such a partition is called the *regular (n/m)-division*. Even though the total number of zones in the regular (n/m)-division is $(n/m)^2$, one can use the connections between the notions of being twin-ordered and that of mixed minors (see the Preliminaries in Chapter 2), to show that actually there will be only $\mathcal{O}_d(n/m)$ *different* zones (Lemma 6.1.2), in the sense that zones are considered equal if they have exactly the same values in corresponding entries.

Our data structure describes the zones in the regular (n/m)-divisions of M for m ranging over a sequence of parameters $m_0 > m_1 > \ldots > m_\ell$ for $\ell = \mathcal{O}(\log \log n)$, where m_j divides m_i whenever $i \leqslant j$. Roughly speaking, we set $m_0 = n$ and $m_i = m_{i-1}^{2/3}$ for $i \geqslant 1$, though for technical reasons we resort to the recursion $m_i = m_{i-1}/2$ once m_i reaches the magnitude of $\log^3 n$. Each different zone present in the regular (n/m_i)-division is represented by a square matrix consisting of $(m_i/m_{i+1})^2$ pointers to representations of its subzones in the regular (n/m_{i+1})-division. When we reach $m_i < c_d \cdot \log n$ for some small constant c_d depending on d, we stop the construction and set $\ell = i$. At this point the number of different zones present in the regular (n/m_ℓ)-division of M is strongly sublinear in n, because we have such an upper bound on the total number of different $(c_d \log n) \times (c_d \log n)$ binary matrices that are d-twin-ordered, and $n/m_\ell \leqslant c_d \log n$. Therefore, all those matrices can be stored in the representation explicitly within bitsize $\mathcal{O}_d(n)$.

The query algorithm is very simple: We follow appropriate pointers through the $\mathcal{O}(\log \log n)$ levels of the data structure and read the relevant entry in a matrix stored explicitly in the last level. The analysis of bitsize is somewhat more complicated, but crucially relies on the fact that in the ith level, it suffices to represent only $\mathcal{O}_d(n/m_i)$ different matrices that are zones in the (n/m_i)-division.

We remark that the idea of dividing the given matrix into a number of polynomially smaller zones, and describing them recursively, is also the cornerstone of the approach used by Chan for the orthogonal point location problem in [Cha13]. However, when it comes to details, his construction is quite different and technically more complicated. For instance, in [Cha13] the recursion can be applied not only on single zones, but also on wide or tall strips consisting of several zones, or even submatrices induced by noncontiguous subsets of rows and columns. The conceptual simplification achieved here comes from the strong properties implied by the assumption that the matrix is d-twin-ordered, which is stronger than the assumption used by Chan that the symbols 1 in the matrix can be partitioned into $\mathcal{O}(n)$ rectangles.

Organization of the chapter. In Section 6.1, we prove several new structural properties of d-twin-ordered matrices. These properties are exploited in Section 6.2 to derive an efficient and compact representation of d-twin-ordered matrices, completing the nonconstructive part of Theorem 1.3.8. The efficient algorithm for construction of the data structure is then given in Section 6.3.

At the end of the chapter, in Section 6.4, we sketch a noncompact analog of the data structure of Theorem 1.3.8. We find that for every fixed $\varepsilon > 0$, we can construct a data structure representing a d-twin-ordered $n \times n$ matrix that requires bitsize $\mathcal{O}_d(n^{1+\varepsilon})$ and has worst-case query time $\mathcal{O}(1/\varepsilon)$.

6.1 Structural properties of divisions

Before we proceed to construct the promised compact representation, we need to describe some new combinatorial properties of twin-ordered matrices. For the remainder of this section, we fix $d \in \mathbb{N}$ and consider a matrix M that is d-twin-ordered. In particular, by Theorem 2.5.5, M is $(2d+2)$-mixed-free.

Strips. We begin by considering nonconstant vertical and horizontal zones of a given division of M. We will show that these zones can be grouped into $\mathcal{O}_d(t)$ *strips* that again are vertical or horizontal, respectively. This partitioning is formalized as follows.

Definition 14. *Let $(\mathcal{R}, \mathcal{C})$ be a division of a matrix M. A* vertical strip *in $(\mathcal{R}, \mathcal{C})$ is an inclusion-wise maximal set of nonconstant vertical zones of $\mathcal{D}$ that are contained in the same column block of $(\mathcal{R}, \mathcal{C})$, span a contiguous interval of row blocks, and whose union is again a vertical submatrix.* Horizontal strips *are defined analogously.*

1	1	1	1	1	1	1	0	0	1	1
0	0	0	0	0	0	0	0	0	1	1
1	1	1	1	1	1	1	0	0	1	1
1	1	1	1	1	1	1	0	0	1	1
0	0	0	1	1	1	1	0	0	0	1
0	0	0	0	0	1	0	0	0	0	1
1	1	1	1	1	1	1	0	0	0	1
0	0	0	0	0	1	0	0	0	0	1

Figure 6.1: Strips in an example 4-division of a matrix. Horizontal strips are painted in shades of yellow. Vertical strips are painted in shades of blue. Unpainted zones are constant or mixed.

Naturally, each nonconstant vertical zone belongs to exactly one vertical strip; and similarly, each nonconstant horizontal zone belongs to exactly one horizontal strip.

We will now show an upper bound on the number of vertical and horizontal strips present in any t-division of M.

Lemma 6.1.1. *For every $t \in \mathbb{N}$, the total number of vertical and horizontal strips in any t-division of M is at most $\mathcal{O}_d(t)$.*

Proof. We focus on the bound for vertical strips only; the proof for horizontal strips is symmetric. Fix some t-division $(\mathcal{R}, \mathcal{C})$ of M. Observe that each vertical strip S of the division either intersects the top row of the matrix, or the top-most zone of S is adjacent from the top to another zone C such that adding C to S yields a submatrix that is not vertical. (We say that C is adjacent to S *from the top*.) Thus, we partition the family of vertical strips in the t-division of M into three types:

(I) strips intersecting the top row of M;

(II) strips adjacent to a mixed zone C from the top; and

(III) strips adjacent to a nonmixed zone C from the top.

Obviously, there are at most t vertical strips of type (I). Next, each vertical strip of type (II) can be assigned a private mixed zone C adjacent to it from the top. Hence, the number of vertical strips of this type is upper bounded by the number of mixed zones in $(\mathcal{R}, \mathcal{C})$, which by Lemma 2.5.6 is bounded by $\mathcal{O}_d(t)$.

Finally, let us consider vertical strips of type (III). Let S be a vertical strip of this type, D be its top-most zone, and C be the nonmixed zone adjacent to D from the top. Since D is vertical, all rows of D are repetitions of the same row vector v_D. Since D is nonconstant, v_D is nonconstant as well.

As C is nonmixed, it is either horizontal or vertical. If C is vertical, then all its rows are repetitions of the same row vector v_C. Observe that since strip S could not be extended by C, we have $v_C \neq v_D$. Now, as v_D is nonconstant, it follows that the union of the bottom-most row of C and the top-most row of D contains a corner. On the other hand, if C is horizontal, then the bottom-most row of C is constant and again there is a corner in the union of the (constant) bottom-most row of C and the (nonconstant) top-most row of D.

So in both cases we conclude that C and D form a mixed cut. By Lemma 2.5.8, the total number of mixed cuts in $(\mathcal{R}, \mathcal{C})$ is bounded by $\mathcal{O}_d(t)$, so also there are at most $\mathcal{O}_d(t)$ vertical strips of type (III). This concludes the proof. □

Regular divisions. We move our focus to a central notion of our data structure: *regular divisions* of a matrix:

Definition 15. *Given M and an integer $s \in \mathbb{N}$, we define the* s-regular division *of M as the $\lceil \frac{n}{s} \rceil$-division of M in which each row block (respectively, column block), possibly except the last one, contains s rows (resp. columns). Precisely, if $s \nmid n$, then the last row block and the last column block contain exactly $n \bmod s$ rows or columns, respectively.*

In the data structure, given a square input matrix M, we will construct multiple regular divisions of M of varying granularity (the value of s). Crucially, in order to ensure the space efficiency of the data structure, we will require that the number of *distinct* zones in each such regular division of M should be small. This is facilitated by the following definition:

Definition 16. *For $s \in \mathbb{N}$, the* s-zone family *of M, denoted $\mathcal{F}_s(M)$, is the set of all different zones participating in the s-regular division of M.*

Let us stress that we treat $\mathcal{F}_s(M)$ as a set of matrices and do not keep duplicates in it. That is, if the regular s-division of M contains two or more isomorphic zones – with same dimensions and equal corresponding entries – then these zones are represented in $\mathcal{F}_s(M)$ only once.

For the remainder of this section, we will prove good bounds on the cardinality of $\mathcal{F}_s(M)$. Trivially, the cardinality of $\mathcal{F}_s(M)$ is bounded by $\lceil \frac{n}{s} \rceil^2$ (i.e., the number of zones in the s-regular division). Also, the same cardinality is trivially bounded by $2^{\mathcal{O}(s^2)}$ (i.e., the total number of distinct matrices with at most s rows and columns). However, given that M is d-twin-ordered, both bounds can be improved dramatically. First, the dependence on $\frac{n}{s}$ in the former bound can be improved to linear:

Lemma 6.1.2. *For every $s \in \{1, \ldots, n\}$, the cardinality of $\mathcal{F}_s(M)$ is bounded by $\mathcal{O}_d(\frac{n}{s})$.*

Proof. First assume that $s \mid n$; hence, each zone in the s-regular division of M has s rows and s columns. Then, the matrices in $\mathcal{F}_s(M)$ can be categorized into four types:

- Constant zones. There are at most 2 of them – constant 0 and constant 1.
- Mixed zones. Here, Lemma 2.5.6 applies directly: Since the considered division is an $\frac{n}{s}$-division of M, there are at most $\mathcal{O}_d(\frac{n}{s})$ mixed zones in M in total.
- Vertical zones. By Lemma 6.1.1, all vertical zones of the considered division can be partitioned into $\mathcal{O}_d(\frac{n}{s})$ vertical strips. As all zones have the same dimensions, the zones belonging to a single vertical strip are pairwise isomorphic. From this we infer the $\mathcal{O}_d(\frac{n}{s})$ upper bound on the number of different vertical zones.
- Horizontal zones are handled symmetrically to vertical zones.

Finally, if $s \nmid n$, then let M' be equal to M, truncated to the first $n - (n \bmod s)$ rows and columns; equivalently, M' is equal to M with all zones with fewer than s rows or columns removed. The argument given above applies to M', yielding at most $\mathcal{O}_d(\frac{n}{s})$ different $s \times s$ zones in M' (and equivalently in M). The proof is concluded by the observation that M contains exactly $2\lceil \frac{n}{s} \rceil - 1 = \mathcal{O}(\frac{n}{s})$ zones in its s-regular division that have fewer than s rows or columns. □

Second, from the works of Bonnet et al. [BGK+21a, BGdM+21] one can easily derive an upper bound that is exponential in s rather than in s^2:

Lemma 6.1.3. *For every $s \in \{1, \ldots, n\}$, the cardinality of $\mathcal{F}_s(M)$ is bounded by $2^{\mathcal{O}_d(s)}$.*

Proof. Observe that a submatrix of a d-twin-ordered matrix is also d-twin-ordered. Thus, it is only necessary to upper bound the total number of different $s \times s$ matrices that are d-twin-ordered. To this end, we use the notion of twin-width of ordered binary relational structures introduced in the work of Bonnet et al. [BGdM+21]. This notion is more general than twin-orderedness in the following sense: Each $s \times s$ matrix that is d-twin-ordered corresponds to a different ordered binary structure over s elements of twin-width at most d. As proved in [BGdM+21], the number of different such structures is upper bounded by $2^{\mathcal{O}_d(s)}$. The claim follows. □

While the bound postulated by Lemma 6.1.2 is more powerful for coarse regular divisions of M (i.e., s-regular divisions for large s), Lemma 6.1.3 yields a better bound for $s \leqslant p_d \cdot \log n$, where $p_d > 0$ is a sufficiently small constant depending on d.

6.2 Data structure

In this section we present the data structure promised in Theorem 1.3.8. Recall that it should represent a given binary $n \times n$ matrix M that is d-twin-ordered, and it should provide access to the following query: For given $(r, c) \in [n]^2$, return the entry $M[r, c]$. Here we focus only on the description of the data structure, implementation of the query, and analysis of the bitsize. The construction algorithm promised in Theorem 1.3.8 is given later, in Section 6.3.

Without loss of generality, we assume that n is a power of 2. Otherwise we enlarge M, so that its order is the smallest power of 2 larger than n. We use dummy 0s to fill additional entries. It is straightforward to see that the resulting matrix is $(d+1)$-twin-ordered. Similarly, in the analysis we may assume that n is sufficiently large compared to any constants present in the context.

Description. Our data structure consists of $\ell + 1$ layers: $\mathcal{L}_0, \ldots, \mathcal{L}_\ell$. Recall from Definition 16 that $\mathcal{F}_s(M)$ is the family of pairwise different zones participating in the s-regular division of M. Each layer $\mathcal{L}_i$ in our data structure corresponds to $\mathcal{F}_{m_i}(M)$ for a carefully chosen parameter m_i. Let $\mathsf{low}(x)$ be the largest power of 2 smaller or equal to x. We define parameters m_i inductively as follows: Set $m_0 = n$ and for $i \geqslant 0$,

$$m_{i+1} = \begin{cases} \mathsf{low}({m_i}^{2/3}) & \text{if } m_i \geqslant \log^3 n \\ m_i/2 & \text{if } \log n/(2\beta_d) \leqslant m_i < \log^3 n \end{cases}$$

where β_d is the constant hidden in the $\mathcal{O}_d(\cdot)$ notation in Lemma 6.1.3, i.e., $|\mathcal{F}_s(M)| \leqslant 2^{\beta_d \cdot s}$. The construction stops when we reach m_i satisfying $m_i < \log n/(2\beta_d)$, in which case we set $\ell = i$. Note that all parameters m_i are powers of 2, so m_j divides m_i whenever $i \leqslant j$.

We also observe the following.

Claim 6.2.1. $\ell \in \mathcal{O}(\log \log n)$.

Proof of the claim. Let k be the least index for which $m_k < \log^3 n$. Observe that for $i \in [1, k]$ we have $m_i \leqslant n^{(2/3)^i}$. So it must be that $k \leqslant \log_{3/2} \log n + 1 \in \mathcal{O}(\log \log n)$, for otherwise we would have $m_{k-1} \leqslant n^{(2/3)^{\log_{3/2} \log n}} = n^{1/\log n} = 2 < \log^3 n$. Next, observe that for $i \in [k+1, \ell]$ we have $m_i = m_k/2^{i-k}$. Therefore, we must have $\ell - k \leqslant \log(\log^3 n) + 1 \in \mathcal{O}(\log \log n)$, for otherwise we have $m_{\ell-1} \leqslant m_k/2^{\log \log^3 n} < \log^3 n/\log^3 n = 1$. The claim follows. ◁

Layer $\mathcal{L}_\ell$ is special and we describe it separately, so let us now describe the content of layer $\mathcal{L}_i$ for each $i < \ell$. Since n is divisible by m_i, every $Z \in \mathcal{F}_{m_i}(M)$ is a unique $m_i \times m_i$ matrix that appears at least once as a zone in the (n/m_i)-regular division of M. Such Z will be represented by an object $\mathsf{obj}(Z)$ in $\mathcal{L}_i$. Each object $\mathsf{obj}(Z)$ stores $(m_i/m_{i+1})^2$ pointers to objects in $\mathcal{L}_{i+1}$; recall here that m_{i+1} divides m_i. Consider the m_{i+1}-regular division of Z. This division consists of $(m_i/m_{i+1})^2$ zones; index them as $\mathsf{subzone}_Z(p, q)$ for $p, q \in [m_i/m_{i+1}]$ naturally. Observe that for all $p, q \in [m_i/m_{i+1}]$, it holds that $\mathsf{subzone}_Z(p, q) \in \mathcal{F}_{m_{i+1}}(M)$. In our data structure, each object $\mathsf{obj}(Z) \in \mathcal{L}_i$, corresponding to a matrix $Z \in \mathcal{F}_{m_i}(M)$, stores an array ptr of $(m_i/m_{i+1})^2$ pointers, where $\mathsf{ptr}[p, q]$ points to the address of $\mathsf{subzone}_Z(p, q)$ for all $p, q \in [m_i/m_{i+1}]$. This concludes the description of layer $\mathcal{L}_i$ for $i < \ell$.

We now describe layer $\mathcal{L}_\ell$. It is also a collection of objects, and for each matrix $Z \in \mathcal{F}_{m_\ell}(M)$ there is an object $\mathsf{obj}(Z) \in \mathcal{L}_\ell$; these objects are pointed to by objects from $\mathcal{L}_{\ell-1}$. However, instead of storing

further pointers, each object $\mathsf{obj}(Z) \in \mathcal{L}_\ell$ stores the entire matrix $Z \in \mathcal{F}_{m_\ell}(M)$ as a binary matrix of order $m_\ell \times m_\ell$, using m_ℓ^2 bits. This concludes the description of $\mathcal{L}_\ell$.

Observe that in $\mathcal{L}_0$ there is only one object corresponding to the entire matrix M. We store a global pointer ptrGlo to this object. Our data structure is accessed via ptrGlo upon each query.

Implementation of the query. The description of the data structure is now complete and we move on to describing how the query is executed. The query is implemented as method $\mathsf{entry}(r, c)$ and returns $M[r, c]$; see Algorithm 1 for the pseudocode (where $\mathsf{ptrIt} \rightarrow$ stands for dereference of a pointer ptrIt, i.e., the object pointed to by ptrIt). Given two integers $r, c \in [n]$, the method starts with pointer ptrGlo, and uses r and c and iterator pointer ptrIt to navigate via pointers down the layers, ending with a pointer to an object in layer $\mathcal{L}_\ell$. Initially, the iterator ptrIt is set to ptrGlo and it points to $\mathsf{obj}(Z)$ for the only matrix $Z \in \mathcal{F}_{m_0}(M)$. Integers r, c are the positions of the desired entry with respect to zone Z. After a number of iterations, ptrIt points to an object $\mathsf{obj}(Z) \in \mathcal{L}_i$ for a matrix $Z \in \mathcal{F}_{m_k}(M)$, and maintains current coordinates r and c. The invariant is that the desired output is the entry $Z[r, c]$. In one step of the iteration, the algorithm finds the matrix Z' in $\mathcal{F}_{m_{i+1}}(M)$ containing the desired entry $Z[r, c]$, which is the zone $\mathsf{subzone}_Z(r \mathbin{\mathsf{div}} m_{i+1}, c \mathbin{\mathsf{div}} m_{i+1}) \in \mathcal{F}_{m_{i+1}}(M)$, and moves the pointer ptrIt to $\mathsf{obj}(Z') \in \mathcal{L}_{i+1}$. The new coordinates of the desired entry with respect to Z' are $(r \mathbin{\mathsf{mod}} m_{i+1})$ and $(c \mathbin{\mathsf{mod}} m_{i+1})$, so r and c are altered accordingly. Once the iteration reaches $\mathcal{L}_\ell$, the object pointed to by ptrIt contains the entire zone explicitly, so it suffices to return the desired entry. Obviously, the running time of the query is $\mathcal{O}(\log \log n)$, since the algorithm iterates through $\ell \in \mathcal{O}(\log \log n)$ layers.

Algorithm 1: Query algorithm

Input : Integers $r, c \in [n]$
Output : $M[r, c]$

```
ptrIt ← ptrGlo
for i ← 0 to ℓ − 1 do
    ptrIt ← (ptrIt → ptr[r div m_{i+1}, c div m_{i+1}])
    r ← r mod m_{i+1}
    c ← c mod m_{i+1}
return ptrIt → Z[r, c]
```

Analysis of bitsize. We now analyze the number of bits occupied by the data structure. First note that the total number of objects stored is bounded by the total number of submatrices of M, which is polynomial in n. Hence, every pointer can be represented using $\mathcal{O}(\log n)$ bits. Keeping this in mind, the total bitsize occupied by the data structure is proportional to

$$\sum_{i=0}^{\ell-1} |\mathcal{F}_{m_i}(M)| \left(\frac{m_i}{m_{i+1}}\right)^2 \log n + |\mathcal{F}_{m_\ell}(M)| m_\ell^2, \tag{6.1}$$

This is because for all layers $\mathcal{L}_i$ for $i < \ell$ we store $|\mathcal{F}_{m_i}(M)|$ objects, each storing $\left(\frac{m_i}{m_{i+1}}\right)^2$ pointers, and in $\mathcal{L}_\ell$ we store $|\mathcal{F}_{m_\ell}(M)|$ objects, each storing a binary matrix of order $m_\ell \times m_\ell$.

We first bound the second term of Eq. (6.1). By Lemma 6.1.3, we have

$$|\mathcal{F}_{m_\ell}(M)| m_\ell^2 \leqslant 2^{\beta_d \cdot m_\ell} \cdot m_\ell^2 \leqslant 2^{\beta_d \cdot \frac{\log n}{2\beta_d}} \cdot \left(\frac{\log n}{2\beta_d}\right)^2 = \sqrt{n} \cdot \left(\frac{\log n}{2\beta_d}\right)^2 \in o(n).$$

We move on to bounding the first term of Eq. (6.1). Let k be the least index for which $m_k < \log^3 n$. We can split the first term of Eq. (6.1) into two sums:

$$\sum_{i=0}^{\ell-1} |\mathcal{F}_{m_i}(M)| \left(\frac{m_i}{m_{i+1}}\right)^2 \log n = \\ = \sum_{i=0}^{k-1} |\mathcal{F}_{m_i}(M)| \left(\frac{m_i}{m_{i+1}}\right)^2 \log n \tag{6.2}$$

$$+ \sum_{i=k}^{\ell-1} |\mathcal{F}_{m_i}(M)| \left(\frac{m_i}{m_{i+1}}\right)^2 \log n. \tag{6.3}$$

We first apply Lemma 6.1.2 to bound the sum (6.2). More precisely, if α_d is the constant hidden in the $\mathcal{O}_d(\cdot)$ notation in Lemma 6.1.2, we have

$$(6.2) \leqslant \log n \cdot \sum_{i=0}^{k-1} \alpha_d \frac{n}{m_i} \cdot 4m_i^{2/3} = 4\alpha_d n \log n \cdot \sum_{i=0}^{k-1} \frac{1}{m_i^{1/3}}. \tag{6.4}$$

Since for $i \in [k-1]$ we have $m_{i+1} = \mathsf{low}(m_i^{2/3})$ and $m_i \geqslant \log^3 n$, we have $m_i/m_{i+1} \geqslant 2$. Therefore $m_i \geqslant 2^{k-i-1} m_{k-1}$ for $i \in [0, k-1]$, so we can continue bounding the last expression in Eq. (6.4):

$$(6.4) \leqslant \alpha_d n \log n \sum_{i=0}^{k-1} \cdot \frac{1}{(2^{k-i-1} m_{k-1})^{1/3}} \leqslant \alpha_d n \cdot \frac{\log n}{m_{k-1}^{1/3}} \cdot \sum_{i=0}^{k-1} \frac{1}{(2^{k-i-1})^{1/3}} \in \mathcal{O}_d(n).$$

It remains to bound sum (6.3). We use Lemma 6.1.2 similarly as above:

$$(6.3) = 4 \log n \cdot \sum_{i=k}^{\ell-1} |\mathcal{F}_{m_i}(M)| \leqslant 4\alpha_d n \log n \cdot \sum_{i=k}^{\ell-1} \frac{1}{m_i} \leqslant 4\alpha_d n \log n \cdot \sum_{i=0}^{\infty} \frac{1}{(\frac{\log n}{2\beta_d}) \cdot 2^i} \in \mathcal{O}_d(n).$$

By summing up all the bounds we infer that the total number of bits occupied by our data structure is $\mathcal{O}_d(n)$.

6.3 Construction algorithm

In this section we complete the proof of Theorem 1.3.8 by presenting an algorithm that constructs the data structure described in Section 6.2 in time $\mathcal{O}_d(n \log n \log\log n)$. Here, we assume that the matrix M is specified on input by a rectangle decomposition $\mathcal{K}$ satisfying $|\mathcal{K}| \leqslant \mathcal{O}_d(n)$. We remark that the our construction algorithm will itself consume superlinear memory. In fact, even storing the input decomposition $\mathcal{K}$ requires $\Omega(n \log n)$ bits of memory. However, we stress that the data structure constructed by the algorithm occupies only $\mathcal{O}_d(n)$ bits.

The construction will proceed in three phases. First, in Section 6.3.1, we will set up a data structure that given a submatrix S of M, returns the *type* of S; that is, verifies whether S is constant, vertical, horizontal, or mixed. Next, in Section 6.3.2 we use the results of Section 6.3.1 to find an effective approximation $\mathcal{G}_s(M)$ of the zone families $\mathcal{F}_s(M)$. Finally, these effective approximations will be used in the construction of the data structure itself (Section 6.3.3).

6.3.1 Data structure for submatrix types

We will now define the announced subproblem formally. In Submatrix Types, we are given a rectangle decomposition $\mathcal{K}$ of an $n \times n$ matrix M, and we are required to preprocess it so as to handle the following queries efficiently: Given a submatrix S of M, return:

- *constant* c ($c \in \{0, 1\}$) if S is constant with c being the common entry;
- *horizontal* if S is nonconstant horizontal;
- *vertical* if S is nonconstant vertical; or
- *mixed* if S is mixed (i.e., neither horizontal nor vertical).

In this section, we prove the following:

Lemma 6.3.1. *Fix $d \in \mathbb{N}$ and assume M is a binary $n \times n$ matrix that is d-twin-ordered. Then there is a data structure for* Submatrix Types *on M that supports queries in worst-case time $\mathcal{O}_d(\log\log n)$ in the word RAM model. The data structure can be constructed in time $\mathcal{O}_d(n \log\log n)$, assuming M is represented on input by a rectangle decomposition $\mathcal{K}$ with $|\mathcal{K}| \leqslant \mathcal{O}_d(n)$.*

Observe that by restricting S to one-element matrices in Lemma 6.3.1, we will produce a data structure testing contents of individual entries of M in doubly-logarithmic time – the same as in the compact representation of M provided in Section 6.2. However, the data structure from Lemma 6.3.1 is by no means compact – in fact, its bitsize is $\mathcal{O}(n \log n \log\log n)$, which is even worse than the bitsize $\mathcal{O}(n \log n)$ achieved by the direct application of Chan's data structure for orthogonal point location [Cha13]. Thus, Submatrix Types can only be used as a building block of an algorithm constructing the compact representation of M.

In order to implement the data structure for Submatrix Types, we shall first define three auxiliary geometric problems. In each problem it can be assumed that each geometric object given on input has integer coordinates between 0 and $\mathcal{O}(n)$.

In Orthogonal Point Location, we are given a set of $\mathcal{O}(n)$ horizontal and vertical segments, where the segments may only intersect at their endpoints. The segments subdivide the plane into regions. In the problem, it is required to preprocess the regions and construct a data structure that can can efficiently locate the region containing a given query point. We will use the following data structure of Chan [Cha13] for this problem.

Theorem 6.3.2 ([Cha13])**.** *There is a data structure for* Orthogonal Point Location *that can answer each query in worst-case time $\mathcal{O}(\log\log n)$ and can be constructed in time $\mathcal{O}(n \log\log n)$.*

In Orthogonal Range Emptiness, we are given a set of $\mathcal{O}(n)$ points in the plane. It is required to preprocess the points in order construct a data structure that can efficiently find whether a queried axis-parallel rectangle contains any of the input points. In the positive case, it is not required to return any points: a yes/no answer suffices. For this problem, we will use the data structure of Chan et al. [CLP11].

Theorem 6.3.3 ([CLP11])**.** *There is a data structure for* Orthogonal Range Emptiness *that can answer each query in worst-case time $\mathcal{O}(\log\log n)$ and can be constructed in time $\mathcal{O}(n \log\log n)$.*

In Orthogonal Segment Intersection Emptiness, we are given a set of $\mathcal{O}(n)$ horizontal segments in the plane. It is required to preprocess the segments in order to construct a data structure that can efficiently decide whether a queried vertical segment intersects any of the horizontal segments. In the positive case, it is not required to return any segments: a yes/no answer suffices.

Theorem 6.3.4. *There is a data structure for* Orthogonal Segment Intersection Emptiness *that can answer each query in worst-case time $\mathcal{O}(\log\log n)$ and can be constructed in time $\mathcal{O}(n \log\log n)$.*

Proof. The problem admits a trivial reduction to the Vertical Ray Shooting problem, in which it is required to preprocess $\mathcal{O}(n)$ horizontal segments in order to construct a data structure that can find, for a given query point p, the lowest horizontal segment intersecting the vertical ray shooting upwards from p. Namely, a vertical segment pq intersects some horizontal input segment if and only if the lowest horizontal segment returned by an instance of Vertical Ray Shooting for query point p is different than the segment returned for query point q. As shown by Chan [Cha13], Vertical Ray Shooting is equivalent to Orthogonal Point Location, so we can use Theorem 6.3.2. □

We are now ready to give the data structure for Submatrix Types.

Proof of Lemma 6.3.1. We interpret M geometrically by representing the matrix as an $n \times n$ square in the plane, in which each entry corresponds to a single unit square. Let

$$\mathcal{A}(M) := \bigcup \{[c-1, c] \times [r-1, r] \mid M[r, c] = 1\}$$

be the area covered by the 1 entries of M in this interpretation. We remark that $\mathcal{A}(M)$ is an orthogonal subset of $[0, n]^2 \subseteq \mathbb{R}^2$. Equivalently, $\mathcal{A}(M)$ can be defined as the (interior-disjoint) union of the rectangles $[c_1 - 1, c_2] \times [r_1 - 1, r_2]$ for each submatrix $M[r_1 \ldots r_2, c_1 \ldots c_2] \in \mathcal{K}$. The boundary $\partial\mathcal{A}(M)$ of $\mathcal{A}(M)$ can be found in $\mathcal{O}_d(n)$ time by observing that a unit segment s with integral coordinates is a subset of $\partial\mathcal{A}(M)$ if and only if it belongs to the boundary of exactly one rectangle corresponding to a submatrix in $\mathcal{K}$.

For convenience, let $\widehat{\mathcal{A}}(M)$ denote the region $\mathcal{A}(M)$ with all coordinates doubled, and $\partial\widehat{\mathcal{A}}(M)$ denote the boundary of $\widehat{\mathcal{A}}(M)$.

We now use Theorem 6.3.2 to set up a data structure $\mathcal{I}_L$ for Orthogonal Point Location for $\partial\widehat{\mathcal{A}}(M)$. Then, given access to $\mathcal{I}_L$, we can verify in $\mathcal{O}(\log\log n)$ time whether $M[r,c] = 1$ for given $(r,c) \in [n]^2$ by querying $\mathcal{I}_L$ whether the point $(2c-1, 2r-1)$ belongs to some region that is a part of $\widehat{\mathcal{A}}(M)$. This, in turn, enables us to locate all corners of M. Indeed, observe that if C is a corner in M, then at least one of the 4 entries of C is a corner of some submatrix in $\mathcal{K}$. Hence, by iterating over all submatrices $M[r_1 \ldots r_2, c_1 \ldots c_2] \in \mathcal{K}$ and examining the neighborhood of each of the cells (r_1, c_1), (r_1, c_2), (r_2, c_1), (r_2, c_2) of M, we can find all corners in M. This takes $\mathcal{O}_d(n)$ queries to $\mathcal{I}_L$, and results in a maximum of $\mathcal{O}_d(n)$ corners in M (reiterating the statement of Lemma 2.5.7). Henceforth, let $\mathcal{B}$ be the set of those pairs (c,r) for which $\{M[r,c],\ M[r,c+1],\ M[r+1,c],\ M[r+1,c+1]\}$ is a corner in M.

Finally, let us consider a query about the type of a submatrix S of M. Say that $S = M[r_1 \ldots r_2, c_1 \ldots c_2]$, that is, S spans the block of rows from r_1 to r_2, inclusive, and the block of columns from c_1 to c_2, inclusive $(1 \leqslant r_1 \leqslant r_2 \leqslant n,\ 1 \leqslant c_1 \leqslant c_2 \leqslant n)$.

We first focus on deciding whether S is mixed or not. Recall from Lemma 2.5.3 that S is mixed if and only if it contains a corner. For this reason, we use Theorem 6.3.3 to set up a data structure $\mathcal{I}_E$ for Orthogonal Range Emptiness for $\mathcal{B}$; this takes time $\mathcal{O}_d(n \log\log n)$. Now, S is mixed if and only if $r_1 < r_2$, $c_1 < c_2$, and the rectangle $[c_1, c_2 - 1] \times [r_1, r_2 - 1] \subseteq \mathbb{R}^2$ covers any point in $\mathcal{B}$. This condition can be verified using $\mathcal{I}_E$ in time $\mathcal{O}_d(\log\log n)$.

From now on assume that S is not mixed. We will now decide whether S is vertical (possibly constant). This can be easily done using the following observation:

Claim 6.3.5. *Assume that S is not mixed. Then S is vertical if and only if the vertical segment s connecting the points $(2c_1 - 1, 2r_1 - 1)$ and $(2c_1 - 1, 2r_2 - 1)$ intersects no horizontal segments of $\partial\widehat{\mathcal{A}}(M)$.*

Proof of the claim. ($\Rightarrow$) If S is vertical, then $M[r_1, c_1] = M[r_1 + 1, c_1] = \cdots = M[r_2, c_1]$. Thus, the open rectangle $R \coloneqq (2(c_1 - 1), 2c_1) \times (2(r_1 - 1), 2r_2)$ is either fully contained within $\widehat{\mathcal{A}}(M)$ (if $M[r_1, c_1] = 1$), or is disjoint with $\widehat{\mathcal{A}}(M)$ (otherwise). Hence, R is disjoint with $\partial\widehat{\mathcal{A}}(M)$. Since $s \subseteq R$, the implication follows.

($\Leftarrow$) Suppose S is not vertical. Hence, it is horizontal and nonconstant, so there exists $r \in \{r_1, r_1 + 1, \ldots, r_2 - 1\}$ for which $M[r, c_1] \neq M[r+1, c_1]$. Define now the horizontal segment m connecting $(2(c_1 - 1), r)$ with $(2c_1, r)$. By $M[r, c_1] \neq M[r + 1, c_1]$ we have that $m \subseteq \partial\widehat{\mathcal{A}}(M)$; thus, m is a part of some horizontal segment m' of $\partial\widehat{\mathcal{A}}(M)$. Since m intersects s, so does m'. ◁

By Claim 6.3.5, we can determine whether S is vertical as follows. We use Theorem 6.3.4 to set up a data structure $\mathcal{I}_H$ for Orthogonal Segment Intersection Emptiness for the set of horizontal segments of $\partial\widehat{\mathcal{A}}(M)$. Since $\partial\widehat{\mathcal{A}}(M)$ consists of $\mathcal{O}_d(n)$ segments, $\mathcal{I}_H$ can be constructed in $\mathcal{O}_d(n \log\log n)$ time. Then, verifying whether S is vertical can be reduced to a single query on $\mathcal{I}_H$, which takes $\mathcal{O}_d(\log\log n)$ time. Using a symmetric data structure for vertical segments of $\partial\widehat{\mathcal{A}}(M)$, we can also verify whether S is horizontal. If S is both vertical and horizontal, then it is constant; in this case, a single call to $\mathcal{I}_L$ is enough to determine whether S is constant 0 or constant 1.

Summing up, the construction of the data structure takes $\mathcal{O}_d(n \log\log n)$ time, and each query requires time $\mathcal{O}_d(\log\log n)$ in the worst case. This concludes the proof. □

6.3.2 Efficient approximation of zone families

In this section, we use the findings of Section 6.3.1 to construct a concise representation of a given family of zones in the input matrix. Recall from Lemma 6.1.2 that for a d-twin-ordered $n \times n$ matrix M, its s-zone family $\mathcal{F}_s(M)$, defined as the set of distinct zones in the s-regular division of M, contains at most $\mathcal{O}_d(\frac{n}{s})$ submatrices. We shall now generalize this result: Given $s \in \mathbb{N}$ and access to M via an oracle for Submatrix Types, we will efficiently compute a subset $\mathcal{G}_s(M)$ of zones of the s-regular division of M that *represents* the s-zone family $\mathcal{F}_s(M)$ in the following sense: We require that every submatrix in $\mathcal{F}_s(M)$ should be represented by at least one zone in $\mathcal{G}_s(M)$ equal to this submatrix. The subset $\mathcal{G}_s(M)$ will still contain at most $\mathcal{O}_d(\frac{n}{s})$ submatrices; hence, it can be regarded as an efficient over-approximation of $\mathcal{F}_s(M)$. Moreover, we will give an effective mapping ξ, sending any zone of the s-regular division of M onto its representative in $\mathcal{G}_s(M)$.

Formally, assume that $s \mid n$. For $p, q \in [\frac{n}{s}]$, by $\mathsf{Zone}_s(p,q)$ we mean the zone of the s-regular division of M in the intersection of the p-th block of rows and the q-th block of columns of the division. Similarly, let $\mathsf{Zone}_s([p_1,p_2],[q_1,q_2]) \coloneqq \bigcup_{p=p_1}^{p_2} \bigcup_{q=q_1}^{q_2} \mathsf{Zone}_s(p,q)$. We shall prove the following observation:

Lemma 6.3.6. *Assume that an $n \times n$ matrix M is d-twin-ordered for a fixed $d \in \mathbb{N}$ and is given through an oracle $\mathcal{T}$ for* Submatrix Types *from Lemma 6.3.1. Then there exists an algorithm* Zone Approximation *which, given an integer $s \mid n$, computes:*

- *a set $\mathcal{G}_s(M) \subseteq \left[\frac{n}{s}\right]^2$ of size $\mathcal{O}_d(\frac{n}{s})$ and*
- *a mapping $\xi_s : \left[\frac{n}{s}\right]^2 \to \mathcal{G}_s(M)$,*

such that for each $p, q \in \left[\frac{n}{s}\right]$, if $(p', q') \coloneqq \xi(p, q)$ then $\mathsf{Zone}_s(p, q) = \mathsf{Zone}_s(p', q')$. Both $\mathcal{G}_s(M)$ and ξ_s are constructed by Zone Approximation *in time $\mathcal{O}_d(\frac{n}{s} \log \frac{n}{s} \log\log n)$. For given $(p, q) \in \left[\frac{n}{s}\right]^2$, the value $\xi_s(p, q)$ can be computed in time $\mathcal{O}_d(\log\log n)$.*

The remainder of this section is devoted to the proof of Lemma 6.3.6.

Sketch of the algorithm. In Zone Approximation, we implement the following strategy. First, create a partition $\mathcal{U}$ of the s-regular partition of M into $\mathcal{O}_d(\frac{n}{s})$ contiguous rectangular submatrices, each comprising pairwise equal zones. Then, form $\mathcal{G}_s(M)$ by picking one zone from each submatrix in $\mathcal{U}$. For the mapping ξ_s, we set up an instance of Orthogonal Point Location (Theorem 6.3.2). Given a query (p, q), we locate the rectangular submatrix of $\mathcal{U}$ containing $\mathsf{Zone}_s(p, q)$, and return the representative of this submatrix.

We consider the following submatrices for $\mathcal{U}$:

- individual mixed zones;
- separate strips (horizontal and vertical); and
- constant submatrices of M.

We now sketch how $\mathcal{U}$ is populated. Roughly speaking, the algorithm traverses all zones $\mathsf{Zone}_s(p, q)$ of the s-regular partition in the row-major order (in the increasing order of p, breaking ties in the increasing order of q). The algorithm will repeatedly choose the zone $Z = \mathsf{Zone}_s(p, q)$ outside of $\bigcup\mathcal{U}$ that is the earliest in the row-major order. Then, for some suitably chosen integers $p' \geqslant p$, $q' \geqslant q$, a new submatrix $\mathsf{Zone}_s(p \mathinner{..} p', q \mathinner{..} q')$, disjoint with $\bigcup\mathcal{U}$, will be created and added to $\mathcal{U}$. The new submatrix will have Z in its top-left corner.

Moreover, this process will at each step preserve the following invariant: Within each column block of the s-partition, $\mathcal{U}$ covers a prefix of zones with respect to the row order. Formally, if $\mathsf{Zone}_s(p, q)$ is a part of some submatrix of $\mathcal{U}$ for $p \geqslant 2$, then so is $\mathsf{Zone}_s(p-1, q)$. Indeed, adding $\mathsf{Zone}_s([p, p'], [q, q'])$ to $\mathcal{U}$ would break the invariant only if there existed an uncovered zone $\mathsf{Zone}_s(\bar{p}, \bar{q})$ for some $\bar{p} \in [p-1]$, $\bar{q} \in [q, q']$. However, by the choice of (p, q), all such zones already belong to $\mathcal{U}$.

Auxiliary data structure. In order to implement Zone Approximation, we first need to show an efficient way to find the earliest zone in the row-major order that is disjoint with $\bigcup\mathcal{U}$, under the aforementioned updates of $\mathcal{U}$:

Lemma 6.3.7. *Given $m \in \mathbb{N}$, we can construct a data structure maintaining an initially empty family $\mathcal{U}$ of pairwise disjoint subsets of $[m]^2$ under the following queries and updates:*

- $\mathsf{GetFirst}()$*: Returns the lexicographically smallest pair of integers $(p, q) \in [m]^2$ outside of $\bigcup\mathcal{U}$, or $\perp$ if no such pair exists;*
- $\mathsf{ExtendRight}()$*: Let $(p, q) \coloneqq \mathsf{GetFirst}()$. Returns the largest integer $q' \in [q, m]$ such that all elements $(p, q), (p, q+1), \ldots, (p, q')$ are disjoint from $\bigcup\mathcal{U}$;*
- $\mathsf{Cover}(p', q')$*: Let $(p, q) \coloneqq \mathsf{GetFirst}()$. Adds a rectangle $[p, p'] \times [q, q']$ as a new subset of $\mathcal{U}$; it is required that $p' \geqslant p$, $q' \geqslant q$, and the rectangle is disjoint with $\bigcup\mathcal{U}$.*

The data structure processes any query in time $\mathcal{O}(\log m)$.

Proof. Consider an array $H[1 \mathinner{..} m]$, where $H[p]$ ($p \in [m]$) is defined as the number of distinct $q \in [m]$ such that $(p, q) \in \bigcup\mathcal{U}$. By the invariant above, for any pair of integers $p, q \in [m]$, we have that $(p, q) \in \bigcup\mathcal{U}$ if and only if $q \leqslant H[p]$. The array will be maintained implicitly using a set $\mathcal{S}$ of triples (h, ℓ, r) of integers, denoting the maximal intervals of equal values in H. Formally, $(h, \ell, r) \in \mathcal{S}$ if and only if $\ell \leqslant r$,

$H[\ell] = H[\ell+1] = \ldots = H[r] = h$, and $H[\ell-1] \neq h$, $H[r+1] \neq h$ (we assume that $H[0] = H[m+1] = \infty$). Initially, $\mathcal{S} = \{(0, 1, m)\}$. The set also maintains the lexicographic order on the triples of integers, as well as a linked list that links the elements of $\mathcal{S}$ in the natural order in $[1 \ldots m]$ (that is, by increasing second, or equivalently third, coordinate). Thus, if $\mathcal{S}$ is implemented using a balanced binary search tree, such as an AVL tree, we can perform any update or query on $\mathcal{S}$ in worst-case $\mathcal{O}(\log m)$ time.

Given the representation of H through $\mathcal{S}$, answering queries $\mathsf{GetFirst}()$ and $\mathsf{ExtendRight}()$ is easy in $\mathcal{O}(\log m)$ time: Let (h, ℓ, r) be the lexicographically smallest element of $\mathcal{S}$. If $h = m$, we return $\perp$; otherwise, $\mathsf{GetFirst}() = (h+1, \ell)$ and $\mathsf{ExtendRight}() = r$. Now, consider $\mathsf{Cover}(p', q')$ for $p' \geqslant h+1$, $q' \geqslant \ell$. We must have that $q' \leqslant r$: By the choice of (h, ℓ, r), we know that $H[r+1] > H[r]$, so the new rectangle cannot extend past the rth column of $[m]^2$. Hence, we can update H through $\mathcal{S}$ by:

- removing (h, ℓ, r);
- adding (p', ℓ, q') back to $\mathcal{S}$; and if $q' < r$, also inserting $(h, q'+1, r)$ back to $\mathcal{S}$; and
- merging (p', ℓ, q') with the neighboring intervals in $\mathcal{S}$ if necessary, to ensure that $\mathcal{S}$ only keeps the maximal intervals of equal values in H.

This involves $\mathcal{O}(1)$ updates to $\mathcal{S}$. Thus, a single update requires $\mathcal{O}(\log m)$ time. □

We remark that Lemma 6.3.7 implements an auxiliary method $\mathsf{ExtendRight}()$. This method is not required to locate the earliest uncovered zone, but will be useful later in the algorithm.

Implementation of the algorithm. We now give the implementation of Zone Approximation. We set up an instance $\mathcal{P}$ of the data structure from Lemma 6.3.7 for $m = \frac{n}{s}$. In $\mathcal{P}$, each element $(p, q) \in [m]^2$ will correspond to the zone $\mathsf{Zone}_s(p, q)$ of the s-regular division. Also, recall that $\mathcal{T}$ is an instance of the data structure for Submatrix Types (Lemma 6.3.1) for the matrix M. We will populate $\mathcal{U}$ while maintaining the following invariants:

(I) within each column block of the s-partition, $\mathcal{U}$ covers a prefix of zones with respect to the row order; and

(II) each strip of the s-partition either is an element of $\mathcal{U}$, or is disjoint with $\bigcup \mathcal{U}$.

Note that we have already shown that invariant (I) is preserved throughout the algorithm.

Zone Approximation consists of a main loop which performs the following operations repeatedly: Let $(p, q) \coloneqq \mathsf{GetFirst}()$. Depending on the type of $\mathsf{Zone}_s(p, q)$, we will create a new rectangular submatrix S of M, disjoint with all elements of $\mathcal{U}$ so far, and add S to $\mathcal{U}$ by calling $\mathsf{Cover}(\cdot, \cdot)$. The loop is repeated until $\mathsf{GetFirst}() = \perp$.

Thus, assume that some submatrices have been already added to $\mathcal{U}$, and let integers $p, q, q_{\max}$ be so that $(p, q) = \mathsf{GetFirst}()$ and $q_{\max} = \mathsf{ExtendRight}()$. Let $Z \coloneqq \mathsf{Zone}_s(p, q)$. We find the type of Z by a single call to $\mathcal{T}$. What we do next depends on the type of the zone:

- *Mixed zone.* In this case, we simply add Z to $\mathcal{U}$ by calling $\mathsf{Cover}(p, q)$ and proceed to the next iteration of the loop.
- *Nonconstant vertical zone.* We perform a binary search to locate the largest integer $p' \in [p, \frac{n}{s}]$ for which the submatrix $Z' \coloneqq \mathsf{Zone}_s([p, p'], q)$ is vertical. This requires $\mathcal{O}(\log \frac{n}{s})$ calls to $\mathcal{T}$. Then we add Z' to $\mathcal{U}$ by calling $\mathsf{Cover}(p', q)$.
- *Nonconstant horizontal zone.* As above, use binary search to find the largest index $q' \in [q, q_{\max}]$ for which the submatrix $Z' \coloneqq \mathsf{Zone}_s(p, [q, q'])$ is horizontal. We add Z' to $\mathcal{U}$ by calling $\mathsf{Cover}(p, q')$.
- *Constant zone.* We use the same binary search as in the vertical case to locate the largest index $p' \in [p, \frac{n}{s}]$ such that the submatrix $Z' \coloneqq \mathsf{Zone}_s([p, p'], q)$ is constant. We then run another binary search to find the largest index $q' \in [q, q_{\max}]$ such that the submatrix $Z'' \coloneqq \mathsf{Zone}_s([p, p'], [q, q'])$ is constant. Then, we add Z'' to $\mathcal{U}$ by calling $\mathsf{Cover}(p', q')$.

It is easy to see that invariants (I) and (II) ensure that the new submatrix is disjoint with $\bigcup \mathcal{U}$. Then, invariant (II) guarantees that the zone Z in the vertical and horizontal cases is the earliest zone in the row-major order of the strip S containing Z, and S is disjoint with $\bigcup \mathcal{U}$. Thus, by the definition of a strip as a maximal vertical of horizontal submatrix, the presented binary search scheme will find the submatrix Z' equal to S. Adding the strip to $\mathcal{U}$ maintains the invariant.

After the main loop terminates, $\mathcal{U}$ is a partition of M into rectangular submatrices of M: mixed zones, strips, and a number of constant submatrices. For each submatrix, we locate its earliest zone $\mathsf{Zone}_s(p,q)$ in the row-major order, and we add (p,q) to $\mathcal{G}_s(M)$. Thus, $|\mathcal{G}_s(M)| = |\mathcal{U}|$. For ξ_s, observe that $\mathcal{U}$ is isomorphic to a subdivision of the square $[0,\frac{n}{s}]^2 \subseteq \mathbb{R}^2$ into rectangular regions, each corresponding to a single submatrix of $\mathcal{U}$. Thus, we instantiate an instance $\mathcal{I}_L$ of Orthogonal Point Location for this set of rectangles. Each query $\xi_s(p,q)$ is relayed to $\mathcal{I}_L$. The answer from $\mathcal{I}_L$ can be translated into a reference to the rectangular submatrix S of M containing $\mathsf{Zone}_s(p,q)$. The value of $\xi_s(p,q)$ can then be immediately deduced from S.

Analysis of the algorithm. First, we bound the number of iterations of the main loop:

Lemma 6.3.8. $|\mathcal{U}| \leqslant \mathcal{O}_d(\frac{n}{s})$.

Before we prove Lemma 6.3.8, let us verify that the time complexity of the algorithm Zone Approximation promised in the statement of Lemma 6.3.6 follows from it. The main loop of the algorithm runs $\mathcal{O}(|\mathcal{U}|) = \mathcal{O}_d(\frac{n}{s})$ times. Therefore, the oracle $\mathcal{T}$ is called $\mathcal{O}_d(\frac{n}{s}\log\frac{n}{s})$ times in our algorithm, requiring $\mathcal{O}_d(\frac{n}{s}\log\frac{n}{s}\log\log n)$ time in total. Next, the time complexity of all calls to $\mathcal{P}$ is bounded by $\mathcal{O}_d(\frac{n}{s}\log\frac{n}{s})$. Finally, the time complexity of the construction of $\mathcal{I}_L$ is bounded by $\mathcal{O}_d(\frac{n}{s}\log\log\frac{n}{s})$, and each call to ξ_s takes time $\mathcal{O}(\log\log\frac{n}{s})$. Thus, the time complexity analysis of Zone Approximation is complete; it remains to prove Lemma 6.3.8.

Proof of Lemma 6.3.8. In the regular s-division of M, there are at most $\mathcal{O}_d(\frac{n}{s})$ mixed zones (Lemma 2.5.6) and at most $\mathcal{O}_d(\frac{n}{s})$ strips (Lemma 6.1.1). It remains to bound the number of constant submatrices in $\mathcal{U}$ by $\mathcal{O}_d(\frac{n}{s})$.

We say that a constant submatrix $S = \mathsf{Zone}_s([p_1,p_2],[q_1,q_2])$ is *guarded* if S either touches the boundary of M (i.e., at least one of the equalities $p_1 = 1$, $q_1 = 1$, $p_2 = \frac{n}{s}$, or $q_2 = \frac{n}{s}$ holds), or the slightly larger submatrix $\mathsf{Zone}_s([p_1-1,\,p_2+1],\,[q_1-1,\,q_2+1])$, called the *shell* of S, is mixed.

Claim 6.3.9. *Every constant submatrix in $\mathcal{U}$ is guarded.*

Proof of the claim. Assume otherwise, and choose an unguarded constant submatrix $S \in \mathcal{U}$. Then, S is not incident to the boundary of M, and each zone adjacent to S (by a side or by a corner) is horizontal or vertical.

Suppose $S = \mathsf{Zone}_s([p_1,p_2],[q_1,q_2])$. Let $\widehat{S} \coloneqq \mathsf{Zone}_s([p_1-1,\,p_2+1],\,[q_1-1,\,q_2+1])$ be the shell of S. Without loss of generality, let 0 be the common entry in S.

Let $S_{\nwarrow}$ be the zone in the top left corner of S (i.e., the earliest zone of S in the row-major order), and $S_{\swarrow}$ be the zone in the bottom left corner of S. We also consider the following zones in $\widehat{S}$: $Y_{\uparrow}$, $Y_{\nwarrow}$, and $Y_{\leftarrow}$, adjacent to $S_{\nwarrow}$ from the top, top left, and left, respectively; and $Y_{\downarrow}$ and $Y_{\swarrow}$, adjacent to $S_{\swarrow}$ from the bottom and bottom left, respectively (Figure 6.2).

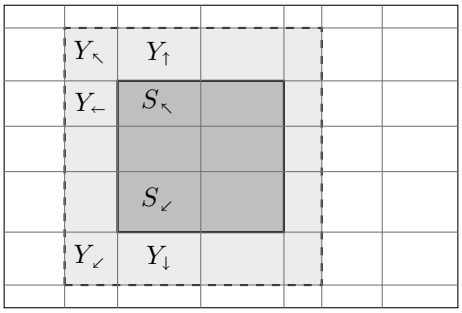


Figure 6.2: An example constant submatrix S (dark gray). The zones in $\widehat{S} \setminus S$ are marked light gray. In this figure, the first row block is at the top, and the first column block is on the left.

Consider the zone $Y_{\downarrow}$. As $\widehat{S}$ is not mixed, $Y_{\downarrow}$ is not mixed either. Moreover, by the construction of S, the submatrix $\mathsf{Zone}_s([p_1,p_2+1],q_1)$ is not constant, so the zone $Y_{\downarrow}$ is not constant 0. Also, $Y_{\downarrow}$ cannot be nonconstant vertical; otherwise, a mixed cut would appear on the boundary between $S_{\swarrow}$ and $Y_{\downarrow}$, and $\widehat{S}$ would be mixed. Hence, $Y_{\downarrow}$ is either nonconstant horizontal or constant 1. It can be now easily verified that the entire shell $\widehat{S}$ is nonconstant horizontal. It immediately follows that: $Y_{\leftarrow}$ is constant 0; both $Y_{\swarrow}$

and $Y_\downarrow$ are horizontal (but not constant 0) and repeat the same column vector; and both $Y_\nwarrow$ and $Y_\uparrow$ are horizontal (possibly constant) and also repeat the same column vector.

Since the elements of $\mathcal{U}$ are rectangular, either $Y_\uparrow$ or $Y_\leftarrow$ must belong to a different element of $\mathcal{U}$ than $Y_\nwarrow$. We perform a case analysis depending on whether $Y_\leftarrow$ and $Y_\nwarrow$ are in one submatrix.

Case 1: $Y_\leftarrow$ is in a different submatrix $A \in \mathcal{U}$ than $Y_\nwarrow$. Then, $Y_\leftarrow$ is the zone in the top-right corner of A. Also, A is constant 0, since $Y_\leftarrow$ is constant 0. Moreover, $Y_\swarrow$ is not constant 0 and thus remains outside of A. Hence, the set of rows spanned by A is a subset of the set of rows spanned by S. But the top-left zone of A is earlier in the row-major order than the top-left zone of S, so A must have been added to $\mathcal{U}$ by the algorithm before S; and when A was being added to $\mathcal{U}$, all zones of S were outside of $\mathcal{U}$. Hence, the binary search scheme would have extended A more to the right, in particular covering $S_\nwarrow$ as a zone – a contradiction since $S_\nwarrow \notin A$.

Case 2: $Y_\leftarrow$ and $Y_\nwarrow$ are in the same submatrix $A \in \mathcal{U}$. Then $Y_\nwarrow$ is constant 0 (as $Y_\leftarrow$ is constant 0), and hence $Y_\uparrow$ is also constant 0 (as $\widehat{S}$ is horizontal). Also, $Y_\uparrow$ is the bottom-left zone of a different submatrix $B \in \mathcal{U}$. But again, B would have been added to $\mathcal{U}$ by the algorithm before S, and B would have extended downwards to cover $S_\nwarrow$ – a contradiction.

Since all cases have been exhausted, we conclude that S must be guarded. ◁

Claim 6.3.10. *$\mathcal{U}$ contains at most $\mathcal{O}_d(\frac{n}{s})$ constant submatrices.*

Proof of the claim. Obviously, there are at most $4 \cdot \frac{n}{s}$ different submatrices of $\mathcal{U}$ touching the boundary of M. Consider then a constant submatrix $S \in \mathcal{U}$ that does not touch the boundary of M. Let $S = \mathsf{Zone}_s([p_1, p_2], [q_1, q_2])$, and let $\widehat{S} := \mathsf{Zone}_s([p_1 - 1, p_2 + 1], [q_1 - 1, q_2 + 1])$ be the shell of S. By Claim 6.3.9, $\widehat{S}$ is mixed, so it contains a corner C. We consider three cases, depending on the location of C.

- If C is fully contained within some (mixed) zone Z, we assign S to Z.
- If C is split in halves by some (mixed) cut μ, we assign S to μ.
- If C is split by the zone boundaries into four 1×1 submatrices, we assign S to C.

As all submatrices of $\mathcal{U}$ are pairwise disjoint, each entry of M belongs to at most 9 shells of the submatrices of $\mathcal{U}$. In particular, each object (mixed zone, mixed cut or corner) belongs to at most 9 shells. It follows that each such object may be assigned to at most $\mathcal{O}(1)$ guarded submatrices. Since the s-regular division of M contains at most $\mathcal{O}_d(\frac{n}{s})$ mixed zones (Lemma 2.5.6), $\mathcal{O}_d(\frac{n}{s})$ mixed cuts (Lemma 2.5.8), and $\mathcal{O}_d(\frac{n}{s})$ split corners (also Lemma 2.5.8), we conclude that $\mathcal{U}$ contains at most $\mathcal{O}_d(\frac{n}{s})$ constant submatrices. ◁

As discussed, with Claim 6.3.10 established, the statement of the lemma is immediate. □

6.3.3 Construction algorithm for Theorem 1.3.8

We now combine the results of Sections 6.3.1 and 6.3.2 to construct the data structure described in Section 6.2. As promised, the construction will take time $\mathcal{O}_d(n \log n \log\log n)$, provided that the input matrix is given by specifying a rectangle decomposition $\mathcal{K}$ with $|\mathcal{K}| \leqslant \mathcal{O}_d(n)$. This will conclude the proof of Theorem 1.3.8.

First, we set up an instance $\mathcal{I}_T$ of the data structure for Submatrix Types on M. $\mathcal{I}_T$ can be instantiated in time $\mathcal{O}_d(n \log\log n)$ from $\mathcal{K}$ (Lemma 6.3.1). Recall that each access to $\mathcal{I}_T$ is realized in worst-case time $\mathcal{O}(\log\log n)$. From now on, we assume that M is accessed only through $\mathcal{I}_T$.

Recap of the data structure. In Section 6.2 we defined parameters $m_0 > m_1 > \ldots > m_\ell$ such that $m_0 = n$, $m_\ell \in \Theta_d(\log n)$, $\ell \in \mathcal{O}(\log\log n)$, and $m_{i+1} \mid m_i$ and $m_{i+1}^2 \geqslant m_i$ for each $i \in \{0, 1, \ldots, \ell - 1\}$. The data structure consists of $\ell + 1$ layers: $\mathcal{L}_0, \ldots, \mathcal{L}_\ell$. Each layer $\mathcal{L}_i$ contains one object $\mathsf{obj}(Z)$ for each element Z of the zone family $\mathcal{F}_{m_i}$. For $i < \ell$, each object $\mathsf{obj}(Z)$ in $\mathcal{L}_i$ contains an $\frac{m_i}{m_{i+1}} \times \frac{m_i}{m_{i+1}}$ array ptr of pointers to the objects in $\mathcal{L}_{i+1}$, corresponding to the elements of the regular m_{i+1}-division of Z. For $i = \ell$, each object $\mathsf{obj}(Z)$ in $\mathcal{L}_\ell$ stores the entire submatrix Z using m_ℓ^2 bits. By carefully choosing the parameters, we guarantee that the data structure occupies $\mathcal{O}_d(n)$ bits.

Strategy. Assume that the suitable parameters $m_0, \ldots, m_\ell$ have already been selected. We construct the data structure bottom-up, starting from layer $\mathcal{L}_\ell$, and concluding with layer $\mathcal{L}_0$. For each $i = \ell, \ell-1, \ldots, 0$, we construct a set $\mathcal{G}_{m_i}(M)$ of representative zones in the regular m_i-division of M; and a mapping $\xi_{m_i} : \left[\frac{n}{m_i}\right]^2 \to \mathcal{G}_{m_i}(M)$, sending any zone of the m_i-regular division of M onto their representative in $\mathcal{G}_{m_i}(M)$. Since $m_i \in \Omega_d(\log n)$, this construction will take time $\mathcal{O}_d(n \log\log n)$ for each i. Moreover, $|\mathcal{G}_{m_i}(M)| \in \mathcal{O}_d(\frac{n}{m_i})$, and the mapping ξ_{m_i} can be evaluated on any zone in $\mathcal{O}(\log\log n)$ time. Next, we construct $\mathcal{F}_{m_i}(M)$ by filtering out identical matrices from $\mathcal{G}_{m_i}(M)$; formally, we construct a surjection ψ_{m_i} mapping $\mathcal{G}_{m_i}(M)$ onto the set of objects in $\mathcal{L}_i$ such that $\psi_i(p,q) = \psi_i(p',q')$ if and only if $\mathsf{Zone}_{m_i}(p,q) = \mathsf{Zone}_{m_i}(p',q')$. For $i = \ell$, this will be done directly, by listing all entries in the zone; for $i < \ell$, this will be done by taking all representative zones and comparing the subzones in their m_{i+1}-regular divisions. Finally, the pointers from $\mathcal{L}_i$ to $\mathcal{L}_{i+1}$ will be derived from the mapping $\xi_{m_{i+1}}$.

The bottom layer $\mathcal{L}_\ell$. We begin by constructing $\mathcal{L}_\ell$. Using Lemma 6.3.6, we find $\mathcal{G}_{m_\ell}(M)$ and ξ_{m_ℓ}. Next, for each representative $(p,q) \in \mathcal{G}_{m_\ell}(M)$, we examine each individual entry in $\mathsf{Zone}_s(p,q)$ using m_ℓ^2 queries to $\mathcal{I}_L$. This requires $\mathcal{O}_d(n \log n)$ queries in total for all elements of $\mathcal{G}_{m_\ell}(M)$, resulting in time $\mathcal{O}_d(n \log n \log\log n)$. Thus, each representative zone is now fully described by a bitvector of length $m_\ell^2 \in \mathcal{O}_d(\log^2 n)$; and two representative zones $\mathsf{Zone}_{m_\ell}(p,q)$, $\mathsf{Zone}_{m_\ell}(p',q')$ are equal if and only if their corresponding bitvectors are equal. The bitvectors can be sorted using radix sort in time $\mathcal{O}_d(\frac{n}{\log n} \cdot \log^2 n) = \mathcal{O}_d(n \log n)$. Then, the zones can be grouped into equivalence classes with respect to their equality; each such class corresponds to one zone in the zone family $\mathcal{F}_{m_\ell}(M)$. Eventually, we pick one matrix from each equivalence class and store it in its entirety as an object of $\mathcal{L}_\ell$.

This concludes the construction of $\mathcal{L}_\ell$. The time complexity is $\mathcal{O}_d(n \log n \log\log n)$, dominated by querying $\mathcal{I}_L$ for individual elements of M. The choice of a matrix from each class induces the surjection ψ_{m_ℓ}.

Layers $\mathcal{L}_i$ for $i < \ell$. Assume that the layer $\mathcal{L}_{i+1}$ has already been constructed, together with the auxiliary set $\mathcal{G}_{m_{i+1}}(M)$ and functions $\xi_{m_{i+1}}$ and $\psi_{m_{i+1}}$. By Lemma 6.3.6, we find $\mathcal{G}_{m_i}(M)$ and ξ_{m_i}.

We enumerate the objects in $\mathcal{L}_{i+1}$ as $A_1, A_2, \ldots, A_s$, where $s = |\mathcal{F}_{m_{i+1}}(M)| \in \mathcal{O}_d(\frac{n}{m_{i+1}})$. Recall that each A_j corresponds to a different matrix in the zone family $\mathcal{F}_{m_{i+1}}$. Then, take some $(a,b) \in \mathcal{G}_{m_i}(M)$ and let $Z = \mathsf{Zone}_{m_i}(a,b)$ denote the corresponding representative zone in M. We list all subzones $\mathsf{subzone}_Z(p,q)$ $(p,q \in [m_i/m_{i+1}])$ in the regular m_{i+1}-division of Z and interpret each of them as an element of $\mathcal{F}_{m_{i+1}}(M)$. Since $\mathsf{subzone}_Z(p,q) = \mathsf{Zone}_{m_{i+1}}((a-1)\frac{m_i}{m_{i+1}} + p, (b-1)\frac{m_i}{m_{i+1}} + q)$, we can find the unique element $j \in [s]$ such that $A_j = \mathsf{obj}(\mathsf{subzone}_Z(p,q))$ in $\mathcal{O}(\log\log n)$ time by locating the representative zone of $\mathsf{subzone}_Z(p,q)$ in the m_{i+1}-regular division of M using $\xi_{m_{i+1}}$, and then using $\psi_{m_{i+1}}$ to find the corresponding object in $\mathcal{L}_{i+1}$. This way, we describe each representative zone Z of the regular m_i-division of M as an $\frac{m_i}{m_{i+1}} \times \frac{m_i}{m_{i+1}}$ square matrix $\mathcal{D}(Z)$ of elements from 1 to s; again, two representative zones Z_1, Z_2 are equal to each other if and only if their descriptions are equal. As $|\mathcal{G}_{m_i}| \in \mathcal{O}_d(\frac{n}{m_i})$ and we spend $(m_i/m_{i+1})^2$ calls to $\xi_{m_{i+1}}$ for each zone in $\mathcal{G}_{m_i}$, this in total requires time $\mathcal{O}_d(\frac{n}{m_i} \cdot \left(\frac{m_i}{m_{i+1}}\right)^2 \cdot \log\log n)$, which by $m_{i+1}^2 \geqslant m_i$ is bounded by $\mathcal{O}_d(n \log\log n)$.

We now filter the repeated occurrences of the descriptions of the representative zones in $\mathcal{G}_{m_i}$. We do it by sorting the descriptions in the lexicographic row-major order using any comparison sort, where in each comparison we simply compare two arrays of length $(m_i/m_{i+1})^2$, and then grouping equal descriptions. This takes time $\mathcal{O}_d(\frac{n}{m_i} \cdot \log \frac{n}{m_i} \cdot \left(\frac{m_i}{m_{i+1}}\right)^2)$, which is bounded by $\mathcal{O}_d(n \log n)$. Afterwards, we pick one representative from each equivalence class and store it as an object in $\mathcal{L}_i$. Again, each object of $\mathcal{L}_i$ corresponds to a single zone in the zone family $\mathcal{F}_{m_i}$, and the construction above naturally gives rise to the surjection ψ_{m_i}. Given an object $\mathsf{obj}(Z) \in \mathcal{L}_i$, the pointers from $\mathsf{obj}(Z)$ to the objects of $\mathcal{L}_{i+1}$ can be immediately deduced from $\mathcal{D}(Z)$ and the sequence $A_1, A_2, \ldots, A_s$.

Summary. We constructed the bottom-most layer $\mathcal{L}_\ell$ in time $\mathcal{O}_d(n \log n \log\log n)$. For each $i = \ell - 1, \ell - 2, \ldots, 0$, the construction of $\mathcal{L}_i$ takes time $\mathcal{O}_d(n \log n)$, dominated by the comparison sort of the descriptions of the zones. Since $\ell \in \mathcal{O}(\log\log n)$, we conclude that the time complexity of the entire construction is $\mathcal{O}_d(n \log n \log\log n)$. Therefore, the constructive part of Theorem 1.3.8 is proved.

6.4 Representation with bitsize $\mathcal{O}(n^{1+\varepsilon})$ and query time $\mathcal{O}(1/\varepsilon)$

In this section we provide a brief sketch of another data structure representing twin-ordered matrices. For any fixed $\varepsilon > 0$, we will construct a data structure that represents a given d-twin-ordered $n \times n$ matrix M in bitsize $\mathcal{O}(n^{1+\varepsilon})$, and can be queried for entries of M in worst-case time $\mathcal{O}(1/\varepsilon)$ per query. Actually, the data structure solves the Orthogonal Point Location problem within the same space and time bounds, provided that the input is given as a set of orthogonal rectangles with pairwise disjoint interiors, and with integer coordinates between 0 and n. As the set of 1 entries in any d-twin-ordered matrix M admits a rectangle decomposition into $\mathcal{O}_d(n)$ rectangles (Lemma 2.5.2), this also yields a data structure representing M.

Notably, Chan [Cha13] observed that Orthogonal Point Location can be reduced to the static variant of the Predecessor Search problem, even if the input coordinates are from 0 to $\mathcal{O}(n)$. Pătraşcu and Thorup proved that each data structure for Predecessor Search with $\mathcal{O}(n \log^{\mathcal{O}(1)} n)$ bitsize necessarily requires $\Omega(\log \log n)$ query time, even in a much more powerful cell probe model [PT06]. Therefore, for general Orthogonal Point Location, one cannot expect to achieve constant query time with bitsize significantly smaller than $\mathcal{O}(n^{1+\varepsilon})$.

Data structure for disjoint intervals. Consider integers $k, h \geqslant 1$, and let $n = k^h$. We will first sketch a data structure that maintains a set of disjoint integer intervals that are subintervals of $[0, n-1]$. The data structure shall allow adding or removing intervals in time $\mathcal{O}(kh)$ and querying whether a point is contained in any interval in time $\mathcal{O}(h)$.

Consider a perfect k-ary tree of depth h. The tree has k^h leaves, numbered from 0 to $n-1$ according to the pre-order traversal of the tree. Each internal node at depth $i \in \{0, 1, \ldots, h-1\}$ in the tree corresponds to a contiguous interval of leaves of length k^{h-i}. Each such interval is called a *base interval*. Each internal node contains an array of k pointers to the children in the tree, allowing access to the j-th child in constant time. Additionally, alongside each node v of the data structure, we store an additional bit b_v, initially set to 0.

Assume an interval $[\ell, r]$ is to be inserted to the set. We traverse the tree recursively, starting from the root, entering only nodes whose base intervals intersect $[\ell, r]$, and cutting the recursion at nodes whose base intervals are entirely within $[\ell, r]$. It can be shown that the recursion visits at most $\mathcal{O}(kh)$ nodes and decomposes $[\ell, r]$ into $\mathcal{O}(kh)$ disjoint base intervals. For each node v corresponding to such a base interval, we set $b_v \leftarrow 1$. Removing an interval from the set is analogous. Now, to verify whether an element y belongs to the set, we descend recursively from the root of the tree to the y-th leaf of the tree and verify if any of the visited nodes v has $b_v = 1$. This requires time $\mathcal{O}(h)$.

Since each update and query to the data structure is essentially a recursive search from the root of the tree, the data structure can be made persistent: On each update, we create a copy of each altered node and each of their ancestors, and we reset the pointers in the copies accordingly. As $\mathcal{O}(kh)$ nodes are updated at each query, and each internal node stores an array of $\mathcal{O}(k)$ pointers, the update time increases to $\mathcal{O}(k^2 h)$ due to the copying of the nodes; and each update increases the bitsize of the data structure by $\mathcal{O}(k^2 h \log n)$. Thus, after $\mathcal{O}_d(n)$ updates, the bitsize of the data structure is $\mathcal{O}_d(nk^2 h \log n)$. The query time remains at $\mathcal{O}(h)$.

Orthogonal point location with small coordinates. Fix any $\varepsilon > 0$. Given a matrix M of order n, we set $h := \lceil 2/\varepsilon \rceil + 1$ and $k := \lceil n^{1/h} \rceil$. We instantiate a persistent k-ary tree of depth h as above. We sweep the set of rectangles from the left of the right, maintaining a vertical sweep line. The tree maintains an intersection of the sweep line with the union of rectangles as a set of disjoint intervals contained in $[0, n]$. Hence, for each rectangle, the tree is updated twice: a vertical interval is added when the sweep line reaches the left end of the rectangle, and is removed as soon as it reaches the right end of the rectangle. At each x coordinate, we store the pointer ver_x to the root of the current version of the tree. After the preprocessing, for each query (x, y), we fetch the pointer ver_x and check whether this version of the tree contains y as an element.

Let us analyze the query time and the bitsize of the data structure. For convenience, let $\delta := 1/h$. We can see that $0 < \delta < \frac{\varepsilon}{2}$. Each query is performed in time $\mathcal{O}(h) = \mathcal{O}(1/\varepsilon)$. Storing pointers ver_x requires bitsize $\mathcal{O}(n \log n)$. Since we processed $\mathcal{O}_d(n)$ rectangles, the persistent tree has bitsize $\mathcal{O}_d(nk^2 h \log n) = \mathcal{O}_d(n^{1+2\delta} \log n/\varepsilon) = \mathcal{O}_d(n^{1+\varepsilon})$.

6.5 Conclusions

We presented a compact data structure that can be queried for entries of a binary d-twin-ordered $n \times n$ matrix in worst-case $\mathcal{O}(\log \log n)$ time. Below we discuss various generalizations and variants of the problem, as well as present possible research directions related to the problem.

- Suppose we want to construct a compact adjacency oracle for a labeled graph of twin-width at most d – a data structure that can be queried whether any given pair of vertices of the graph is connected by an edge. For convenience assume that $V(G) = [n]$. By Observation 2.5.1 there is a total order $<$ of $[n]$ such that the adjacency matrix of G, with rows and columns ordered according to $<$, is $(d+2)$-twin-ordered; in particular, this is true whenever $<$ is consistent with some d-sequence (a contraction sequence of error value at most d). So if G has a d-sequence in which all parts are intervals of $[n]$, then the adjacency matrix of G, with rows and columns ordered according to the natural order on $[n]$, is $(d+2)$-twin-ordered. In such a case we can use the data structure of Theorem 1.3.8 to store this adjacency matrix of G. This implies an adjacency oracle for G that occupies $\mathcal{O}_d(n)$ bits and that can be queried in worst-case time $\mathcal{O}(\log \log n)$ per query.

  However, we cannot hope for an adjacency oracle occupying $\mathcal{O}_d(n)$ for arbitrary labeled graphs of bounded twin-width. Consider for instance the n-vertex path graph P_n for $n \geqslant 2$. We can assign labels from $[n]$ to the vertices of P_n in $n!/2$ nonisomorphic ways, and each such assignment produces a labeled graph of twin-width at most 1. Adjacency oracles storing different labeled graphs must have different representations in memory, so any adjacency oracle that can store any labeled n-vertex path graph must necessarily occupy at least $\log(n!/2) \in \Omega(n \log n)$ bits.

- A similar lower bound applies to matrices of bounded twin-width: The class of permutation matrices (square binary matrices containing exactly one entry 1 in each row and each column) has bounded twin-width since the rows and the columns of each permutation matrix can be shuffled so as to produce the identity matrix. However, there are precisely $n!$ $n \times n$ permutation matrices, so an oracle that can store any such matrix must occupy at least $\Omega(n \log n)$ bits.

- It remains an interesting question whether the result of Observation 2.5.1 can be used to produce an adjacency oracle for ordered graphs of bounded twin-width with bitsize $\mathcal{O}_d(n)$ (see Section 1.1.3 for the definition of twin-width of ordered graphs), or an analogous oracle with bitsize $\mathcal{O}_k(n)$ for k-mixed-free $n \times n$ matrices.

- Our result easily generalizes to the setting of d-twin-ordered matrices over a constant alphabet: Fix a constant $d \in \mathbb{N}$ and a constant finite alphabet Σ. Then for a given $n \times n$ matrix M with entries from Σ that is d-twin-ordered (the notion of d-twin-ordered matrices naturally lifts to nonbinary matrices), we can construct a data structure that occupies $\mathcal{O}_{d,\Sigma}(n)$ bits and that can be queried for entries of M in worst-case time $\mathcal{O}_\Sigma(\log \log n)$ per query. To observe this, note that for every $c \in \Sigma$, the binary matrix M_c constructed from M by replacing all entries c with 1 and all the remaining entries with 0 is also d-twin-ordered. Hence a compact oracle for M can be produced by constructing compact oracles for each binary matrix M_c.

- As mentioned before, it is an open question whether the query time in Theorem 1.3.8 could be improved to constant – $\mathcal{O}_d(1)$ or perhaps even $\mathcal{O}(1)$. A promising approach that could yield such an improvement is to use the notion of *delayed decompositions* of graphs of bounded twin-width of Bourneuf and Thomassé [BT23], inspired by our research on χ-boundedness of graphs of bounded twin-width (Chapter 7). In Section 7.5 we overview this notion, as well as briefly discuss how it could be helpful in the design of compact oracles for d-twin-ordered matrices.

Chapter 7

Twin-width and χ-boundedness

In this chapter we present a proof that classes of graphs of bounded twin-width are *quasi-polynomially χ-bounded*, improving upon the exponential bound by Bonnet et al. [BGK+21b]. We recall the statement of the result below:

Theorem 1.3.10 ([PS23]). *For every $d \in \mathbb{N}$ there exists a constant $\gamma_d \in \mathbb{N}$ such that for every graph G of twin-width at most d and clique number ω, we have*

$$\chi(G) \leqslant 2^{\gamma_d \cdot \log^{4d+3} \omega}.$$

This result makes a significant progress towards resolving the question in [BGK+21b], who asked whether the classes of graphs of bounded twin-width are polynomially χ-bounded. In fact, our techniques have been subsequently refined by Bourneuf and Thomassé [BT23], who finally resolved the question positively.

Let us briefly comment on the key conceptual differences between the proof of (exponential) χ-boundedness of Bonnet et al. [BGK+21b] and our proof of Theorem 1.3.10. The proof of Bonnet et al. applies a standard strategy in the area of χ-boundedness. Namely, they show that one can partition the vertex set of a graph of twin-width t into $t+2$ subsets so that each subset induces a subgraph with the clique number smaller by at least 1. Then induction is applied to each of these subgraphs, with the clique number being the progress measure in the induction. Without modifications, this strategy inherently leads to an exponential bound on the χ-bounding function. In our proof, we use two different induction steps:

- In one step, we induct on induced subgraphs in which the clique number drops significantly: at least by a constant fraction.
- In the second step, we induct on graphs that may possibly have even larger clique number (but bounded polynomially in the original one), but in which an auxiliary progress measure – the largest size of an almost mixed minor – drops by at least one. This auxiliary measure is bounded in terms of the twin-width, so this step can be applied only a constant number of times, provided the twin-width is originally bounded by a constant.

In the first step, after we partition the graph into induced subgraphs with significantly smaller clique numbers, we employ a variant of the bucketing scheme from the proof by Chudnovsky et al. [CPST13] that polynomial χ-boundedness is preserved under closure by the substitution operation. That is, we assign the induced subgraphs to a number of buckets – where each bucket contains subgraphs with similar clique numbers – and we use a separate palette of colors for each bucket in a proper coloring of the graph. More details can be found in Section 7.2.4.

The auxiliary progress measure used in the second step is expressed through the nonexistence of certain structures in the adjacency matrix of the graph. For this reason, the entire reasoning needs to be conducted in the matrix setting.

7.1 Almost mixed minors

We begin by explaining the central progress measure behind our proof of quasi-polynomial χ-boundedness – *almost mixed minors*. On the technical level, this detail is somewhat analogous to the choice of working with the notion of *quasi-index* in [GPT21].

A d-division $\mathcal{D}$ of a matrix M is a *d-almost mixed minor* if for each pair of indices $i, j \in [d]$ with $i \neq j$, the zone $\mathcal{D}[i,j]$ is mixed. A matrix is *d-almost mixed-free* if it admits no d-almost mixed minors.

Almost mixed minors differ from mixed minors in that it is not required that the "diagonal" zones are mixed. This variant of mixed minors will prove useful as a technical tool in several key lemmas where we will be unable to reason about the mixedness of these zones.

Clearly, every d-almost mixed-free matrix is also d-mixed-free. On the other hand, every d-mixed-free matrix is $2d$-almost mixed-free. To see that, note that if there was a $2d$-almost mixed minor, then its first d row blocks and last d column blocks would induce a submatrix with a d-mixed minor. Also, note that any submatrix of a d-(almost) mixed-free matrix is also d-(almost) mixed-free; we will often use this fact implicitly.

7.2 Obtaining the recurrence

In this section we provide the main step towards the proof of Theorem 1.3.10, which is a recursive upper bound on the χ-bounding function of graphs admitting a d-almost mixed-free adjacency matrix. After giving some preliminary tools in Section 7.2.1, we formulate this main step in Lemma 7.2.7 in Section 7.2.2, and devote the remainder of Section 7.2 to the proof of this lemma.

7.2.1 Compressions

We start with some auxiliary results on *horizontal* and *vertical compressions* of matrices. These will be later used for handling recursive steps that apply to submatrices with smaller excluded almost mixed minors.

Call a matrix M *graphic* if it is symmetric, has all entries in $\{0,1\}$, and all entries on the diagonal of M are 0. In other words, M is graphic if it is the adjacency matrix of some graph (with respect to some vertex ordering).

Definition 17. *Let $\mathcal{D}$ be a symmetric s-division of a graphic matrix M. We define the* horizontal compression $G^{\mathsf{H}}_{\mathcal{D}}$ *as the graph over vertex set $[s]$ where for any $1 \leqslant i < j \leqslant s$, we have $ij \in E(G^{\mathsf{H}}_{\mathcal{D}})$ if and only if the zone $\mathcal{D}[i,j]$ is nonzero horizontal (equivalently, $\mathcal{D}[j,i]$ is nonzero vertical). We define the* vertical compression $G^{\mathsf{V}}_{\mathcal{D}}$ *symmetrically. The* mixed compression $G^{\mathsf{M}}_{\mathcal{D}}$ *is defined analogously, but we put an edge ij whenever $i \neq j$ and the zone $\mathcal{D}[i,j]$ is mixed.*

Let $M^{\mathsf{H}}_{\mathcal{D}}$, $M^{\mathsf{V}}_{\mathcal{D}}$, and $M^{\mathsf{M}}_{\mathcal{D}}$ be the adjacency matrices of $G^{\mathsf{H}}_{\mathcal{D}}$, $G^{\mathsf{V}}_{\mathcal{D}}$, and $G^{\mathsf{M}}_{\mathcal{D}}$ respectively, with respect to the natural vertex orderings inherited from M.

We first note that thanks to the Marcus-Tardos Theorem (Theorem 2.5.4), the mixed compression of a d-mixed-free matrix is always sparse, and hence colorable with few colors.

Lemma 7.2.1. *For every $d \in \mathbb{N}$ there exists a constant C_d such that the following holds. Let M be a d-mixed-free graphic matrix and $\mathcal{D}$ be a symmetric division of M. Then $\chi(G^{\mathsf{M}}_{\mathcal{D}}) \leqslant C_d$.*

Proof. Let k be the number of row blocks (equivalently, of column blocks) of $\mathcal{D}$. By Theorem 2.5.4 and the fact that d-mixed-freeness is closed under taking submatrices, there exists a constant $c_d > 0$ depending only on d such that for every subset $S \subseteq [k]$, there are at most $c_d|S|$ pairs $i, j \in S$, $i < j$, such that the zone $\mathcal{D}[i,j]$ is mixed. Thus, $|E(G^{\mathsf{M}}_{\mathcal{D}}[S])| \leqslant c_d|S|$. Since S was chosen arbitrarily, it follows that $G^{\mathsf{M}}_{\mathcal{D}}$ is $2c_d$-degenerate, and hence $\chi(G^{\mathsf{M}}_{\mathcal{D}}) \leqslant 2c_d + 1$. So we may set $C_d := 2c_d + 1$. □

We next observe that almost mixed minors in $M^{\mathsf{H}}_{\mathcal{D}}$ lift to almost mixed minors in M.

Lemma 7.2.2. *Let M be a graphic matrix and $\mathcal{D}$ be a symmetric division of M. Suppose $M^{\mathsf{H}}_{\mathcal{D}}$ contains a d-almost mixed minor $\mathcal{E}$ for some $d \geqslant 2$. (Note that $\mathcal{E}$ is not necessarily symmetric.) Then M contains a d-almost mixed minor $\mathcal{L}$ such that:*

- *$\mathcal{L}$ is a coarsening of $\mathcal{D}$ (that is, each row block of $\mathcal{L}$ is the union of some convex subset of row blocks of $\mathcal{D}$, and the same applies for column blocks); and*
- *each zone of $\mathcal{L}$ is formed by an intersection of at least two row blocks of $\mathcal{D}$ and at least two column blocks of $\mathcal{D}$.*

Proof. We lift $\mathcal{E}$ to a d-almost mixed division $\mathcal{L}$ of M naturally as follows. For $i \in [d]$, if the ith row block of $\mathcal{E}$ spans rows $r_1 \dots r_2$ of $M^{\mathsf{H}}_{\mathcal{D}}$, then we set the ith row block of $\mathcal{L}$ to be the union of row blocks of $\mathcal{E}$

from the r_1th to the r_2th. Similarly for column blocks. Thus, if $i, j \in [d]$ and $\mathcal{E}[i,j]$ is the intersection of rows $r_1, \ldots, r_2$ and columns $c_1, \ldots, c_2$ of $M^{\mathsf{H}}_{\mathcal{D}}$, then

$$\mathcal{L}[i,j] = \mathcal{D}[[r_1, r_2],\, [c_1, c_2]].$$

Clearly, $\mathcal{L}$ is a coarsening of $\mathcal{D}$. Furthermore, since $d \geqslant 2$ and $\mathcal{E}$ is a d-almost mixed minor of $M^{\mathsf{H}}_{\mathcal{D}}$, it is easy to see that each row block and each column block of $\mathcal{L}$ has to span at least two blocks of $\mathcal{D}$, for otherwise no corner in $M^{\mathsf{H}}_{\mathcal{D}}$ would fit into this block. So it remains to show that each zone $\mathcal{L}[i,j]$ with $i, j \in [d]$, $i \neq j$, is mixed.

Fix some $i, j \in [d]$ with $i \neq j$. Assume that $\mathcal{E}[i,j]$ is formed by the intersection of rows $r_1, \ldots, r_2$ and columns $c_1, \ldots, c_2$ of $M^{\mathsf{H}}_{\mathcal{D}}$. Since $\mathcal{E}[i,j]$ is mixed, there is a corner $C = M^{\mathsf{H}}_{\mathcal{D}}[[r, r+1],\, [c, c+1]]$ for some $r_1 \leqslant r < r_2$ and $c_1 \leqslant c < c_2$. It is then enough to show that the submatrix $A := \mathcal{D}[[r, r+1],\, [c, c+1]]$ of M is mixed. Indeed, since A is a submatrix of $\mathcal{L}[i,j] = \mathcal{D}[[r_1, r_2], [c_1, c_2]]$, it will follow that $\mathcal{L}[i,j]$ is mixed as well.

We perform a case study, depending on the value of $c - r$ (which intuitively signifies how close C is to the diagonal of $M^{\mathsf{H}}_{\mathcal{D}}$):

Case 1a: $c \geqslant r + 2$. Then C is strictly above the diagonal of $M^{\mathsf{H}}_{\mathcal{D}}$. So for each $i \in \{r, r+1\}$ and $j \in \{c, c+1\}$, we have $M^{\mathsf{H}}_{\mathcal{D}}[i,j] = 1$ if and only if $\mathcal{D}[i,j]$ is nonzero horizontal. If A were horizontal, then for each $i \in \{r, r+1\}$ we would have $M^{\mathsf{H}}_{\mathcal{D}}[i,c] = M^{\mathsf{H}}_{\mathcal{D}}[i,c+1]$, and C would be horizontal; a contradiction with C being a corner. Similarly, if A were vertical, then for each $j \in \{c, c+1\}$ we would have $M^{\mathsf{H}}_{\mathcal{D}}[r,j] = M^{\mathsf{H}}_{\mathcal{D}}[r+1,j]$ and C would be vertical; again a contradiction with C being a corner. So A must be mixed.

Case 1b: $c \leqslant r - 2$. Then C is strictly below the diagonal of $M^{\mathsf{H}}_{\mathcal{D}}$. So for each $i \in \{r, r+1\}$ and $j \in \{c, c+1\}$, we have $M^{\mathsf{H}}_{\mathcal{D}}[i,j] = 1$ if and only if $\mathcal{D}[i,j]$ is nonzero vertical. Hence, the proof is symmetric to Case 1a.

Case 2a: $c = r + 1$. Then C intersects the diagonal of $M^{\mathsf{H}}_{\mathcal{D}}$ at entry $M^{\mathsf{H}}_{\mathcal{D}}[r+1,c]$, while the remaining entries are above the diagonal. Observe that $\mathcal{D}[r+1,c]$ is graphic, and thus either constant 0 or mixed. If it is mixed, then A is already mixed as well. So from now on assume that $\mathcal{D}[r+1,c]$ is constant 0. Consequently, $M^{\mathsf{H}}_{\mathcal{D}}[r+1,c] = 0$.

First, assume that A is horizontal. Then, $\mathcal{D}[r+1,c+1]$ is constant 0 as well and thus $M^{\mathsf{H}}_{\mathcal{D}}[r+1,c+1] = 0$. Moreover, $\mathcal{D}[\{r\},\, [c,c+1]]$ is horizontal, implying that $M^{\mathsf{H}}_{\mathcal{D}}[r,c] = M^{\mathsf{H}}_{\mathcal{D}}[r,c+1]$. (Note that both these entries are equal to 1 if and only if $\mathcal{D}[r,c]$ is nonzero, or equivalently if $\mathcal{D}[r,c+1]$ is nonzero.) So C cannot be a corner in $M^{\mathsf{H}}_{\mathcal{D}}$, a contradiction.

Next, assume that A is vertical. Analogously, $\mathcal{D}[r,c]$ is constant 0, hence $M^{\mathsf{H}}_{\mathcal{D}}[r,c] = 0$, while $\mathcal{D}[[r, r+1],\, \{c+1\}]$ is vertical, implying that $M^{\mathsf{H}}_{\mathcal{D}}[r,c+1] = M^{\mathsf{H}}_{\mathcal{D}}[r+1,c+1]$. So again, we conclude that C cannot be a corner, a contradiction.

Case 2b: $c = r - 1$. This case is symmetric to Case 2a.

Case 3: $c = r$. That is, C intersects the diagonal of $M^{\mathsf{H}}_{\mathcal{D}}$ at $M^{\mathsf{H}}_{\mathcal{D}}[r,c]$ and $M^{\mathsf{H}}_{\mathcal{D}}[r+1,c+1]$, while $M^{\mathsf{H}}_{\mathcal{D}}[r+1,c]$ is below the diagonal and $M^{\mathsf{H}}_{\mathcal{D}}[r,c+1]$ is above the diagonal. As in Case 2a, each of $\mathcal{D}[r,c]$ and $\mathcal{D}[r+1,c+1]$ is either mixed or constant 0. If any of them is mixed, then A is mixed as well and we are done; so assume that that both $\mathcal{D}[r,c]$ and $\mathcal{D}[r+1,c+1]$ are constant 0. Now, if A were horizontal or vertical, then both $\mathcal{D}[r+1,c]$ and $\mathcal{D}[r,c+1]$ would be constant 0 as well, implying that C would be constant 0. This is a contradiction with C being a corner.

This finishes the proof. □

Naturally, a statement symmetric to Lemma 7.2.2 applies to $M^{\mathsf{V}}_{\mathcal{D}}$. Thus:

Corollary 7.2.3. *Suppose M is a graphic matrix that is d-almost mixed-free for some $d \geqslant 2$, and $\mathcal{D}$ is a symmetric division of M. Then both $M^{\mathsf{H}}_{\mathcal{D}}$ and $M^{\mathsf{V}}_{\mathcal{D}}$ are d-almost mixed-free.*

The next lemma is the main outcome of this section. It shows that if M is the adjacency matrix of a graph G (with respect to some vertex ordering) and M has no large almost mixed minor, then the clique numbers of the compressions of M are controlled in terms of the clique number of G.

Lemma 7.2.4. *Let G be a graph and denote $\omega = \omega(G)$. Let M be the adjacency matrix of G in some vertex ordering and let $\mathcal{D}$ be a symmetric division of M. Suppose M has no d-almost mixed minor that is a coarsening of $\mathcal{D}$, for some $d \geqslant 1$. Then*

$$\omega(G^{\mathsf{H}}_{\mathcal{D}}) < 2\binom{\omega + d - 2}{d - 1} \leqslant 2\omega^{d-1}.$$

Proof. Let

$$\mu(\omega, d) = 2\binom{\omega + d - 2}{d - 1} - 1.$$

It can be easily verified that $\mu(\omega, d)$ satisfies the following recursive definition:

- $\mu(1, \cdot) = \mu(\cdot, 1) = 1$; and
- $\mu(\omega, d) = \mu(\omega - 1, d) + \mu(\omega, d - 1) + 1$ for $\omega, d \geqslant 2$.

We now show that $\omega(G^{\mathsf{H}}_{\mathcal{D}}) \leqslant \mu(\omega, d)$ by induction on ω and d.

If $\omega = 1$, then G must be edgeless. So $G^{\mathsf{H}}_{\mathcal{D}}$ is edgeless as well, implying that $\omega(G^{\mathsf{H}}_{\mathcal{D}}) \leqslant 1$. There are no 1-almost mixed-free matrices, so $\omega(G^{\mathsf{H}}_{\mathcal{D}}) \leqslant 1$ is vacuously true for $d = 1$. This resolves the base of the induction, so from now on assume that $d, \omega \geqslant 2$.

Assume for contradiction that $\omega(G^{\mathsf{H}}_{\mathcal{D}}) > \mu(\omega, d)$. By restricting G to a suitable induced subgraph, without loss of generality, we may assume that $G^{\mathsf{H}}_{\mathcal{D}}$ is a complete graph with $k \coloneqq \mu(\omega, d) + 1$ vertices. Note that this means that for each $1 \leqslant i < j \leqslant k$, the zone $\mathcal{D}[i, j]$ is nonzero horizontal. Since we assume that $\mathcal{D}$ is a symmetric division, $\mathcal{D}$ yields a partition of $V(G)$ into k subsets $V_1, V_2, \ldots, V_k$. These subsets are convex in the vertex ordering used in the construction of M, and are indexed naturally according to this vertex ordering.

Let $\ell \coloneqq \mu(\omega, d - 1) + 1$ and

$$Y \coloneqq V_1 \cup V_2 \cup \cdots \cup V_\ell \qquad \text{and} \qquad Z \coloneqq V(G) \setminus Y.$$

Note that $k = \mu(\omega, d) + 1 = \mu(\omega, d - 1) + \mu(\omega - 1, d) + 2$, hence Y is the union of $\ell = \mu(\omega, d - 1) + 1$ parts V_i and Z is the union of $\mu(\omega - 1, d) + 1$ parts V_i.

Claim 7.2.5. $\omega(G[Z]) = \omega$.

Proof. Let $\mathcal{D}_Z$ be the restriction of the division $\mathcal{D}$ restricted to blocks corresponding to the vertices of Z. By construction, we have that $\omega(G[Z]^{\mathsf{H}}_{\mathcal{D}_Z}) \geqslant \mu(\omega - 1, d) + 1$. Therefore, by induction assumption applied to the graph $G[Z]$ with division $\mathcal{D}_Z$, we infer that either $\omega(G[Z]) > \omega - 1$ or M has a d-almost mixed minor that is a coarsening of $\mathcal{D}_Z$. The latter case leads to the contradiction (such an almost mixed minor would lift to a d-almost mixed minor that is a coarsening of $\mathcal{D}$), so it is indeed the case that $\omega(G[Z]) > \omega - 1$. □

Claim 7.2.5 implies that there is no vertex $v \in Y$ that would be complete towards Z (i.e., adjacent to every vertex of Z). Indeed, if this were the case, then v together with the largest clique in $G[Z]$ would form a clique in G of size at least $\omega + 1$, a contradiction.

Since no vertex of Y is complete towards Z, for every $i \in [\ell]$ one can find $j(i) > \ell$ such that the zone $\mathcal{D}[i, j(i)]$ in M is *not* constant 1. However, as $M^{\mathsf{H}}_{\mathcal{D}}[i, j(i)] = 1$, we know that $\mathcal{D}[i, j(i)]$ is nonzero horizontal. Hence, $\mathcal{D}[i, j(i)]$ is nonconstant horizontal.

For every $i \in [\ell]$, consider the submatrix

$$R_i \coloneqq \mathcal{D}[\{i\},\ [\ell + 1, k]].$$

Note that R_i contains the zone $\mathcal{D}[i, j(i)]$. This means that R_i cannot be vertical, as its zone $\mathcal{D}[i, j(i)]$ is nonconstant horizontal. It also cannot be horizontal, as then there would exist a vertex $v \in V_i \subseteq Y$ that would be complete towards Z. Therefore, R_i is mixed. By symmetry, the submatrix $S_i \coloneqq \mathcal{D}[[\ell + 1, k],\ \{i\}]$ is mixed as well (Figure 7.1a).

Let

$$M' \coloneqq \mathcal{D}[[1, \ell],\ [1, \ell]]$$

be the adjacency matrix of $G[Y]$ in the vertex ordering inherited from G. We observe that M' cannot contain any $(d - 1)$-almost mixed minor $\mathcal{E}$ that would be a coarsening of $\mathcal{D}$ (restricted to blocks corresponding to the vertices of Y). Indeed, otherwise we could form a d-almost mixed minor $\mathcal{E}'$ of M by adding to $\mathcal{E}$ one column block, spanning the suffix of columns of M corresponding to the vertices of Z, and one row block, spanning the suffix of rows of M corresponding to the vertices of Z (Figure 7.1b). By the mixedness of R_i and S_i for each $i \in [\ell]$, we see that for each $j \in [d - 1]$, the zones $\mathcal{E}'[d, j]$ and $\mathcal{E}'[j, d]$ would be mixed. So $\mathcal{E}'$ would be a d-almost mixed minor in M that would be coarsening of $\mathcal{D}$, a contradiction.

As $\ell = \mu(\omega, d - 1) + 1$, we may now apply the induction assumption to the graph $G[Y]$ with division $\mathcal{D}$ restricted to blocks corresponding to the vertices of Y. Analogously to the proof of Claim 7.2.5, we infer that

$$\omega(G[Y]) > \omega.$$

As $G[Y]$ is an induced subgraph of G, this is a contradiction that finishes the proof. □

We remark that a symmetric reasoning shows that the same conclusion applies also to $G^{\mathsf{V}}_{\mathcal{D}}$: Under the assumptions of Lemma 7.2.4, we also have $\omega(G^{\mathsf{V}}_{\mathcal{D}}) \leqslant 2\omega^{d-1}$.

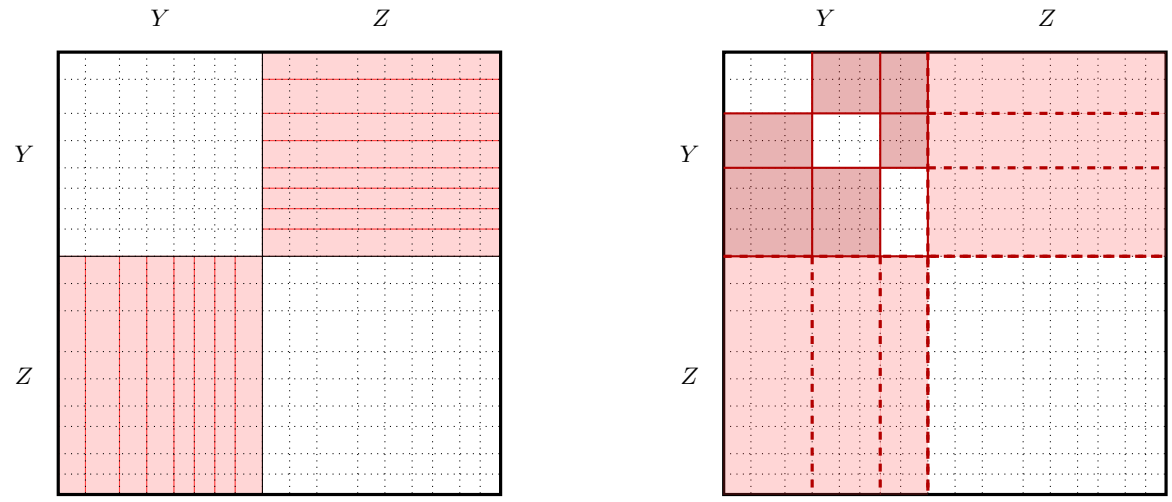


Figure 7.1: Setup in the proof of Lemma 7.2.4.
(a) A symmetric division $\mathcal{D}$ of matrix M (dotted). Each red horizontal strip (R_i) and each red vertical strip (S_i) is mixed.
(b) Each $(d-1)$-almost mixed minor of M' that is a coarsening of $\mathcal{D}$ (dark red) can be extended to a d-almost mixed minor of M by adding Z as the final row and column block.

7.2.2 Statement of the main lemma

With auxiliary tools prepared, we can proceed to the main result of this section. Let $f_d\colon \mathbb{Z}_{>0} \to \mathbb{Z}_{>0}$ be defined as follows: For $\omega \in \mathbb{N}$, $f_d(\omega)$ is the maximum chromatic number among graphs of clique number at most ω that admit a vertex ordering yielding a d-almost mixed-free adjacency matrix. Note that by Theorem 2.5.5 and [BGK+21b, Theorem 4], $f_d(\omega)$ is finite for all $\omega \in \mathbb{N}$. We have the following easy observation.

Lemma 7.2.6. *Every graph of twin-width at most t and clique number ω has chromatic number at most $f_{4t+4}(\omega)$.*

Proof. Every graph of twin-width at most t has a vertex ordering that yields a $(2t+2)$-mixed-free adjacency matrix (Theorem 2.5.5), and every $(2t+2)$-mixed-free matrix is also $(4t+4)$-almost mixed-free. Given this, the claim follows from the definition of $f_{4t+4}(\omega)$. □

For convenience, let us extend the domain of f_d to $\mathbb{R}_{>0}$ by setting $f_d(x) = f_d(\lceil x \rceil)$ for every positive noninteger x. The main step towards a better bound on f_d is the recurrence provided by the following lemma.

Lemma 7.2.7. *Let d, ω, k be integers satisfying $d \geqslant 3$, $\omega \geqslant 5$, and $1 \leqslant k < \omega/4$. Then there exists a constant C_d depending only on d such that*

$$f_d(\omega) \leqslant f_d(\omega - k) + C_d \left[f_d(\omega - k) + 8C_d f_{d-1}^2(2\omega^{d-1}) \cdot \sum_{u=0}^{\lfloor \log_2 k \rfloor} f_d(2^{u+1}) \cdot f_d\left(\frac{2k}{2^u} + 1\right) \right]. \tag{7.1}$$

The remainder of this section is devoted to the proof of Lemma 7.2.7. For this, fix d, ω, k as in the premise of Lemma 7.2.7, as well as a graph G that, under some vertex ordering $\leqslant$, admits a d-almost mixed-free adjacency matrix M. Our goal is to construct a proper coloring of G with the number of colors bounded by the right hand side of Eq. (7.1). The construction is obtained by a sequence of *coloring steps*, each of which constructs a part of the coloring and reduces the remaining task to a simpler one.

For simplicity we may assume that $V(G) = [n]$ and $\leqslant$ is the standard order on $[n]$, so that vertices are equal to the indices of their rows and columns in M.

7.2.3 Forming blobs and simplifying connections

The first step of the construction is to partition the vertex set of G into parts, called *blobs*, which are significantly simpler in terms of the clique number. Formally, we construct blobs $B_1, B_2, \ldots, B_{m+1}$ by

an inductive procedure as follows. Supposing $B_1, \ldots, B_{i-1}$ are already defined for some $i \geqslant 1$, we let B_i be the smallest prefix of $V(G) \setminus \bigcup_{j=1}^{i-1} B_j$ in the ordering $\leqslant$ that satisfies $\omega(G[B_i]) \geqslant \omega - k$. If no such prefix exists, we finish the construction by setting $m := i-1$ and $B_{m+1} := V(G) \setminus \bigcup_{j=1}^{i-1} B_j$. Since adding one vertex can increase the clique number by at most 1, it is easy to see that the blobs satisfy the following assertions:

- Blobs $B_1, \ldots, B_{m+1}$ form a partition of $V(G)$.
- Blobs $B_1, \ldots, B_m$ are nonempty.
- Each blob is *convex* in the ordering $\leqslant$, that is, its vertices form an interval in $\leqslant$. Moreover, the blobs are ordered by $\leqslant$ in according to their indices: if $1 \leqslant i < j \leqslant m+1$, then $u < v$ for all $u \in B_i$ and $v \in B_j$.
- For each $i \in [m]$ we have $\omega(G[B_i]) = \omega - k$.
- We have $\omega(G[B_{m+1}]) < \omega - k$.

The first step in constructing a coloring of G is to resolve the part $G[B_{m+1}]$.

Coloring Step 1. *Color $G[B_{m+1}]$ using a separate palette of $f_d(\omega - k)$ colors; such a proper coloring exists due to $\omega(G[B_{m+1}]) \leqslant \omega - k$. From now on, we may disregard B_{m+1} from further considerations. That is, our goal is to properly color $G[B_1 \cup B_2 \cup \cdots \cup B_m]$ using*

$$C_d \left[f_d(\omega - k) + 8C_d f_{d-1}^2(2\omega^{d-1}) \cdot \sum_{u=0}^{\lfloor \log_2 k \rfloor} f_d(2^{u+1}) \cdot f_d\left(\frac{2k}{2^u} + 1\right) \right] \quad \text{colors.}$$

For a subset of rows X and a subset of columns Y, by $M[X,Y]$ we denote the matrix obtained from M by deleting all rows not belonging to X and all columns not belonging to Y. We sometimes call $M[X,Y]$ the *connection* between X and Y. The next goal is to resolve mixed connections between the blobs. This is easy thanks to Lemma 7.2.1.

Lemma 7.2.8. *Let C_d be the constant provided by Lemma 7.2.1. Then there exists an integer $\ell \leqslant C_d$ and a partition of $[m]$ into (not necessarily convex) subsets $A_1, A_2, \ldots, A_\ell$ so that for any given $i \in [\ell]$ and a pair $b, c \in A_i$ with $b \neq c$, the matrix $M[B_b, B_c]$ is* not *mixed.*

Proof. It suffices to apply Lemma 7.2.1 to the mixed compression of the matrix $M[\bigcup_{i=1}^m B_i, \bigcup_{i=1}^m B_i]$ along its division into blobs $B_1, \ldots, B_m$. □

Coloring Step 2. *Let $[m] = A_1 \cup A_2 \cup \cdots \cup A_\ell$ be the partition provided by Lemma 7.2.8 (with $\ell \leqslant C_d$). Assign a separate palette of colors to each set A_i, $i \in [\ell]$. That is, supposing we properly color each graph $G[\bigcup_{a \in A_i} B_a]$ using*

$$f_d(\omega - k) + 8C_d f_{d-1}^2(2\omega^{d-1}) \cdot \sum_{u=0}^{\lfloor \log_2 k \rfloor} f_d(2^{u+1}) \cdot f_d\left(\frac{2k}{2^u} + 1\right) \quad \text{colors,} \tag{7.2}$$

we may construct a proper coloring of $G[B_1 \cup \cdots \cup B_m]$ by taking the union of the colorings of $G[\bigcup_{a \in A_i} B_a]$, $i \in [m]$, on disjoint palettes.

For the simplicity of presentation, from now on we focus on a single set A_i. That is, by restricting attention to the induced subgraph $G[\bigcup_{a \in A_i} B_a]$ with vertex ordering inherited from G, we may assume that $V(G) = B_1 \cup \cdots \cup B_m$ and there are no mixed connections between any pair of different blobs. Then our goal is to properly color G using the number of colors given by Eq. (7.2).

Let $\mathcal{D}$ be the symmetric m-division of M given by the partition into blobs. Thus, $\mathcal{D}$ has no mixed zones outside of the main diagonal, so for $i, j \in [m]$, $i \neq j$, the connection $\mathcal{D}[i,j] = M[B_i, B_j]$ is of one of the following types:

- *Empty* if $\mathcal{D}[i,j]$ is constant 0.
- *Nonconstant horizontal* if $\mathcal{D}[i,j]$ is horizontal but not constant. In graph-theoretic terms, this means that B_i is semi-pure towards B_j. (Recall that this means that each vertex $v \in B_i$ is either complete or anti-complete towards B_j.)
- *Nonconstant vertical* if $\mathcal{D}[i,j]$ is vertical but not constant. Again, in graph-theoretic terms this means that B_j is semi-pure towards B_i.

Here, note that the connection $\mathcal{D}[i,j]$ cannot be constant 1. This is because then the pair B_i, B_j would be complete, implying that

$$\omega(G) \geqslant \omega(G[B_i]) + \omega(G[B_j]) \geqslant 2(\omega - k) > \omega,$$

a contradiction.

Given a vertex v, say $v \in B_i$ for some $i \in [m]$, we say that v is *rich* if there exists a blob B_j, $j \neq i$, such that v is complete towards B_j. Otherwise, we say that v is *poor*. We next observe that poor vertices can be only adjacent within single blobs.

Lemma 7.2.9. *Suppose u and v are poor vertices such that $u \in B_i$ and $v \in B_j$ for some $i \neq j$. Then u and v are nonadjacent.*

Proof. Assume otherwise. Then $M[u,v] = M[v,u] = 1$. Thus, the zone $\mathcal{D}[i,j]$ is not constant 0 due to containing $M[u,v]$. If it was horizontal, then u would be pure towards the entire blob B_j, so u would be rich. Symmetrically, if $\mathcal{D}[i,j]$ was vertical, then v would be rich. Since both u and v are assumed to be poor, we have a contradiction. □

Let Z be the set of all poor vertices. By Lemma 7.2.9, we see that $G[Z]$ is the disjoint union of graphs $G[Z \cap B_1], \ldots, G[Z \cap B_m]$. It follows that $\omega(G[Z]) \leqslant \omega - k$.

Coloring Step 3. *Color Z using a separate palette of $f_d(\omega - k)$ colors; this can be done due to $\omega(G[Z]) \leqslant \omega - k$. It remains to properly color the graph $G - Z$ using*

$$8C_d f_{d-1}^2(2\omega^{d-1}) \cdot \sum_{u=0}^{\lfloor \log_2 k \rfloor} f_d(2^{u+1}) \cdot f_d\left(\frac{2k}{2^u} + 1\right) \quad \textit{colors.}$$

For each $i \in [m]$, let $B_i' := B_i \setminus Z$. Intuitively, our next goals are to first construct a proper coloring of each subgraph $G[B_i']$ separately, and then use the coloring of the vertices within the blobs to resolve the semi-pure interblob connections.

7.2.4 Forming and analyzing subblobs

Fix $i \in [m]$ and consider the set B_i' with the ordering inherited from G. We now want to find a proper coloring of $G[B_i']$. We partition B_i' into four (not necessarily convex) subsets $B_{i,x,y}'$ with $x, y \in \{0,1\}$. We put each $v \in B_i'$ into $B_{i,x,y}'$ for (x,y) defined as follows:

- $x = M[v, \mathsf{first}]$, where $\mathsf{first} = 1$ is the first vertex of G in the ordering $\leqslant$; and
- $y = M[v, \mathsf{last}]$, where $\mathsf{last} = |V(G)|$ is the last vertex of G in the ordering $\leqslant$.

Now, fix $x, y \in \{0,1\}$ for a moment. We partition $B_{i,x,y}'$ into sets $I_{i,x,y}^1 \cup I_{i,x,y}^2 \cup \cdots \cup I_{i,x,y}^t$, called *subblobs*, by induction as follows. Assuming $I_{i,x,y}^1, \ldots, I_{i,x,y}^{j-1}$ are already defined, $I_{i,x,y}^j$ is the largest prefix of $\leqslant$ restricted to $B_{i,x,y}' \setminus \bigcup_{s=1}^{j-1} I_{i,x,y}^s$ with the following property: All vertices of $I_{i,x,y}^j$ are *twins* with respect to $V(G) \setminus B_i$ (that is, in G they have exactly the same neighborhood in $V(G) \setminus B_i$). The construction finishes when every vertex of $B_{i,x,y}'$ is placed in a subblob. Note that subblobs $I_{i,x,y}^j$ are not necessarily convex in $\leqslant$, but they are convex in $\leqslant$ restricted to $B_{i,x,y}'$, and they are ordered naturally by $\leqslant$: $w < w'$ for all $w \in I_{i,x,y}^j$ and $w' \in I_{i,x,y}^{j'}$ with $j < j'$ (Figure 7.2). Note that we require that the vertices within every subblob are twins with respect to all blobs different than B_i in the entire graph G (where G contains both rich and poor vertices).

We observe that every subblob induces a graph of small clique number.

Lemma 7.2.10. *For each $j \in [t]$, we have $\omega(G[I_{i,x,y}^j]) \leqslant k$.*

Proof. Since $I_{i,x,y}^j$ consists of rich twins with respect to $G - B_i$, it follows that there exists some blob $B_{i'}$, $i' \neq i$, such that the pair $I_{i,x,y}^j, B_{i'}$ is complete. But we have $\omega(G[B_{i'}]) = \omega - k$, so it follows that $\omega(G[I_{i,x,y}^j]) \leqslant \omega - (\omega - k) = k$. □

We now divide the subblobs into a logarithmic number of *buckets* according to the clique number of the subgraphs induced by them. We remark that this bucketing approach is inspired by the proof of Chudnovsky et al. [CPST13] that polynomial χ-boundedness is preserved under closure by the substitution operation. Intuitively, the idea is to capture the tradeoff between the heaviness of the subblobs in terms

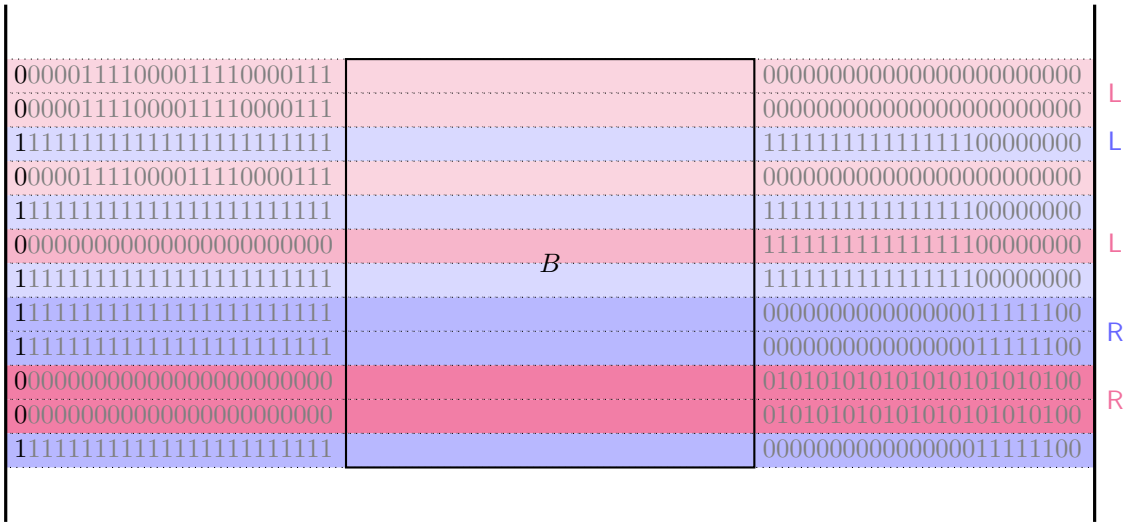


Figure 7.2: A blob B with 12 rich vertices and its partition into the subblobs. For simplicity, each vertex of B has $y = 0$ (i.e., no vertices of B are connected to last).
The subblobs containing vertices with $x = 0$ are marked with different shades of red, and the subblobs containing vertices with $x = 1$ are marked with different shades of blue. For each subblob, its type $z \in \{\mathsf{L}, \mathsf{R}\}$ is marked on the right of the matrix.

of their clique number, and the sparseness of the graph of connections between the subblobs. These two quantities are respectively represented by the terms $f_d(2^{u+1})$ and $f_d\left(\frac{2k}{2^u} + 1\right)$ in the right hand side of Eq. (7.1), and clearly these two terms "play against each other".

Let $\ell := \lfloor \log k \rfloor$. We partition the subblobs $I^1_{i,x,y}, \ldots, I^t_{i,x,y}$ into $2(\ell+1)$ *buckets* $\mathcal{S}_{i,x,y,z,u}$ for $z \in \{\mathsf{L}, \mathsf{R}\}$ and $u \in \{0, 1, \ldots, \ell\}$ using the following process. For every $j \in [t]$, let v_j be any vertex of $I^j_{i,x,y}$. We put $I^j_{i,x,y}$ into the bucket $\mathcal{S}_{i,x,y,z,u}$ for (z, u) defined as follows:

- If $j = 1$ then $z = \mathsf{L}$. Otherwise, that is for $j \geqslant 2$, we know that there exists a vertex $a \in B_{i'}$ with $i' \neq i$ such that $M[v_j, a] \neq M[v_{j-1}, a]$. Pick any such vertex. If $i' < i$, then we put $z = \mathsf{L}$; otherwise, put $z = \mathsf{R}$ (Figure 7.2).
- u is such that $2^u \leqslant \omega(G[I^j_{i,x,y}]) < 2^{u+1}$.

Observe that Lemma 7.2.10 ensures that every subblob is placed in a bucket.

If a subblob $I^j_{i,x,y} \subseteq B'_{i,x,y}$ belongs to the bucket $\mathcal{S}_{i,x,y,z,u}$, we call $I^j_{i,x,y}$ an (x, y, z, u)-*subblob*. Let $W_{x,y,z,u}$ be the union of all (x, y, z, u)-subblobs in G, that is,

$$W_{x,y,z,u} = \bigcup_{i=1}^{m} \bigcup \mathcal{S}_{i,x,y,z,u}.$$

The idea is to assign a separate palette to every choice of (x, y, z, u) as above.

Coloring Step 4. *For each quadruple of parameters $(x, y, z, u) \in \{0, 1\}^2 \times \{\mathsf{L}, \mathsf{R}\} \times \{0, 1, \ldots, \ell\}$, we assign a separate palette for coloring the subgraph $G[W_{x,y,z,u}]$ with*

$$C_d \cdot f^2_{d-1}(2\omega^{d-1}) \cdot f_d(2^{u+1}) \cdot f_d\left(\frac{2k}{2^u} + 1\right) \quad \textit{colors.} \tag{7.3}$$

That is provided we properly color every subgraph $G[W_{x,y,z,u}]$ with that many colors, we can color the whole graph G using the union of those coloring on separate palettes.

Therefore, from now on we fix a quadruple $(x, y, z, u) \in \{0, 1\}^2 \times \{\mathsf{L}, \mathsf{R}\} \times \{0, 1, \ldots, \ell\}$ and focus on coloring $G[W_{x,y,z,u}]$ using as many colors as specified in Eq. (7.3). Denote $W := W_{x,y,z,u}$ for brevity.

First, we construct a coloring that at least deals with edges within subblobs.

Coloring Step 5. *For each (x, y, z, u)-subblob I, properly color the subgraph $G[I]$ using $f_d(2^{u+1})$ colors; this is possible due to $\omega(G[I]) \leqslant 2^{u+1}$. Take the union of these colorings using* the same *palette of $f_d(2^{u+1})$ colors. This is a coloring of W with the property that every two adjacent vertices of W belonging to the same subblob receive different colors. Call this coloring λ_1.*

Coloring λ_1 defined above already properly colors all the edges within subblobs. Our next goal is to refine λ_1 to a coloring that also properly color edges connecting vertices from different subblobs. These come in two different types: The subblobs may be either contained in the same blob, or be contained in different blobs. Consequently, the refinement is done in two steps corresponding to the two types.

Let us fix $i \in [m]$ and enumerate the bucket $\mathcal{S}_{i,x,y,z,u}$ as $\{I^{j(1)}_{i,x,y}, I^{j(2)}_{i,x,y}, \ldots, I^{j(\alpha)}_{i,x,y}\}$, where $\alpha := |\mathcal{S}_{i,x,y,z,u}|$ and $j(1) < j(2) < \cdots < j(\alpha)$. Recall that for each subblob, we have previously chosen an arbitrary vertex $v_{j(b)} \in I^{j(b)}_{i,x,y}$. We observe the following.

Lemma 7.2.11. *For every $c \in \{2, 3, \ldots, \alpha\}$, we have the following.*

- *If $z = \mathsf{L}$, then the submatrix $M[\{w \mid v_{j(c-1)} \leqslant w \leqslant v_{j(c)}\}, \bigcup_{i'<i} B_{i'}]$ is mixed;*
- *If $z = \mathsf{R}$, then the submatrix $M[\{w \mid v_{j(c-1)} \leqslant w \leqslant v_{j(c)}\}, \bigcup_{i'>i} B_{i'}]$ is mixed.*

Proof. Assume that $z = \mathsf{L}$. The proof for $z = \mathsf{R}$ is symmetric, so we omit it.

Since $I^{j(c)}_{i,x,y} \in \mathcal{S}_{i,x,y,z,u}$, the vertices $v_{j(c)}$ and $v_{j(c)-1}$ have different neighborhoods in the set $\bigcup_{i'<i} B_{i'}$. That is, there is a vertex $a \in \bigcup_{i'<i} B_{i'}$ such that $M[v_{j(c)-1}, a] \neq M[v_{j(c)}, a]$. Note that the existence of a implies that $i > 1$, so in particular $\mathsf{first} \notin B_i$. By the construction, we have $M[v_{j(c)-1}, \mathsf{first}] = M[v_{j(c)}, \mathsf{first}] = x$. It follows that $M[\{v_{j(c-1)}, \ldots, v_{j(c)}\}, \bigcup_{i'<i} B_{i'}]$ contains a mixed 2×2 submatrix $M[\{v_{j(c)-1}, v_{j(c)}\}, \{\mathsf{first}, a\}]$, so it is mixed as well. □

Let $G_i := G[\bigcup \mathcal{S}_{i,x,y,z,u}]$, and let $M_i := M[\bigcup \mathcal{S}_{i,x,y,z,u}, \bigcup \mathcal{S}_{i,x,y,z,u}]$ be its adjacency matrix in the order inherited from G. Let also $\mathcal{D}_i$ be the (symmetric) α-division of M_i according to the boundaries of subblobs in $\mathcal{S}_{i,x,y,z,u}$.

Naturally, for each $p < q$, if the zone $\mathcal{D}_i[p, q]$ is nonzero, then it is of at least one of the following types: mixed (type M), nonzero horizontal (type H) or nonzero vertical (type V). Our goal is to construct three colorings of the subblobs in $\mathcal{S}_i$ that respectively take care of these three types of connections; these will be called ϕ^{M}, ϕ^{H}, and ϕ^{V}, respectively. Hence, it is natural to define G^{H}_i to be the horizontal compression of M_i along its division $\mathcal{D}_i$, and similarly let G^{V}_i and G^{M}_i be the corresponding vertical and mixed compressions. So ϕ^{M}, ϕ^{H}, and ϕ^{V} should be just proper colorings of G^{M}_i, G^{H}_i, and G^{V}_i, respectively.

Obtaining ϕ^{M} is easy. Namely, by Lemma 7.2.1, the graph G^{M}_i admits a proper coloring ϕ^{M} with at most C_d colors; here, C_d is the constant provided by Lemma 7.2.1.

We now show how to obtain colorings ϕ^{H} and ϕ^{V}. Let M^{H}_i and M^{V}_i be the adjacency matrices of G^{H}_i and G^{V}_i, respectively, in the natural order inherited from G. The next lemma is the key conceptual step: We observe that the complexity of the matrices M^{H}_i and M^{V}_i has dropped.

Lemma 7.2.12. *M^{H}_i is $(d-1)$-almost mixed-free.*

Proof. We only prove the lemma for $z = \mathsf{L}$. For $z = \mathsf{R}$ the proof is analogous, so we omit it.

Aiming towards a contradiction, suppose that M^{H}_i contains a $(d-1)$-almost mixed minor $\mathcal{E}$. Since we assume that $d \geqslant 3$ (see the statement of Lemma 7.2.7), Lemma 7.2.2 applies, and there exists a $(d-1)$-almost mixed minor $\mathcal{L}$ of M_i which is a coarsening of $\mathcal{D}_i$, and each (row or column) block of $\mathcal{L}$ spans at least two (row or column) blocks of $\mathcal{D}_i$. We now construct a d-almost mixed minor $\mathcal{L}'$ of M in the following way:

- the first row block of $\mathcal{L}'$ spans rows of $\bigcup_{i'<i} B_{i'}$ of M;
- the first column block of $\mathcal{L}'$ spans columns of $\bigcup_{i'<i} B_{i'}$ of M;
- the ith row block of $\mathcal{L}'$ ($i \in \{2, 3, \ldots, d\}$) spans all rows in M that are spanned by the $(i-1)$st row block of $\mathcal{L}$ in M_i; analogously for the ith column block.

There is a technical detail here: As defined above, formally $\mathcal{L}'$ is a division of a submatrix of M induced by rows and columns of $\bigcup_{i'<i} B_{i'} \cup V(G_i)$. This can be easily fixed by expanding row and column blocks of $\mathcal{L}'$ in any convex way so that they cover all of rows and columns of M. This way the zones can only get larger.

It remains to show that $\mathcal{L}'$ is indeed a d-almost mixed minor of M. Naturally, each $\mathcal{L}'[p, q]$ for $p, q \geqslant 2$, $p \neq q$, is mixed due to the mixedness of $\mathcal{L}[p-1, q-1]$. Also, $\mathcal{L}'[p, 1]$ is mixed for $p \geqslant 2$ for the following reason: The $(p-1)$st row block of $\mathcal{L}$ contains the rows corresponding to the vertices $v_{j(c-1)}, v_{j(c)}$ for some $c \in \{2, 3, \ldots, \alpha\}$, and then, by Lemma 7.2.11, the submatrix $M[\{w \mid v_{j(c-1)} \leqslant w \leqslant v_{j(c)}\}, \bigcup_{i'<i} B_{i'}]$ is mixed. This submatrix is also a submatrix of the zone $\mathcal{L}'[p, 1]$, so this zone is mixed as well. That $\mathcal{L}'[1, q]$ is mixed for every $q \geqslant 2$ follows from a symmetric argument. Thus, $\mathcal{L}'$ is indeed a d-almost mixed minor of M; a contradiction. □

A symmetric proof shows that M_i^{V} is also $(d-1)$-almost mixed-free. By Lemma 7.2.4, we have $\omega(G_i^{\mathsf{H}}) \leqslant 2\omega^{d-1}$, so by we conclude that G_i^{H} admits a proper coloring ϕ^{H} using at most $f_{d-1}(2\omega^{d-1})$ colors. By a symmetric reasoning, G_i^{V} also admits a proper coloring ϕ^{V} using at most $f_{d-1}(2\omega^{d-1})$ colors.

Coloring Step 6. *For every $i \in [m]$, construct a coloring λ_2^i of $V(G_i)$ as follows: For every subblob $I \in \mathcal{S}_{i,x,y,z,u}$ and $v \in I$, we let*

$$\lambda_2^i(v) = (\lambda_1(v), \phi^{\mathsf{M}}(I), \phi^{\mathsf{H}}(I), \phi^{\mathsf{V}}(I)),$$

where ϕ^{M}, ϕ^{H}, and ϕ^{V} are constructed as above. Let λ_2 be the union of colorings λ_2^i for $i \in [m]$ using the same palette of $C_d \cdot f_d(2^{u+1}) \cdot f_{d-1}(2\omega^{d-1})^2$ colors. Thus, λ_2 is a coloring of W using at most $C_d \cdot f_d(2^{u+1}) \cdot f_{d-1}^2(2\omega^{d-1})$ colors that satisfies the following property: For every pair of adjacent vertices $w, w' \in W$ that belong to the same blob, we have $\lambda_2(w) \neq \lambda_2(w')$.

Denote $\Lambda \coloneqq C_d \cdot f_d(2^{u+1}) \cdot f_{d-1}^2(2\omega^{d-1})$. Let F^t for $t \in [\Lambda]$ be the color classes of λ_2. Clearly, each F^t is a subset of W such that $F^t \cap B_i$ is an independent set for each $i \in [m]$. We observe that sets F^t induce subgraphs with relatively small clique numbers.

Lemma 7.2.13. *For each $t \in [\Lambda]$, we have $\omega(G[F^t]) \leqslant 2\left\lfloor \frac{k}{2^u} \right\rfloor + 1$.*

Proof. Denote $\beta \coloneqq \left\lfloor \frac{k}{2^u} \right\rfloor$ for brevity. Suppose that $G[F^t]$ contains a clique K of size $2\beta + 2$. As the intersection of a blob and F^t is an independent set, every vertex of K comes from a different blob. For $v \in K$, let $B(v)$ be the blob containing v.

Construct a digraph T on vertex set K as follows: For distinct $v, v' \in K$, add an arc (v, v') to T if $B(v)$ is semi-pure towards $B(v')$. Since the division $\mathcal{D}$ induced by blobs is assumed to have no mixed zones (as a result of Coloring Step 2), T is semi-complete, that is, for all distinct $v, v' \in K$ at least one of the arcs (v, v') and (v', v) is present. It follows that there exists $w \in K$ whose indegree in T is at least $\beta + 1$. In other words, there are vertices $v_1, \ldots, v_{\beta+1} \in K$, different from w, such that $B(v_i)$ is semi-pure towards $B(w)$ for each $i \in [\beta + 1]$. In particular, since $v_i w \in E(G)$, this implies that each v_i is complete towards $B(w)$.

Let R be a maximum clique in the blob $B(w)$; recall that $|R| = \omega - k$. Next, for each $p \in [\beta + 1]$, let I_p be the subblob of $B(v_p)$ containing v_p, and S_p be a maximum clique in $G[I_p]$; note that S_p might not necessarily contain v_p. Recall also that $2^u \leqslant |S_p| < 2^{u+1}$. Finally, let

$$L \coloneqq R \cup S_1 \cup \cdots \cup S_{\beta+1}.$$

Observe that

$$|L| \geqslant 2^u \cdot \left(\left\lfloor \frac{k}{2^u} \right\rfloor + 1 \right) + (\omega - k) > \omega.$$

So to reach a contradiction, it remains to show that L is a clique in G. To this end, pick two different vertices $a, b \in L$. If a and b come from the same blob, then, by the construction, they are adjacent. Otherwise, we consider two cases:

- $a \in S_p$ and $b \in S_q$ for two different $p, q \in [\beta + 1]$. Since a and v_p are in the same (x, y, z, u)-subblob, we get that a and v_p are twins with respect to the blob $B(v_q)$. In particular, a and v_p are both adjacent or both nonadjacent to b, or equivalently $M[a, b] = M[v_p, b]$. Similarly, b and v_q are twins with respect to the blob $B(v_p)$, implying $M[v_p, b] = M[v_p, v_q]$. But v_p and v_q are adjacent due to belonging to the clique K, so $M[a, b] = M[v_p, v_q] = 1$. Hence a and b are adjacent as well.
- $a \in S_p$ and $b \in R$ for some $p \in [\beta + 1]$. As above, we have $M[a, b] = M[v_p, b]$. But since the blob $B(v_p)$ is semi-pure towards $B(w)$, and $b \in B(w)$, we have $M[v_p, b] = M[v_p, w]$. But v_p and w are adjacent due to belonging to the clique K, so $M[a, b] = M[v_p, w] = 1$. Hence again, a and b are adjacent. □

We can now finalize the construction as follows.

Coloring Step 7. *For each $t \in [\Lambda]$, properly color $G[F^t]$ using a separate palette of*

$$f_d\left(2\left\lfloor \frac{k}{2^u} \right\rfloor + 1\right) \leqslant f_d\left(\frac{2k}{2^u} + 1\right) \quad \textit{colors.}$$

This is possible by Lemma 7.2.13.

Thus, we have obtained a proper coloring of $G[W] = G[W_{x,y,z,u}]$ using $\Lambda \cdot f_d\left(\frac{2k}{2^u} + 1\right)$ colors, which fulfills the task set out in Coloring Step 4. So this concludes the proof of Lemma 7.2.7.

7.3 Solving the recurrence

Our goal in this section is to prove the following statement, which will be later combined with Lemma 7.2.7.

Lemma 7.3.1. *Suppose $f_2, f_3, \dots$ are functions from $\mathbb{Z}_{>0}$ to $\mathbb{Z}_{>0}$ satisfying the following:*

- *$f_2(n) \leqslant n$ for all integers $n \geqslant 1$;*
- *$f_d(1) = 1$ for all integers $d \geqslant 2$; and*
- *for every integer $d \geqslant 3$ there exists $\alpha_d \in \mathbb{N}$ such that for every integer $n \geqslant 8$,*

$$f_d(n) \leqslant \alpha_d \left[f_d(\lceil 7n/8 \rceil) + f_{d-1}^2(2n^{d-1}) \cdot \sum_{u=0}^{\lfloor \log n \rfloor - 3} f_d(2^{u+1}) \cdot f_d\left(\left\lceil \frac{n}{2^{u+1}} \right\rceil + 1 \right) \right]. \tag{7.4}$$

Then for every integer $d \geqslant 2$ there exists a constant $\beta_d \in \mathbb{N}$ such that

$$f_d(n) \leqslant 2^{\beta_d \log^{d-1} n} \qquad \text{for all } n \in \mathbb{Z}_{>0}. \tag{7.5}$$

Proof. We proceed by induction on d. Clearly, for $d = 2$ we may choose $\beta_2 = 1$.

In the induction step for $d \geqslant 3$, we choose β_d to be large enough so that the following inequalities are satisfied:

$$2^{\beta_d} \geqslant f_d(i) \quad \text{for all } i \in \{1, \dots, 2^{16} - 1\}, \tag{7.6}$$

$$\beta_d \cdot \frac{d-1}{2^{d+1}} \geqslant 3\beta_{d-1} d^{d-2}, \tag{7.7}$$

$$2^{\beta_d} \geqslant 2\alpha_d. \tag{7.8}$$

We now verify Eq. (7.5) by induction on n. The base cases $n \in \{1, 2, \dots, 2^{16} - 1\}$ hold trivially thanks to Eq. (7.6), so from now on let us assume that $n \geqslant 2^{16}$.

Let

$$g_d(x) \coloneqq 2^{\beta_d \log^{d-1} x}$$

and denote the right-hand side of Eq. (7.4) by RHS. Using both induction assumptions, we have

$$\mathsf{RHS} \leqslant \alpha_d \left[g_d(\lceil 7n/8 \rceil) + g_{d-1}^2(2n^{d-1}) \cdot \sum_{u=0}^{\lfloor \log n \rfloor - 3} g_d(2^{u+1}) \cdot g_d\left(\left\lceil \frac{n}{2^{u+1}} \right\rceil + 1 \right) \right]. \tag{7.9}$$

In order to estimate each summand of the form $g_d(2^{u+1}) \cdot g_d\left(\left\lceil \frac{n}{2^{u+1}} \right\rceil + 1\right)$, we use the following fact:

Claim 7.3.2. *For all reals $a, b \geqslant 2$, we have*

$$g_d(a) \cdot g_d(b) \leqslant g_d(2) \cdot g_d(ab/2).$$

Proof. Assume without loss of generality that $a \leqslant b$. Define the following values: $x_1 \coloneqq \log 2 = 1$, $x_2 \coloneqq \log a$, $x_3 \coloneqq \log b$, $x_4 \coloneqq \log(ab/2)$. Then $0 < x_1 \leqslant x_2 \leqslant x_3 \leqslant x_4$ and $x_1 + x_4 = x_2 + x_3$. By the convexity of the function $x \mapsto x^{d-1}$ on $\mathbb{R}_{>0}$, we have that[20] $x_1^{d-1} + x_4^{d-1} \geqslant x_2^{d-1} + x_3^{d-1}$, or equivalently:

$$\log^{d-1}(ab/2) + \log^{d-1} 2 \geqslant \log^{d-1} a + \log^{d-1} b.$$

Therefore $\log g_d(a) + \log g_d(b) \leqslant \log g_d(2) + \log g_d(ab/2)$, as required. □

Applying Claim 7.3.2 for $u \in \{0, 1, \dots, \lfloor \log n \rfloor - 3\}$, we find that

$$g_d(2^{u+1}) \cdot g_d\left(\left\lceil \frac{n}{2^{u+1}} \right\rceil + 1 \right) \leqslant 2^{\beta_d} \cdot g_d\left(2^u \left\lceil \frac{n}{2^{u+1}} \right\rceil + 2^u \right) \leqslant 2^{\beta_d} \cdot g_d\left(n/2 + 2^{u+1} \right) \leqslant 2^{\beta_d} \cdot g_d(3n/4).$$

By combining this with Eq. (7.9) and observing that for $n \geqslant 2^{16}$ we have $\lceil 7n/8 \rceil \leqslant 15n/16$, we conclude that

$$\mathsf{RHS} \leqslant \alpha_d \left[g_d(15n/16) + g_{d-1}^2(2n^{d-1}) \cdot n \cdot 2^{\beta_d} \cdot g_d(3n/4) \right]. \tag{7.10}$$

In the estimation of the right hand side of Eq. (7.10) we will need the following simple claim.

[20] This follows e.g. from Karamata's inequality [Kar32].

Claim 7.3.3. *For all reals $\varepsilon \in [0, 1/2]$ we have*

$$(1-\varepsilon)^{d-1} \leqslant 1 - \frac{d-1}{2^{d-2}} \cdot \varepsilon.$$

Proof. Let $h(t) = (1+t)^{d-1} - \frac{d-1}{2^{d-2}} \cdot t - 1$. Observe that for $t \in [-1/2, 0]$, we have

$$h'(t) = (d-1)(1+t)^{d-2} - \frac{d-1}{2^{d-2}} \geqslant \frac{d-1}{2^{d-2}} - \frac{d-1}{2^{d-2}} = 0.$$

Since $h(0) = 0$, it follows that $h(t) \leqslant 0$ for $t \in [-1/2, 0]$; this is equivalent to the claim. ◁ □

First, let

$$\nu \coloneqq g_d(15n/16).$$

Observe that, by Claim 7.3.3,

$$\begin{aligned} \log \nu &= \beta_d (\log n - \log 16/15)^{d-1} = \beta_d \log^{d-1} n \left(1 - \frac{\log 16/15}{\log n}\right)^{d-1} \\ &\leqslant \beta_d \log^{d-1} n \left(1 - \frac{(d-1)\log 16/15}{2^{d-2}} \cdot \frac{1}{\log n}\right) \leqslant \beta_d \log^{d-1} n - \frac{(d-1)\beta_d}{2^{d+2}} \log^{d-2} n. \end{aligned} \tag{7.11}$$

Next, let

$$\mu \coloneqq g_{d-1}^2(2n^{d-1}) \cdot n \cdot 2^{\beta_d} \cdot g_d(3n/4).$$

Observe that

$$\begin{aligned} \log \mu &= 2\beta_{d-1} \log^{d-2}(2n^{d-1}) + \log n + \beta_d + \beta_d \log^{d-1}(3n/4) \\ &\leqslant 2\beta_{d-1} d^{d-2} \log^{d-2} n + \log n + \beta_d + \beta_d (\log n - \log 4/3)^{d-1} \\ &\leqslant 3\beta_{d-1} d^{d-2} \log^{d-2} n + \beta_d + \beta_d (\log n - \log 4/3)^{d-1}. \end{aligned}$$

By Claim 7.3.3, we have

$$\begin{aligned} (\log n - \log 4/3)^{d-1} &= \log^{d-1} n \cdot \left(1 - \frac{\log 4/3}{\log n}\right)^{d-1} \\ &\leqslant \log^{d-1} n \cdot \left(1 - \frac{(d-1)\log 4/3}{2^{d-2}} \cdot \frac{1}{\log n}\right) \leqslant \log^{d-1} n - \frac{d-1}{2^d} \cdot \log^{d-2} n. \end{aligned}$$

Therefore,

$$\begin{aligned} \log \mu &\leqslant \beta_d \log^{d-1} n + \beta_d + \log^{d-2} n \cdot \left(3\beta_{d-1} d^{d-2} - \beta_d \cdot \frac{d-1}{2^d}\right) \\ &\leqslant \beta_d \log^{d-1} n + \beta_d - \frac{(d-1)\beta_d}{2^{d+1}} \log^{d-2} n, \end{aligned}$$

where in the last inequality we used Eq. (7.7). Furthermore, since $n \geqslant 2^{16}$ and $d \geqslant 3$, we have

$$\frac{d-1}{2^{d+1}} \log^{d-2} n \geqslant \frac{(d-1) \cdot 16^{d-2}}{2^{d+1}} \geqslant \frac{2 \cdot 2^{4d-8}}{2^{d+1}} = 2^{3d-8} \geqslant 2,$$

hence

$$\log \mu \leqslant \beta_d \log^{d-1} n - \frac{(d-1)\beta_d}{2^{d+2}} \log^{d-2} n. \tag{7.12}$$

We now combine Eqs. (7.11) and (7.12) with Eq. (7.10), thus obtaining:

$$\begin{aligned} \mathsf{RHS} &\leqslant \alpha_d \left[2^{\log \nu} + 2^{\log \mu}\right] \\ &\leqslant \alpha_d \cdot 2^{\beta_d \log^{d-1} n} \cdot \left[2^{-\frac{(d-1)\beta_d}{2^{d+2}} \log^{d-2} n} + 2^{-\frac{(d-1)\beta_d}{2^{d+2}} \log^{d-2} n}\right] \\ &= 2^{\beta_d \log^{d-1} n} \cdot \frac{2\alpha_d}{2^{\frac{(d-1)\beta_d}{2^{d+2}} \log^{d-2} n}} \leqslant 2^{\beta_d \log^{d-1} n} \cdot \frac{2\alpha_d}{2^{\frac{(d-1)\beta_d}{2^{d+2}} 16^{d-2}}} \\ &\leqslant 2^{\beta_d \log^{d-1} n} \cdot \frac{2\alpha_d}{2^{\beta_d \cdot 2^{3d-9}}} \leqslant 2^{\beta_d \log^{d-1} n} \cdot \frac{2\alpha_d}{2^{\beta_d}} \leqslant 2^{\beta_d \log^{d-1} n} = g_d(n), \end{aligned}$$

where the last inequality follows from Eq. (7.8). As $f_d(n) \leqslant \mathsf{RHS}$, this concludes the proof. □

7.4 Wrapping up the proof

In this section we combine Lemmas 7.3.1 and 7.2.7 to obtain a proof of Theorem 1.3.10. However, since the statement of Lemma 7.2.7 assumes $d \geqslant 3$, we need to consider the base case $d = 2$ separately. This is provided by the following lemma. Recall here that a *cograph* is a P_4-free graph [CLB81], that is, a graph that does not contain the path on 4 vertices as an induced subgraph.

Lemma 7.4.1. *Let G be a graph that admits, under some vertex ordering, a 2-almost mixed-free adjacency matrix. Then G is a cograph.*

Proof. By contraposition and the fact that 2-almost mixed-freeness is preserved under taking submatrices, it suffices to show that no vertex ordering of a P_4 yields a 2-almost mixed-free adjacency matrix. Observe that if one partitions the vertex set of a P_4 into two parts of size 2, then regardless of the choice of the partition, no part will be semi-pure towards the other. Therefore, for every vertex ordering of a P_4, dividing the corresponding adjacency matrix into four 2×2 matrices yields a 2-almost mixed minor. □

It is well known that cographs are *perfect*, that is, $\chi(G) = \omega(G)$ whenever G is a cograph. Hence, from Lemma 7.4.1 we conclude that

$$f_2(\omega) \leqslant \omega \qquad \text{for all } \omega \in \mathbb{Z}_{>0}.$$

Furthermore, we clearly have $f_d(1) = 1$ for every $d \geqslant 2$, since graphs of clique number 1 are edgeless. Finally, applying Lemma 7.2.7 for $k = \lfloor \omega/8 \rfloor$ yields that for all $\omega \geqslant 8$, $f_d(\omega)$ is upper bounded by

$$\begin{aligned} & f_d(\lceil 7\omega/8 \rceil) + C_d \left[f_d(\lceil 7\omega/8 \rceil) + 8C_d f_{d-1}^2(2\omega^{d-1}) \cdot \sum_{u=0}^{\lfloor \log_2 \omega/8 \rfloor} f_d(2^{u+1}) \cdot f_d\left(\left\lceil \frac{\omega}{2^{u+2}} \right\rceil + 1 \right) \right] \\ \leqslant \quad & 8C_d(C_d+1) \cdot \left[f_d(\lceil 7\omega/8 \rceil) + f_{d-1}^2(2\omega^{d-1}) \cdot \sum_{u=0}^{\lfloor \log \omega \rfloor - 3} f_d(2^{u+1}) \cdot f_d\left(\left\lceil \frac{\omega}{2^{u+1}} \right\rceil + 1 \right) \right]. \end{aligned}$$

We may now apply Lemma 7.3.1 to functions $f_2, f_3, \ldots$ to conclude the following.

Theorem 7.4.2. *For every integer $d \geqslant 2$ there is a constant $\beta_d \in \mathbb{N}$ such that $f_d(\omega) \leqslant 2^{\beta_d \cdot \log^{d-1} \omega}$ for every $\omega \in \mathbb{Z}_{>0}$. In other words, for every graph G that has clique number ω and admits a d-almost mixed-free adjacency matrix under some vertex ordering, we have $\chi(G) \leqslant 2^{\beta_d \cdot \log^{d-1} \omega}$.*

Theorem 1.3.10 now follows from combining Theorem 7.4.2 with Lemma 7.2.6.

7.5 Conclusions

In this chapter we showed that every class of graphs of bounded twin-width is quasi-polynomially χ-bounded. This result has raised a natural question – whether the χ-bounding function for any class of graphs of bounded twin-width could be optimized to polynomial. This question has since been resolved positively by Bourneuf and Thomassé [BT23].

7.5.1 Polynomial χ-boundedness

We now follow with an overview of the proof of polynomial χ-boundedness of classes of graphs of bounded twin-width by Bourneuf and Thomassé [BT23] and compare their arguments to our techniques.

The authors borrow our idea of performing the structural induction on the size of the largest almost mixed minor in the adjacency matrix of the graph. As in our setting, given a graph G of twin-width t, they fix an ordering $v_1, \ldots, v_n$ of vertices of the graph so that the adjacency matrix M of G, with rows and columns arranged according to this ordering, is d-almost mixed-free for $d := 4t + 4$.

Next, we construct a decomposition tree T, called the *delayed decomposition* of G, as a rooted tree whose nonleaf nodes are identified with *intervals* of $V(G)$, i.e., sets of the form $\{v_i, \ldots, v_j\}$ for $i \leqslant j$, and whose set of leaf nodes is precisely $V(G)$. The root of the decomposition is a nonleaf node identified with the interval $V(G) = \{v_1, \ldots, v_n\}$. Then, recursively, given a nonleaf node $x \in V(T)$, identified with an interval $I = \{v_\ell, \ldots, v_r\}$:

- If $I = \{v_i\}$ is a singleton, we put v_i as the only child of x.

- Otherwise, if is identified with an interval $I = \{v_\ell, \dots, v_r\}$ that is a module in G (i.e., $N(u) \setminus I = N(v) \setminus I$ for all $u, v \in I$), we split I into two roughly equal parts. That is, we set $m \coloneqq \lfloor \frac{\ell+r}{2} \rfloor$ and create two children y_1, y_2 of x identified with intervals $\{v_\ell, \dots, v_m\}$ and $\{v_{m+1}, \dots, v_r\}$, respectively.
- Otherwise, let $P(I)$ be the partition of I into the maximal intervals of vertices with the same neighborhood in $V(G) \setminus I$. We create $|P(I)|$ children of x identified with respective elements of $P(I)$.

Observe that the construction above guarantees the following property: Suppose two nodes z, z' of T – identified with subsets $I_z, I_{z'}$ of $V(G)$, respectively – are *cousins* – i.e., they share their grandparent, but not their parent. Then the pair $I_z, I_{z'}$ is pure in $V(G)$. Therefore, we can encode the edges of G in the delayed decomposition T by additionally specifying a function g that assigns to each nonleaf node $x \in V(T)$ a graph G_x with vertex set equal to the set of grandchildren of x in T. In G_x, two cousins z, z' – identified with intervals $I_z, I_{z'}$ of $V(G)$ – are connected by an edge if the pair $I_z, I_{z'}$ is complete. If z, z' have the same parent, we define that there is no edge between z and z' in G_x. It follows from our discussion that G uniquely determines a pair (T, g), which we will call the *augmented delayed decomposition*. Conversely, as long as some graph is consistent with an augmented delayed decomposition (T, g), such a graph is unique: Let $u, v \in V(G)$ be two different vertices of G and let x be the lowest common ancestor of u and v in T. By construction, x is not a parent of either u and v, so let z_u and z_v be the grandchildren of x that are ancestors of u and v, respectively. Then z_u and z_v are cousins in T and the adjacency between u and v in G can be determined by testing whether $z_u z_v \in E(G_x)$.

The last case in the construction of T above – the partitioning of an interval I of vertices into maximal intervals of twins with respect to $\overline{I}$ – is inspired by our partitioning of blobs into subblobs. This is not coincidental: In our proof, we have shown that the graphs representing the interactions between subblobs in a blob are structurally simpler (see Lemma 7.2.12). In turns out that an analog of this property also holds in delayed decompositions.

Proposition 7.5.1 (Implicit in [BT23]). *Suppose the adjacency matrix M of G is d-almost mixed-free with $d \geqslant 3$. Let $G = \{v_1, \dots, v_n\}$ and (T, g) be the augmented delayed decomposition of G. Let x be a nonleaf node of T with children $y_1, \dots, y_k$, identified with intervals $I_1, \dots, I_k$ of $V(G)$, respectively. Let $G_x = g(x)$. Define the graphs G_x^{M}, G_x^{H}, G_x^{V} with vertex set $\{y_1, \dots, y_k\}$ such that, for $1 \leqslant i < j \leqslant k$:*

- *$v_i v_j \in E(G_x^{\mathsf{M}})$ if the connection between I_i and I_j in G is mixed, i.e., $M[I_i][I_j]$ is mixed;*
- *$v_i v_j \in E(G_x^{\mathsf{H}})$ if I_i, I_j is nonempty and I_i is semi-pure towards I_j, i.e., $M[I_i][I_j]$ is nonzero horizontal;*
- *$v_i v_j \in E(G_x^{\mathsf{V}})$ if I_i, I_j is nonempty and I_j is semi-pure towards I_i, i.e., $M[I_i][I_j]$ is nonzero vertical.*

Then G_x^{M} is $\mathcal{O}_d(1)$-colorable, and both G_x^{H} and G_x^{V} can be vertex-partitioned into $\mathcal{O}_d(1)$ induced subgraphs whose adjacency matrices are $(d-1)$-almost mixed-free. Moreover, $\omega(G_x^{\mathsf{H}}), \omega(G_x^{\mathsf{V}}) \leqslant \omega(G_x)^d$.

Sketch of the proof. The claim for G_x^{M} is immediate from the Marcus–Tardos theorem. The properties of G_x^{H}, G_x^{V} are proved similarly to Lemma 7.2.4 and Lemma 7.2.12, only that the proofs in [BT23] are streamlined and presented in a more general setting of *right extensions* of classes of graphs. □

The main idea behind delayed decompositions of graphs is that, in contrast to our approach, a graph is decomposed into smaller pieces recursively until it is ultimately split into one-vertex subgraphs. Crucially, polynomial χ-boundedness is preserved by delayed decompositions in the following sense:

Proposition 7.5.2 ([BT23]). *For a hereditary class $\mathcal{C}$ of graphs, define the* delayed extension *$\mathcal{C}_d$ of $\mathcal{C}$ as the hereditary class of graphs consisting of graphs G admitting a delayed decomposition (T, g) such that every graph in g belongs to $\mathcal{C}$. If $\mathcal{C}$ is polynomially χ-bounded, then so is $\mathcal{C}_d$.*

The proof of Proposition 7.5.2 actually shows the following statement: Every graph $G \in \mathcal{C}_d$ is an edge-sum of two graphs in the *substitution closure* $\mathcal{C}_s$ of $\mathcal{C}$. It is proved by Chudnovsky, Penev, Scott, and Trotignon [CPST13] that the polynomial χ-boundedness of $\mathcal{C}$ implies the polynomial χ-boundedness of $\mathcal{C}_s$; hence in such a case, $\mathcal{C}_d$ is also polynomially χ-bounded.

The polynomial χ-boundedness of classes of graphs of bounded twin-width now follows straightforwardly from Propositions 7.5.1 and 7.5.2:

Corollary 7.5.3 ([BT23]). *Let $\mathcal{A}_d$, $d \geqslant 2$, be the class of graphs whose adjacency matrix is d-almost mixed-free. Then $\mathcal{A}_d$ is polynomially χ-bounded.*

Proof. For $d \geqslant 2$, let $f_d(\omega)$ be the χ-bounding function of $\mathcal{A}_d$. Every graph in $\mathcal{A}_2$ is a cograph (Lemma 7.4.1), so $\mathcal{A}_2$ is a class of perfect graphs.

For $d \geqslant 3$, let $\mathcal{B}_d$ be the class of graphs containing every graph $g(x)$ for the delayed decomposition (T, g) of any graph $G \in \mathcal{A}_d$ and any nonleaf node $x \in V(T)$. Observe that every graph in $\mathcal{B}_d$ can be properly

colored using $f_{d-1}(\omega)^{\mathcal{O}_d(1)}$ colors: Let $H \in \mathcal{B}_d$ and let $H = g(x)$ for the delayed decomposition (T, g) of a graph $G \in \mathcal{A}_d$ and a nonleaf node $x \in V(T)$. Let $y_1, \ldots, y_k$ be the children of x in T, and $Z_1, \ldots, Z_k$ be the sets of children of $y_1, \ldots, y_k$ in T, respectively. By construction, each induced subgraph $H[Z_i]$ is edgeless. For $t \in \{\mathsf{M}, \mathsf{H}, \mathsf{V}\}$, let ϕ^t be the proper coloring of G_x^t, where $G_x^{\mathsf{M}}, G_x^{\mathsf{H}}, G_x^{\mathsf{V}}$ are defined as in Proposition 7.5.1. Observe now that the coloring of H that assigns to every node $z \in Z_i$ the product coloring $(\phi^{\mathsf{M}}(y_i), \phi^{\mathsf{H}}(y_i), \phi^{\mathsf{V}}(y_i))$ is proper. Hence $\chi(H) \leqslant \chi(G_x^{\mathsf{M}})\chi(G_x^{\mathsf{H}})\chi(G_x^{\mathsf{V}})$. But now from Proposition 7.5.1 we have that $\chi(G_x^{\mathsf{M}}) \leqslant \mathcal{O}_d(1)$, $\chi(G_x^{\mathsf{H}}) \leqslant \mathcal{O}_d(1) \cdot f_{d-1}(\omega(G_x^{\mathsf{H}}))$ and $\chi(G_x^{\mathsf{V}}) \leqslant \mathcal{O}_d(1) \cdot f_{d-1}(\omega(G_x^{\mathsf{V}}))$; hence $\chi(H) \leqslant f_{d-1}(\omega(H))^{\mathcal{O}(d)}$ by induction and the fact that $\omega(G_x^{\mathsf{H}}), \omega(G_x^{\mathsf{V}}) \leqslant \omega(H)^d$. Hence $\mathcal{B}_d$ is polynomially χ-bounded. But then $\mathcal{A}_d$ is also polynomially χ-bounded thanks to Proposition 7.5.2. □

7.5.2 Algorithmic application: Decomposing pattern-free permutations

We now briefly describe the algorithmic result of Bonnet, Bourneuf, Geniet, and Thomassé [BBGT24] inspired by the notion of delayed decompositions of graphs: Pattern-free permutations can be decomposed into a bounded number of simple pieces.

For a permutation π of $\{1, \ldots, n\}$, we define the permutation graph G_π of π as the graph with vertex set $\{1, \ldots, n\}$, where $ij \in E(G_\pi)$ for $i < j$ if i, j are swapped by π, i.e., $\pi(i) > \pi(j)$. Let also M_π – called the *adjacency matrix* of π – be the adjacency matrix of G_π with the natural order of rows and columns. Then a permutation π is *d-almost mixed-free* if M_π is d-almost mixed free. Next, a permutation π is *separable* if it excludes two permutations $(2, 4, 1, 3)$ and $(3, 1, 4, 2)$ as subpatterns; equivalently, π is separable if its permutation graph is a cograph. It can be shown that every 2-almost mixed free permutation is separable; however, converse is not necessarily true, since $(2, 3, 1, 4)$ is separable but not 2-almost mixed free.

We then state the result of Bonnet et al. below:

Theorem 7.5.4 ([BBGT24], stronger form of Theorem 1.3.12)**.** *Let σ be a fixed pattern that is a permutation of $\{1, \ldots, k\}$. Then there exists an integer $c_k \in \mathbb{N}$ such that every permutation π of $\{1, \ldots, n\}$ that excludes σ as a subpattern is a product of at most c_k permutations that are 2-almost mixed-free. Moreover, this product can be found in time $\mathcal{O}_k(n)$.*

The proof of Theorem 7.5.4 again involves structural induction on the size of the largest almost mixed minor in a matrix; this time, the adjacency matrix of a permutation is considered. Unfortunately, it may happen that an adjacency matrix of a pattern-free permutation has a very large almost mixed minor: Take for example *even-odd shuffles* $\pi_1, \pi_2, \ldots$, where π_n is the permutation of $\{1, \ldots, 2n\}$ defined as $\pi_n = (1, 3, \ldots, 2n-1, 2, 4, \ldots, 2n)$. Then it can be shown that all even-odd shuffles exclude the permutation $(3, 2, 1)$ as a subpattern, but the adjacency matrix of σ_n contains a $\lfloor \frac{n}{2} \rfloor$-mixed minor. Fortunately, pattern-free permutations can be represented as a product of two almost mixed-free permutations:

Lemma 7.5.5 ([BBGT24])**.** *Fix a pattern σ of $\{1, \ldots, k\}$. Then every permutation π excluding σ as a subpattern can be represented as a product $\pi = \pi_1 \circ \pi_2$, where each π_i is $2^{\mathcal{O}(k)}$-almost mixed-free. Moreover, given π on input, we can find π_1, π_2 in time $\mathcal{O}_k(|\pi|)$.*

Now the proof of Theorem 7.5.4 proceeds roughly as follows: We decompose each π_1, π_2 separately. Given a permutation π whose adjacency matrix is d-almost mixed-free ($d \geqslant 3$), we construct an augmented delayed decomposition (T, g) of G_π; this is easy to do in time $|\pi|^{\mathcal{O}(1)}$ by simply following the definition of (T, g), but the construction time can be nontrivially optimized to $\mathcal{O}(|\pi|)$ by exploiting structural properties of G_π. Then, by analyzing the properties of delayed decompositions of permutation graphs (in particular, using properties similar to Proposition 7.5.1), we can prove that each such π can be represented as a product of $d^{\mathcal{O}(1)}$ permutations, each of which is $(d-1)$-almost mixed free.

Theorem 7.5.4 is interesting from the point of view of compact data structures for twin-width since it provides a "succinct" representation of pattern-free permutations as products of very simple 2-almost mixed-free permutations. It is thus natural to claim that:

Conjecture 7.5.6 (Compact data structure for pattern-free permutations)**.** *Let σ be a fixed pattern that is a permutation of $\{1, \ldots, k\}$. One can construct a data structure representing a permutation π of $\{1, \ldots, n\}$ excluding σ as a subpattern that occupies $\mathcal{O}_k(n)$ bits. The data structure can be queried for the values of π in worst-case time $\mathcal{O}_k(1)$. The construction time of the data structure is $\mathcal{O}_k(n)$.*

Note that it is enough to show that Conjecture 7.5.6 holds just for 2-almost mixed-free permutations; then Conjecture 7.5.6 will quickly follow from Theorem 7.5.4. This subject of a work in progress. After proving Conjecture 7.5.6, a natural next step would be to generalize the result to provide a compact data structure with constant query time for arbitrary matrices of bounded twin-width:

Conjecture 7.5.7. *Let $d \in \mathbb{N}$ be a fixed constant. Then for a given binary $n \times n$ matrix M that is d-twin-ordered one can construct a data structure that occupies $\mathcal{O}_d(n)$ bits and can be queried for entries of M in worst-case time $\mathcal{O}_d(1)$ per query.*

When proven, Conjecture 7.5.7 would improve upon our previous result Theorem 1.3.8 proved in Chapter 6.

Bibliography

[ABF+02] Jochen Alber, Hans L. Bodlaender, Henning Fernau, Ton Kloks, and Rolf Niedermeier. Fixed Parameter Algorithms for DOMINATING SET and Related Problems on Planar Graphs. *Algorithmica*, 33(4):461–493, 2002.

[ABMN18] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. *CoRR*, abs/1812.09519, 2018.

[ABMN19] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. In *38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019*, pages 89–103. ACM, 2019.

[ACK17] Ittai Abraham, Shiri Chechik, and Sebastian Krinninger. Fully dynamic all-pairs shortest paths with worst-case update-time revisited. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 440–452. SIAM, 2017.

[ACP87] Stefan Arnborg, Derek C. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Alg. Disc. Meth.*, 8:277–284, 1987.

[AFN04] Jochen Alber, Michael R. Fellows, and Rolf Niedermeier. Polynomial-time data reduction for dominating set. *J. ACM*, 51(3):363–384, 2004.

[AHdLT05] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, 2005.

[AKK+17] Isolde Adler, Stavros G. Kolliopoulos, Philipp Klaus Krause, Daniel Lokshtanov, Saket Saurabh, and Dimitrios M. Thilikos. Irrelevant vertices for the planar Disjoint Paths Problem. *J. Comb. Theory, Ser. B*, 122:815–843, 2017.

[ALS91] Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy Problems for Tree-Decomposable Graphs. *J. Algorithms*, 12(2):308–340, 1991.

[AMV20] Josh Alman, Matthias Mnich, and Virginia Vassilevska Williams. Dynamic parameterized problems and algorithms. *ACM Trans. Algorithms*, 16(4):45:1–45:46, 2020.

[AP89] Stefan Arnborg and Andrzej Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discret. Appl. Math.*, 23(1):11–24, 1989.

[Arn85] Stefan Arnborg. Efficient Algorithms for Combinatorial Problems with Bounded Decomposability - A survey. *BIT*, 25(1):1–23, 1985.

[AYZ95] Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.

[Bag06] Guillaume Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *20th International Workshop on Computer Science Logic, CSL 2006*, volume 4207 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2006.

[Bak94] Brenda S. Baker. Approximation algorithms for NP-complete problems on planar graphs. *J. ACM*, 41(1):153–180, 1994.

[BB73] Umberto Bertelè and Francesco Brioschi. On Non-serial Dynamic Programming. *J. Comb. Theory, Ser. A*, 14(2):137–148, 1973.

[BBD22] Pierre Bergé, Édouard Bonnet, and Hugues Déprés. Deciding Twin-Width at Most 4 Is NP-Complete. In Mikolaj Bojańczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 18:1–18:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[BBGT24] Édouard Bonnet, Romain Bourneuf, Colin Geniet, and Stéphan Thomassé. Factoring Pattern-Free Permutations into Separable ones. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 752–779, 2024.

[BBL13] Hans L. Bodlaender, Paul Bonsma, and Daniel Lokshtanov. The fine details of fast dynamic programming over tree decompositions. In Gregory Gutin and Stefan Szeider, editors, *Parameterized and Exact Computation*, pages 41–53, Cham, 2013. Springer International Publishing.

[BCK+22] Édouard Bonnet, Dibyayan Chakraborty, Eun Jung Kim, Noleen Köhler, Raul Lopes, and Stéphan Thomassé. Twin-width VIII: delineation and win-wins. In Holger Dell and Jesper Nederlof, editors, *17th International Symposium on Parameterized and Exact Computation, IPEC 2022, September 7-9, 2022, Potsdam, Germany*, volume 249 of *LIPIcs*, pages 9:1–9:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[BD02] Patrick Bellenbaum and Reinhard Diestel. Two short proofs concerning tree-decompositions. *Comb. Probab. Comput.*, 11(6):541–547, 2002.

[BDD+16] Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michał Pilipczuk. A $c^k n$ 5-approximation algorithm for treewidth. *SIAM Journal on Computing*, 45(2):317–378, 2016.

[BF99] Gerth Stølting Brodal and Rolf Fagerberg. Dynamic representation of sparse graphs. In *Algorithms and Data Structures, 6th International Workshop, WADS*, volume 1663 of *Lecture Notes in Computer Science*, pages 342–351. Springer, 1999.

[BFL+16] Hans L. Bodlaender, Fedor V. Fomin, Daniel Lokshtanov, Eelko Penninkx, Saket Saurabh, and Dimitrios M. Thilikos. (Meta) Kernelization. *J. ACM*, 63(5):44:1–44:69, 2016.

[BFLP24] Édouard Bonnet, Florent Foucaud, Tuomo Lehtilä, and Aline Parreau. Neighbourhood complexity of graphs of bounded twin-width. *Eur. J. Comb.*, 115:103772, 2024.

[BG91] Jonathan F. Buss and Judy Goldsmith. Nondeterminism within p. In Christian Choffrut and Matthias Jantzen, editors, *STACS 91*, pages 348–359, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[BGdM+21] Édouard Bonnet, Ugo Giocanti, Patrice Ossona de Mendez, Pierre Simon, Stéphan Thomassé, and Szymon Toruńczyk. Twin-width IV: ordered graphs and matrices. *CoRR*, abs/2102.03117, 2021.

[BGK+21a] Édouard Bonnet, Colin Geniet, Eun Jung Kim, Stéphan Thomassé, and Rémi Watrigant. Twin-width II: small classes. In *2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*, pages 1977–1996. SIAM, 2021.

[BGK+21b] Édouard Bonnet, Colin Geniet, Eun Jung Kim, Stéphan Thomassé, and Rémi Watrigant. Twin-width III: Max Independent Set, Min Dominating Set, and Coloring. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021*, volume 198 of *LIPIcs*, pages 35:1–35:20. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2021.

[BGR24] Jakub Balabán, Robert Ganian, and Mathis Rocton. Computing Twin-Width Parameterized by the Feedback Edge Number. In Olaf Beyersdorff, Mamadou Moustapha Kanté, Orna Kupferman, and Daniel Lokshtanov, editors, *41st International Symposium on Theoretical Aspects of Computer Science (STACS 2024)*, volume 289 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:19, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[BH98] Hans L. Bodlaender and Torben Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. *SIAM J. Comput.*, 27(6):1725–1746, 1998.

[BH21] Jakub Balabán and Petr Hlinený. Twin-width is linear in the poset width. In *Proceedings of the 16th International Symposium on Parameterized and Exact Computation, IPEC 2021*, volume 214 of *LIPIcs*, pages 6:1–6:13. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2021.

[BK96] Hans L. Bodlaender and Ton Kloks. Efficient and Constructive Algorithms for the Pathwidth and Treewidth of Graphs. *J. Algorithms*, 21(2):358–402, 1996.

[BKR+21] Édouard Bonnet, Eun Jung Kim, Amadeus Reinald, Stéphan Thomassé, and Rémi Watrigant. Twin-width and polynomial kernels. *CoRR*, abs/2107.02882, 2021.

[BKTW20] Édouard Bonnet, Eun Jung Kim, Stéphan Thomassé, and Rémi Watrigant. Twin-width I: tractable FO model checking. In *IEEE 61st Annual Symposium on Foundations of Computer Science, FOCS 2020*, pages 601–612. IEEE Computer Society, 2020.

[BKW22] Édouard Bonnet, O-joung Kwon, and David R. Wood. Reduced bandwidth: a qualitative strengthening of twin-width in minor-closed classes (and beyond). *CoRR*, abs/2202.11858, 2022.

[Bod93a] Hans L. Bodlaender. Dynamic algorithms for graphs with treewidth 2. In *19th International Workshop on Graph-Theoretic Concepts in Computer Science, , WG 1993*, volume 790 of *Lecture Notes in Computer Science*, pages 112–124. Springer, 1993.

[Bod93b] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–21, 1993.

[Bod96] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.

[BP20] Marthe Bonamy and Michał Pilipczuk. Graphs of bounded cliquewidth are polynomially χ-bounded. *Advances in Combinatorics*, (2020:8), 2020.

[BP22] Mikołaj Bojańczyk and Michał Pilipczuk. Optimizing tree decompositions in MSO. *Log. Methods Comput. Sci.*, 18(1), 2022.

[BT23] Romain Bourneuf and Stéphan Thomassé. Bounded twin-width graphs are polynomially χ-bounded. *CoRR*, abs/2303.11231, 2023.

[BTV10] Binh-Minh Bui-Xuan, Jan Arne Telle, and Martin Vatshelle. H-join decomposable graphs and algorithms with runtime single exponential in rankwidth. *Discret. Appl. Math.*, 158(7):809–819, 2010.

[CC08] Miroslav Chlebík and Janka Chlebíková. Approximation hardness of dominating set problems in bounded degree graphs. *Inf. Comput.*, 206(11):1264–1275, 2008.

[CCD+20] Jiehua Chen, Wojciech Czerwiński, Yann Disser, Andreas Emil Feldmann, Danny Hermelin, Wojciech Nadara, Marcin Pilipczuk, Michał Pilipczuk, Manuel Sorge, Bartlomiej Wróblewski, and Anna Zych-Pawlewicz. Efficient fully dynamic elimination forests with applications to detecting long paths and cycles. *CoRR*, abs/2006.00571, 2020. Full version of the SODA 2021 paper.

[CCD+21] Jiehua Chen, Wojciech Czerwiński, Yann Disser, Andreas Emil Feldmann, Danny Hermelin, Wojciech Nadara, Marcin Pilipczuk, Michał Pilipczuk, Manuel Sorge, Bartlomiej Wróblewski, and Anna Zych-Pawlewicz. Efficient fully dynamic elimination forests with applications to detecting long paths and cycles. In *2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*, pages 796–809. SIAM, 2021.

[CE12] Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic — A Language-Theoretic Approach*, volume 138 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 2012.

[CER93] Bruno Courcelle, Joost Engelfriet, and Grzegorz Rozenberg. Handle-Rewriting Hypergraph Grammars. *J. Comput. Syst. Sci.*, 46(2):218–270, 1993.

[CFK+15] Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

[Cha13] Timothy M. Chan. Persistent Predecessor Search and Orthogonal Point Location on the Word RAM. *ACM Trans. Algorithms*, 9(3):22:1–22:22, 2013.

[CJ03] Liming Cai and David W. Juedes. On the existence of subexponential parameterized algorithms. *J. Comput. Syst. Sci.*, 67(4):789–807, 2003.

[CK07] Bruno Courcelle and Mamadou Moustapha Kanté. Graph Operations Characterizing Rank-Width and Balanced Graph Expressions. In *33rd International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2007*, volume 4769 of *Lecture Notes in Computer Science*, pages 66–75. Springer, 2007.

[CK19] Josef Cibulka and Jan Kynčl. Better upper bounds on the Füredi-Hajnal limits of permutations, 2019.

[CKX10] Jianer Chen, Iyad A. Kanj, and Ge Xia. Improved upper bounds for vertex cover. *Theor. Comput. Sci.*, 411(40-42):3736–3756, 2010.

[CLB81] D.G. Corneil, H. Lerchs, and L.Stewart Burlingham. Complement reducible graphs. *Discrete Applied Mathematics*, 3(3):163–174, 1981.

[CLP11] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal Range Searching on the RAM, Revisited. In *27th ACM Symposium on Computational Geometry, SoCG 2011*, pages 1–10. ACM, 2011.

[CMR00] Bruno Courcelle, Johann A. Makowsky, and Udi Rotics. Linear Time Solvable Optimization Problems on Graphs of Bounded Clique-Width. *Theory Comput. Syst.*, 33(2):125–150, 2000.

[CMR01] Bruno Courcelle, Johann A. Makowsky, and Udi Rotics. On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. *Discret. Appl. Math.*, 108(1-2):23–52, 2001.

[CNP+22] Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michał Pilipczuk, Johan M. M. van Rooij, and Jakub Onufry Wojtaszczyk. Solving Connectivity Problems Parameterized by Treewidth in Single Exponential Time. *ACM Trans. Algorithms*, 18(2):17:1–17:31, 2022.

[CO07] Bruno Courcelle and Sang-il Oum. Vertex-minors, monadic second-order logic, and a conjecture by Seese. *J. Comb. Theory, Ser. B*, 97(1):91–126, 2007.

[CO21] Wojciech Czerwiński and Łukasz Orlikowski. Reachability in Vector Addition Systems is Ackermann-complete. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 1229–1240. IEEE, 2021.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[Cou90] Bruno Courcelle. The Monadic Second-Order Logic of graphs. I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.

[Cou95] Bruno Courcelle. The monadic second-order logic of graphs VIII: Orientations. *Ann. Pure Appl. Log.*, 72(2):103–143, 1995.

[Cou06] Bruno Courcelle. The monadic second-order logic of graphs XV: On a conjecture by D. Seese. *J. Appl. Log.*, 4(1):79–114, 2006.

[CPP19] Vincent Cohen-Addad, Michał Pilipczuk, and Marcin Pilipczuk. A polynomial-time approximation scheme for Facility Location on planar graphs. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019*, pages 560–581. IEEE Computer Society, 2019.

[CPST13] Maria Chudnovsky, Irena Penev, Alex Scott, and Nicolas Trotignon. Substitution and χ-boundedness. *Journal of Combinatorial Theory, Series B*, 103(5):567–586, 2013.

[CSTV93] Robert F. Cohen, Sairam Sairam, Roberto Tamassia, and Jeffrey Scott Vitter. Dynamic algorithms for optimization problems in bounded tree-width graphs. In *3rd Conference on Integer Programming and Combinatorial Optimization Conference, IPCO 1993*, pages 99–112. CIACO, 1993.

[CW21] Parinya Chalermsook and Bartosz Walczak. Coloring and Maximum Weight Independent Set of Rectangles. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 860–868. SIAM, 2021.

[Dav22] James Davies. Improved bounds for colouring circle graphs. *Proceedings of the American Mathematical Society*, 150:5121–5135, 07 2022.

[DEM+23] Jan Dreier, Ioannis Eleftheriadis, Nikolas Mählmann, Rose McCarty, Michał Pilipczuk, and Szymon Toruńczyk. First-order model checking on monadically stable graph classes, 2023.

[DF95] Rodney G. Downey and Michael R. Fellows. Fixed-Parameter Tractability and Completeness I: Basic Results. *SIAM J. Comput.*, 24(4):873–921, 1995.

[DF99] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity.* Monographs in Computer Science. Springer, 1999.

[DF13] Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity.* Texts in Computer Science. Springer, 2013.

[DFHT05] Erik D. Demaine, Fedor V. Fomin, Mohammad Taghi Hajiaghayi, and Dimitrios M. Thilikos. Subexponential parameterized algorithms on bounded-genus graphs and H-minor-free graphs. *J. ACM*, 52(6):866–893, 2005.

[DGKS06] Anuj Dawar, Martin Grohe, Stephan Kreutzer, and Nicole Schweikardt. Approximation schemes for first-order definable optimisation problems. In *21th IEEE Symposium on Logic in Computer Science, LICS 2006*, pages 411–420. IEEE Computer Society, 2006.

[DH04] Erik D. Demaine and Mohammad Taghi Hajiaghayi. Equivalence of local treewidth and linear local treewidth and its algorithmic applications. In *15th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004*, pages 840–849. SIAM, 2004.

[Die12] Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics.* Springer, 2012.

[Dir61] Gabriel Andrew Dirac. On rigid circuit graphs. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 25:71–76, 1961.

[DKT14] Zdenek Dvořák, Martin Kupec, and Vojtech Tůma. A dynamic data structure for MSO properties in graphs with bounded tree-depth. In *22th Annual European Symposium on Algorithms, ESA 2014*, volume 8737 of *Lecture Notes in Computer Science*, pages 334–345. Springer, 2014.

[DM88] Alessandro D'Atri and Marina Moscarini. Distance-hereditary graphs, steiner trees, and connected domination. *SIAM Journal on Computing*, 17(3):521–538, 1988.

[DMS23] Jan Dreier, Nikolas Mählmann, and Sebastian Siebertz. First-order model checking on structurally sparse graph classes. In Barna Saha and Rocco A. Servedio, editors, *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, pages 567–580. ACM, 2023.

[DT13] Zdenek Dvořák and Vojtech Tůma. A dynamic data structure for counting subgraphs in sparse graphs. In *13th International Symposium on Algorithms and Data Structures, WADS 2013*, volume 8037 of *Lecture Notes in Computer Science*, pages 304–315. Springer, 2013.

[Dvo18] Zdenek Dvořák. Thin graph classes and polynomial-time approximation schemes. In *29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 1685–1701. SIAM, 2018.

[Dvo20] Zdenek Dvořák. Baker game and polynomial-time approximation schemes. In *31st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2020*, pages 2227–2240. SIAM, 2020.

[Dvo22] Zdenek Dvořák. Approximation metatheorems for classes with bounded expansion. In *18th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2022*, volume 227 of *LIPIcs*, pages 22:1–22:17. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2022.

[Edm65] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[EGIS96] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. Separator based sparsification. I. Planary testing and minimum spanning trees. *J. Comput. Syst. Sci.*, 52(1):3–27, 1996.

[EIT+92] David Eppstein, Giuseppe F. Italiano, Roberto Tamassia, Robert Endre Tarjan, Jeffery R. Westbrook, and Moti Yung. Maintenance of a Minimum Spanning Forest in a Dynamic Plane Graph. *J. Algorithms*, 13(1):33–54, 1992.

[EKM14] David Eisenstat, Philip N. Klein, and Claire Mathieu. Approximating k-center in planar graphs. In *25th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014*, pages 617–627. SIAM, 2014.

[Epp00] David Eppstein. Diameter and treewidth in minor-closed graph families. *Algorithmica*, 27(3):275–291, 2000.

[FG01] Jörg Flum and Martin Grohe. Fixed-Parameter Tractability, Definability, and Model-Checking. *SIAM Journal on Computing*, 31(1):113–145, 2001.

[FG06] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[FK22] Fedor V. Fomin and Tuukka Korhonen. Fast FPT-approximation of branchwidth. In *54th Annual ACM SIGACT Symposium on Theory of Computing, STOC 22*, pages 886–899. ACM, 2022.

[FKS19] Eli Fox-Epstein, Philip N. Klein, and Aaron Schild. Embedding planar graphs into low-treewidth graphs with applications to efficient approximation schemes for metric problems. In *30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*, pages 1069–1088. SIAM, 2019.

[FL81] Aviezri S Fraenkel and David Lichtenstein. Computing a perfect strategy for n × n chess requires time exponential in n. *Journal of Combinatorial Theory, Series A*, 31(2):199–214, 1981.

[FLM+08] Michael R. Fellows, Daniel Lokshtanov, Neeldhara Misra, Frances A. Rosamond, and Saket Saurabh. Graph Layout Problems Parameterized by Vertex Cover. In Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, editors, *Algorithms and Computation, 19th International Symposium, ISAAC 2008, Gold Coast, Australia, December 15-17, 2008. Proceedings*, volume 5369 of *Lecture Notes in Computer Science*, pages 294–305. Springer, 2008.

[FLMT18] Fedor V. Fomin, Mathieu Liedloff, Pedro Montealegre, and Ioan Todinca. Algorithms Parameterized by Vertex Cover and Modular Width, Through Potential Maximal Cliques. *Algorithmica*, 80(4):1146–1169, 2018.

[FLP+18] Fedor V. Fomin, Daniel Lokshtanov, Fahad Panolan, Saket Saurabh, and Meirav Zehavi. Long directed (s,t)-path: Fpt algorithm. *Information Processing Letters*, 140:8–12, 2018.

[FLPS16] Fedor V. Fomin, Daniel Lokshtanov, Fahad Panolan, and Saket Saurabh. Efficient Computation of Representative Families with Applications in Parameterized and Exact Algorithms. *J. ACM*, 63(4):29:1–29:60, 2016.

[FLPS17] Fedor V. Fomin, Daniel Lokshtanov, Fahad Panolan, and Saket Saurabh. Representative Families of Product Families. *ACM Trans. Algorithms*, 13(3):36:1–36:29, 2017.

[FLST20] Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, and Dimitrios M. Thilikos. Bidimensionality and kernels. *SIAM J. Comput.*, 49(6):1397–1422, 2020.

[FLSZ19] Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, and Meirav Zehavi. *Kernelization: Theory of Parameterized Preprocessing.* Cambridge University Press, 2019.

[Fre85] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985.

[Fre98] Greg N. Frederickson. Maintaining regular properties dynamically in k-terminal graphs. *Algorithmica*, 22(3):330–350, 1998.

[FRRS09] Michael R. Fellows, Frances A. Rosamond, Udi Rotics, and Stefan Szeider. Clique-Width is NP-Complete. *SIAM J. Discret. Math.*, 23(2):909–939, 2009.

[FW90] Michael L. Fredman and Dan E. Willard. Blasting through the information theoretic barrier with fusion trees. In Harriet Ortiz, editor, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 1–7. ACM, 1990.

[FY17] Martin Fürer and Huiwen Yu. Space saving by dynamic algebraization based on tree-depth. *Theory Comput. Syst.*, 61(2):283–304, 2017.

[Gas19] William I. Gasarch. Guest Column: The Third P=?NP Poll. *SIGACT News*, 50(1):38–59, mar 2019.

[GH10] Robert Ganian and Petr Hlinený. On parse trees and Myhill-Nerode-type tools for handling graphs of bounded rank-width. *Discret. Appl. Math.*, 158(7):851–867, 2010.

[GHL+15] Jakub Gajarský, Petr Hlinený, Daniel Lokshtanov, Jan Obdrzálek, Sebastian Ordyniak, M. S. Ramanujan, and Saket Saurabh. FO Model Checking on Posets of Bounded Width. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 963–974. IEEE Computer Society, 2015.

[GK09] Martin Grohe and Stephan Kreutzer. Methods for algorithmic meta theorems. In Martin Grohe and Johann A. Makowsky, editors, *Model Theoretic Methods in Finite Combinatorics — AMS-ASL Joint Special Session*, volume 558 of *Contemporary Mathematics*, pages 181–206. American Mathematical Society, 2009.

[GKKT15] David Gibb, Bruce M. Kapron, Valerie King, and Nolan Thorn. Dynamic graph connectivity with improved worst case update time and sublinear space. *CoRR*, abs/1509.06464, 2015.

[GLS84] Martin Grötschel, László Lovász, and Alexander Schrijver. Polynomial Algorithms for Perfect Graphs. In C. Berge and V. Chvátal, editors, *Topics on Perfect Graphs*, volume 88 of *North-Holland Mathematics Studies*, pages 325–356. North-Holland, 1984.

[GM14] Sylvain Guillemot and Dániel Marx. Finding small patterns in permutations in linear time. In Chandra Chekuri, editor, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 82–101. SIAM, 2014.

[GMP+22] Alejandro Grez, Filip Mazowiecki, Michał Pilipczuk, Gabriele Puppis, and Cristian Riveros. Dynamic data structures for timed automata acceptance. *Algorithmica*, 84(11):3223–3245, 2022.

[GPT21] Jakub Gajarský, Michał Pilipczuk, and Szymon Toruńczyk. Stable graphs of bounded twin-width. *CoRR*, abs/2107.03711, 2021.

[Gro03] Martin Grohe. Local tree-width, excluded minors, and approximation algorithms. *Combinatorica*, 23(4):613–632, 2003.

[GRST21] Gramoz Goranci, Harald Räcke, Thatchaphol Saranurak, and Zihan Tan. The expander hierarchy and its applications to dynamic graph algorithms. In *2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*, pages 2212–2228. SIAM, 2021.

[GW20] Maximilian Probst Gutenberg and Christian Wulff-Nilsen. Fully-Dynamic All-Pairs Shortest Paths: Improved Worst-Case Time and Space Bounds. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 2562–2574. SIAM, 2020.

[Gyá87] András Gyárfás. Problems from the world surrounding perfect graphs. *Applicationes Mathematicae*, 19(3-4):413–441, 1987.

[Hal76] Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8:171–186, 1976.

[HdLT01] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, jul 2001.

[HHHK02] Sun-yuan Hsieh, Chin-wen Ho, Tsan-sheng Hsu, and Ming-tat Ko. Efficient algorithms for the hamiltonian problem on distance-hereditary graphs. In Oscar H. Ibarra and Louxin Zhang, editors, *Computing and Combinatorics*, pages 77–86, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[HHK+23] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. Fully Dynamic Connectivity in $O(\log n(\log\log n)^2)$ Amortized Expected Time. *TheoretiCS*, 2, 2023.

[HJ23] Petr Hlinený and Jan Jedelský. Twin-width of planar graphs is at most 8, and at most 6 when bipartite planar. In Kousha Etessami, Uriel Feige, and Gabriele Puppis, editors, *50th International Colloquium on Automata, Languages, and Programming, ICALP 2023, July 10-14, 2023, Paderborn, Germany*, volume 261 of *LIPIcs*, pages 75:1–75:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[HJL+23] Meike Hatzel, Lars Jaffke, Paloma T. Lima, Tomás Masarík, Marcin Pilipczuk, Roohani Sharma, and Manuel Sorge. Fixed-parameter tractability of Directed Multicut with three terminal pairs parameterized by the size of the cutset: twin-width meets flow-augmentation. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 3229–3244. SIAM, 2023.

[HK99] Monika Rauch Henzinger and Valerie King. Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation. *J. ACM*, 46(4):502–516, 1999.

[HO08] Petr Hlinený and Sang-il Oum. Finding Branch-Decompositions and Rank-Decompositions. *SIAM J. Computing*, 38(3):1012–1032, 2008.

[How77] Edward Howorka. A Characterization of Distance-Hereditary Graphs. *The Quarterly Journal of Mathematics*, 28(4):417–420, 12 1977.

[HR20a] Jacob Holm and Eva Rotenberg. Fully-dynamic planarity testing in polylogarithmic time. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 167–180. ACM, 2020.

[HR20b] Jacob Holm and Eva Rotenberg. Worst-Case Polylog Incremental SPQR-trees: Embeddings, Planarity, and Triconnectivity. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 2378–2397. SIAM, 2020.

[HRT18] Jacob Holm, Eva Rotenberg, and Mikkel Thorup. Dynamic Bridge-Finding in $\widetilde{O}(\log^2 n)$ Amortized Time. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 35–52. SIAM, 2018.

[HRW15] Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. Faster fully-dynamic minimum spanning forest. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms - ESA 2015*, pages 742–753, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[HS65] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[HvR23] Jacob Holm, Ivor van der Hoog, and Eva Rotenberg. Worst-case deterministic fully-dynamic biconnectivity in changeable planar embeddings. In Erin W. Chambers and Joachim Gudmundsson, editors, *39th International Symposium on Computational Geometry, SoCG 2023, June 12-15, 2023, Dallas, Texas, USA*, volume 258 of *LIPIcs*, pages 40:1–40:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[Imm99] Neil Immerman. *Descriptive complexity*. Graduate texts in computer science. Springer, 1999.

[IO14] Yoichi Iwata and Keigo Oka. Fast dynamic graph algorithms for parameterized problems. In *14th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2014*, volume 8503 of *Lecture Notes in Computer Science*, pages 241–252. Springer, 2014.

[JKO17] Jisu Jeong, Eun Jung Kim, and Sang-il Oum. The "Art of Trellis Decoding" Is Fixed-Parameter Tractable. *IEEE Trans. Inf. Theory*, 63(11):7178–7205, 2017.

[JKO21] Jisu Jeong, Eun Jung Kim, and Sang-il Oum. Finding Branch-Decompositions of Matroids, Hypergraphs, and More. *SIAM J. Discret. Math.*, 35(4):2544–2617, 2021.

[Joh74] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, 1974.

[JP22] Hugo Jacob and Marcin Pilipczuk. Bounding twin-width for bounded-treewidth graphs, planar graphs, and bipartite graphs. In Michael A. Bekos and Michael Kaufmann, editors, *Graph-Theoretic Concepts in Computer Science - 48th International Workshop, WG 2022, Tübingen, Germany, June 22-24, 2022, Revised Selected Papers*, volume 13453 of *Lecture Notes in Computer Science*, pages 287–299. Springer, 2022.

[Jun78] H.A Jung. On a class of posets and the corresponding comparability graphs. *Journal of Combinatorial Theory, Series B*, 24(2):125–133, 1978.

[Kam18] Shahin Kamali. Compact representation of graphs of small clique-width. *Algorithmica*, 80(7):2106–2131, 2018.

[Kar32] Jovan Karamata. Sur une inégalité rélative aux fonctions convexes. *Publ. Math. Univ. Belgrade*, 1:145–148, 1932.

[Kar72] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[Kin99] Valerie King. Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 81–91. IEEE Computer Society, 1999.

[KKM13] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In Sanjeev Khanna, editor, *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1131–1142. SIAM, 2013.

[KKR12] Ken-ichi Kawarabayashi, Yusuke Kobayashi, and Bruce A. Reed. The disjoint paths problem in quadratic time. *J. Comb. Theory, Ser. B*, 102(2):424–435, 2012.

[KL22] Tuukka Korhonen and Daniel Lokshtanov. An improved parameterized algorithm for treewidth. *CoRR*, abs/2211.07154, 2022. Full version of the STOC 2023 paper.

[KL23] Tuukka Korhonen and Daniel Lokshtanov. An Improved Parameterized Algorithm for Treewidth. In *55th Annual ACM Symposium on Theory of Computing, STOC 2023*, pages 528–541. ACM, 2023.

[KMN+23] Tuukka Korhonen, Konrad Majewski, Wojciech Nadara, Michal Pilipczuk, and Marek Sokołowski. Dynamic treewidth. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 1734–1744. IEEE, 2023.

[KNPS24] Tuukka Korhonen, Wojciech Nadara, Michał Pilipczuk, and Marek Sokołowski. Fully dynamic approximation schemes on planar and apex-minor-free graphs. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 296–313, 2024.

[Kor21] Tuukka Korhonen. A single-exponential time 2-approximation algorithm for treewidth. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021*, pages 184–192. IEEE, 2021.

[Kos84] Alexandr V. Kostochka. Lower bound of the Hadwiger number of graphs by their average degree. *Combinatorica*, 4(4):307–316, 1984.

[KPR93] Philip N. Klein, Serge A. Plotkin, and Satish Rao. Excluded minors, network decomposition, and multicommodity flow. In *25th Annual ACM Symposium on Theory of Computing, STOC 1993*, pages 682–690. ACM, 1993.

[KPS24] Tuukka Korhonen, Michał Pilipczuk, and Giannos Stamoulis. Minor Containment and Disjoint Paths in almost-linear time, 2024.

[Kre12] Stephan Kreutzer. On the parameterized intractability of monadic second-order logic. *Log. Methods Comput. Sci.*, 8(1), 2012.

[KS13] Wojciech Kazana and Luc Segoufin. Enumeration of monadic second-order queries on trees. *ACM Trans. Comput. Log.*, 14(4):25:1–25:12, 2013.

[KS24] Tuukka Korhonen and Marek Sokołowski. Almost-linear time parameterized algorithm for rankwidth via dynamic rankwidth. *CoRR*, abs/2402.12364, 2024. Accepted for presentation at STOC 2024.

[KSB06] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[KT10] Stephan Kreutzer and Siamak Tazari. On Brambles, Grid-Like Minors, and Parameterized Intractability of Monadic Second-Order Logic. In *Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010*, pages 354–364. SIAM, 2010.

[KT16] Yasuaki Kobayashi and Hisao Tamaki. Treedepth Parameterized by Vertex Cover Number. In Jiong Guo and Danny Hermelin, editors, *11th International Symposium on Parameterized and Exact Computation, IPEC 2016, August 24-26, 2016, Aarhus, Denmark*, volume 63 of *LIPIcs*, pages 18:1–18:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[Lag98] Jens Lagergren. Upper bounds on the size of obstructions and intertwines. *J. Comb. Theory, Ser. B*, 73(1):7–40, 1998.

[Lam20] Michael Lampis. Finer Tight Bounds for Coloring on Clique-Width. *SIAM J. Discret. Math.*, 34(3):1538–1558, 2020.

[Lev73] Leonid A. Levin. Universal Sequential Search Problems. *Probl. Peredachi Inf.*, 9:115–116, 1973.

[LP84] Andrea S. Lapaugh and Christos H. Papadimitriou. The even-path problem for graphs and digraphs. *Networks*, 14(4):507–513, 1984.

[LPPS22] Daniel Lokshtanov, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. Fixed-parameter tractability of graph isomorphism in graphs with an excluded minor. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2022, page 914–923, New York, NY, USA, 2022. Association for Computing Machinery.

[Mak04] Johann A. Makowsky. Algorithmic uses of the Feferman-Vaught theorem. *Ann. Pure Appl. Log.*, 126(1-3):159–213, 2004.

[Mao23] Xiao Mao. Fully-Dynamic All-Pairs Shortest Paths: Likely Optimal Worst-Case Update Time. *CoRR*, abs/2306.02662, 2023. Accepted for presentation at STOC 2024.

[MN93] Haiko Müller and Falk Nicolai. Polynomial time algorithms for hamiltonian problems on bipartite distance-hereditary graphs. *Information Processing Letters*, 46(5):225–230, 1993.

[MPS23] Konrad Majewski, Michał Pilipczuk, and Marek Sokołowski. Maintaining CMSO_2 properties on dynamic structures with bounded feedback vertex number. In *40th International Symposium on Theoretical Aspects of Computer Science, STACS 2023*, volume 254 of *LIPIcs*, pages 46:1–46:13. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2023.

[MPZ24] Konrad Majewski, Michał Pilipczuk, and Anna Zych-Pawlewicz. Parameterized dynamic data structure for Split Completion. *CoRR*, abs/2402.08816, 2024.

[MS12] Dániel Marx and Ildikó Schlotter. Obtaining a planar graph by vertex deletion. *Algorithmica*, 62(3-4):807–822, 2012.

[MT92] Jirí Matousek and Robin Thomas. On the complexity of finding iso- and other morphisms for partial k-trees. *Discret. Math.*, 108(1-3):343–364, 1992.

[MT04] Adam Marcus and Gábor Tardos. Excluded permutation matrices and the Stanley–Wilf conjecture. *Journal of Combinatorial Theory, Series A*, 107(1):153–160, 2004.

[NdM08] Jaroslav Nešetřil and Patrice Ossona de Mendez. Grad and classes with bounded expansion I. Decompositions. *European Journal of Combinatorics*, 29(3):760–776, 2008.

[NdMRS21] Jaroslav Nešetřil, Patrice Ossona de Mendez, Roman Rabinovich, and Sebastian Siebertz. Classes of graphs with low complexity: The case of classes with bounded linear rankwidth. *Eur. J. Comb.*, 91:103223, 2021.

[Ned99] Zhivko Prodanov Nedev. Finding an even simple path in a directed planar graph. *SIAM Journal on Computing*, 29(2):685–695, 1999.

[Nie18] Matthias Niewerth. MSO queries on trees: Enumerating answers under updates using forest algebras. In *33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*, pages 769–778. ACM, 2018.

[NPSW23] Jesper Nederlof, Michal Pilipczuk, Céline M. F. Swennenhuis, and Karol Wegrzycki. Hamiltonian Cycle Parameterized by Treedepth in Single Exponential Time and Polynomial Space. *SIAM J. Discret. Math.*, 37(3):1566–1586, 2023.

[NS17] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, Las Vegas, and $O(n^{1/2-\varepsilon})$-time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, page 1122–1129, New York, NY, USA, 2017. Association for Computing Machinery.

[NSW17] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In Chris Umans, editor, *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 950–961. IEEE Computer Society, 2017.

[OPR+23] Jędrzej Olkowski, Michał Pilipczuk, Mateusz Rychlicki, Karol Węgrzycki, and Anna Zych-Pawlewicz. Dynamic data structures for parameterized string problems. In *40th International Symposium on Theoretical Aspects of Computer Science, STACS 2023*, volume 254 of *LIPIcs*, pages 50:1–50:22. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2023.

[OS06] Sang-il Oum and Paul D. Seymour. Approximating clique-width and branch-width. *J. Comb. Theory, Ser. B*, 96(4):514–528, 2006.

[OS07] Sang-il Oum and Paul D. Seymour. Testing branch-width. *J. Combinatorial Theory Ser. B*, 97(3):385–393, 2007.

[Oum05] Sang-il Oum. Rank-width and vertex-minors. *J. Comb. Theory, Ser. B*, 95(1):79–100, 2005.

[Oum08a] Sang-il Oum. Approximating rank-width and clique-width quickly. *ACM Trans. Algorithms*, 5(1):10:1–10:20, 2008.

[Oum08b] Sang-il Oum. Rank-width is less than or equal to branch-width. *J. Graph Theory*, 57(3):239–244, 2008.

[Oum17] Sang-il Oum. Rank-width: Algorithmic and structural results. *Discret. Appl. Math.*, 231:15–24, 2017.

[Par78] T. D. Parsons. Pursuit-evasion in a graph. In Yousef Alavi and Don R. Lick, editors, *Theory and Applications of Graphs*, pages 426–441, Berlin, Heidelberg, 1978. Springer Berlin Heidelberg.

[PD06] Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006.

[Pil20] Michał Pilipczuk. Computing tree decompositions. In Fedor V. Fomin, Stefan Kratsch, and Erik Jan van Leeuwen, editors, *Treewidth, Kernels, and Algorithms — Essays Dedicated to Hans L. Bodlaender on the Occasion of His 60th Birthday*, volume 12160 of *Lecture Notes in Computer Science*, pages 189–213. Springer, 2020.

[PP20] Adam Paszke and Michał Pilipczuk. VC Density of Set Systems Definable in Tree-Like Graphs. In Javier Esparza and Daniel Král', editors, *45th International Symposium on Mathematical Foundations of Computer Science, MFCS 2020, August 24-28, 2020, Prague, Czech Republic*, volume 170 of *LIPIcs*, pages 78:1–78:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[PS23] Michał Pilipczuk and Marek Sokołowski. Graphs of bounded twin-width are quasi-polynomially χ-bounded. *J. Comb. Theory, Ser. B*, 161:382–406, 2023.

[PSS19] Richard Peng, Bryce Sandlund, and Daniel Dominic Sleator. Optimal Offline Dynamic 2, 3-Edge/Vertex Connectivity. In Zachary Friggstad, Jörg-Rüdiger Sack, and Mohammad R. Salavatipour, editors, *Algorithms and Data Structures - 16th International Symposium, WADS 2019, Edmonton, AB, Canada, August 5-7, 2019, Proceedings*, volume 11646 of *Lecture Notes in Computer Science*, pages 553–565. Springer, 2019.

[PSZ22] Michał Pilipczuk, Marek Sokołowski, and Anna Zych-Pawlewicz. Compact Representation for Matrices of Bounded Twin-Width. In Petra Berenbrink and Benjamin Monmege, editors, *39th International Symposium on Theoretical Aspects of Computer Science, STACS 2022, March 15-18, 2022, Marseille, France (Virtual Conference)*, volume 219 of *LIPIcs*, pages 52:1–52:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[PT06] Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *38th Annual ACM Symposium on Theory of Computing, STOC 2006*, pages 232–240. ACM, 2006.

[PW18] Michal Pilipczuk and Marcin Wrochna. On Space Efficiency of Algorithms Working on Structural Decompositions of Graphs. *ACM Trans. Comput. Theory*, 9(4):18:1–18:36, 2018.

[Rin65] Gerhard Ringel. Das Geschlecht des vollständigen paaren Graphen. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 28:139–150, 1965.

[Rob83] John Robson. The complexity of go. volume 9, pages 413–417, 01 1983.

[RS84] Neil Robertson and Paul D. Seymour. Graph Minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.

[RS86a] Neil Robertson and Paul D. Seymour. Graph Minors. II. Algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.

[RS86b] Neil Robertson and Paul D. Seymour. Graph Minors. V. Excluding a planar graph. *J. Comb. Theory, Ser. B*, 41(1):92–114, 1986.

[RS95] Neil Robertson and Paul D. Seymour. Graph Minors. XIII. The Disjoint Paths Problem. *J. Comb. Theory, Ser. B*, 63(1):65–110, 1995.

[RVS19] Felix Reidl, Fernando Sánchez Villaamil, and Konstantinos Stavropoulos. Characterising bounded expansion by neighbourhood complexity. *European Journal of Combinatorics*, 75:152–168, 2019.

[SE81] Yossi Shiloach and Shimon Even. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, jan 1981.

[See91] Detlef Seese. The Structure of Models of Decidable Monadic Theories of Graphs. *Ann. Pure Appl. Log.*, 53(2):169–195, 1991.

[Sei74] D Seinsche. On a property of the class of n-colorable graphs. *Journal of Combinatorial Theory, Series B*, 16(2):191–193, 1974.

[SS20] Alex Scott and Paul D. Seymour. A survey of χ-boundedness. *J. Graph Theory*, 95(3):473–504, 2020.

[ST83] Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, apr 1975.

[Tho90] Robin Thomas. A Menger-like Property of Tree-Width: The Finite Case. *J. Comb. Theory, Ser. B*, 48(1):67–76, 1990.

[Tho04] Mikkel Thorup. Fully-Dynamic All-Pairs Shortest Paths: Faster and Allowing Negative Cycles. In Torben Hagerup and Jyrki Katajainen, editors, *Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004, Proceedings*, volume 3111 of *Lecture Notes in Computer Science*, pages 384–396. Springer, 2004.

[TP93] Jan Arne Telle and Andrzej Proskurowski. Practical Algorithms on Partial k-Trees with an Application to Domination-like Problems. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, Nicola Santoro, and Sue Whitesides, editors, *Algorithms and Data Structures, Third Workshop, WADS '93, Montréal, Canada, August 11-13, 1993, Proceedings*, volume 709 of *Lecture Notes in Computer Science*, pages 610–621. Springer, 1993.

[Tre01] Luca Trevisan. Non-approximability results for optimization problems on bounded degree instances. In Jeffrey Scott Vitter, Paul G. Spirakis, and Mihalis Yannakakis, editors, *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pages 453–461. ACM, 2001.

[vRBR09] Johan M. M. van Rooij, Hans L. Bodlaender, and Peter Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In Amos Fiat and Peter Sanders, editors, *Algorithms - ESA 2009, 17th Annual European Symposium, Copenhagen, Denmark, September 7-9, 2009. Proceedings*, volume 5757 of *Lecture Notes in Computer Science*, pages 566–577. Springer, 2009.

[WAPL14] Yu Wu, Per Austrin, Toniann Pitassi, and David Liu. Inapproximability of treewidth and related problems. *J. Artif. Intell. Res.*, 49:569–600, 2014.

[WS11] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.

[Wul17] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1130–1143. ACM, 2017.