University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Jakub Łącki

Dynamic Graph Algorithms for Connectivity Problems

PhD dissertation

Supervisor

dr hab. Piotr Sankowski prof. UW

Institute of Informatics
University of Warsaw

January 2015

Author's declaration:
aware of legal responsibility I hereby declare that I have written this dissertation myself and all the contents of the dissertation have been obtained by legal means.

January 22, 2015 .....
date *Jakub Łącki*

Supervisor's declaration:
the dissertation is ready to be reviewed

January 22, 2015 .....
date *dr hab. Piotr Sankowski prof. UW*

Abstract

In this thesis we present several new algorithms for dynamic graph problems. The common theme of the problems we consider is connectivity. In particular, we study the maintenance of connected components in a dynamic graph, and the Steiner tree problem over a dynamic set of terminals.

First, we present an algorithm for decremental connectivity in planar graphs. It processes any sequence of edge deletions intermixed with a set of connectivity queries. Each connectivity query asks whether two given vertices belong to the same connected component. The running time of this algorithm is optimal, that is, it handles any sequence of updates in linear time, and answers queries in constant time. This improves over the best previously known algorithm, whose total update time is $O(n \log n)$.

Then, we study the dynamic Steiner tree problem. In this problem, given a weighted graph G on a vertex set V and a dynamic set $S \subseteq V$ of terminals, subject to insertions and deletions, the goal is to maintain a constant-approximate Steiner tree spanning S in G. For general graphs and every integer $k \geqslant 2$, we show an $(8k-4)$-approximate algorithm, which processes updates in $O(kn^{1/k} \log^4 n)$ amortized expected time. In the case of planar graphs we show a different solution, whose amortized update time is $O(\varepsilon^{-1} \log^6 n)$ and the approximation ratio is $4 + \varepsilon$.

Finally, we study graph connectivity in a semi-offline model. We consider a problem, in which the input is a sequence of graphs $G_1, \ldots, G_t$, such that G_{i+1} is obtained from G_i by adding or removing a single edge. In the beginning, this sequence is given to the algorithm for preprocessing. After that, the algorithm should efficiently answer queries of one of two kinds. A `forall`(a, b, u, w) query, where $1 \leqslant a \leqslant b \leqslant t$ and u, w are vertices, asks whether u and w are connected with a path in *each* of $G_a, G_b, \ldots, G_b$. Similarly, an `exists`(a, b, u, w) query asks if the given vertices are connected in *any* of $G_a, G_b, \ldots, G_b$. For `forall` queries, we show an algorithm that after preprocessing in $O(t \log t(\log n + \log \log t))$ expected time answers queries in $O(\log n \log \log t)$ time. In the case of `exists` queries, the preprocessing time is $O(m + nt)$ and the query time is constant.

Key words: dynamic graph algorithms, dynamic connectivity, decremental connectivity, Steiner tree

AMS Classification: 05C85 Graph algorithms, 68P05 Data structures, 68Q25 Analysis of algorithms and problem complexity, 68W40 Analysis of algorithms

Streszczenie

W niniejszej pracy przedstawiamy nowe algorytmy dla dynamicznych problemów grafowych. Wszystkie omawiane problemy dotyczą zagadnienia spójności. W szczególności zajmujemy się utrzymywaniem spójnych składowych w zmieniającym się grafie oraz utrzymywaniem drzewa Steinera rozpinającego zmieniający się zbiór terminali.

Po pierwsze pokazujemy algorytm dla dekrementalnej spójności w grafach planarnych. Algorytm ten przetwarza ciąg operacji składający się z usunięć krawędzi oraz zapytań o spójność. Każde zapytanie sprawdza, czy dwa podane wierzchołki należą do tej samej spójnej składowej. Czas działania tego algorytmu jest optymalny: przetwarza on dowolny ciąg aktualizacji w czasie liniowym i odpowiada na zapytania w czasie stałym. Poprawia to wcześniejszy algorytm, którego łączny czas aktualizacji to $O(n \log n)$.

Następnie prezentujemy algorytm, który dla ważonego grafu G na zbiorze wierzchołków V i dla zmieniającego się zbioru terminali $S \subseteq V$ (wierzchołki są do niego dodawane i z niego usuwane) utrzymuje stałą aproksymację drzewa Steinera rozpinającego zbiór S w G. Dla grafów dowolnych i każdego $k \geqslant 2$ pokazujemy algorytm $(8k-4)$-aproksymacyjny, który przetwarza aktualizacje w oczekiwanym czasie zamortyzowanym $O(kn^{1/k} \log^4 n)$. W przypadku grafów planarnych pokazujemy szybszy algorytm o zamortyzowanym czasie aktualizacji $O(\varepsilon^{-1} \log^6 n)$ i współczynniku aproksymacji $4+\varepsilon$.

Ponadto badamy spójność grafów w modelu częściowo *offline*. Rozważamy problem, w którym wejściem jest ciąg grafów $G_1, \ldots, G_t$, taki że G_{i+1} otrzymuje się z G_i przez dodanie lub usunięcie jednej krawędzi. Algorytm poznaje ten ciąg na początku swego działania i może wykonać preprocessing. Następnie powinien efektywnie odpowiadać na nadchodzące zapytania jednego z dwóch rodzajów. Zapytanie `forall`(a, b, u, w), gdzie $1 \leqslant a \leqslant b \leqslant t$, a u, w są wierzchołkami, sprawdza, czy u i w są połączone ścieżką w *każdym* z grafów $G_a, G_b, \ldots, G_b$. Podobnie zapytanie `exists`(a, b, u, w) sprawdza, czy podane wierzchołki są połączone w *którymkolwiek* z grafów $G_a, G_b, \ldots, G_b$. Dla zapytań `forall` pokazujemy algorytm, który po preprocessingu w oczekiwanym czasie $O(t \log t(\log n + \log\log t))$ odpowiada na zapytania w czasie $O(\log n \log\log t)$. W przypadku zapytań `exists` czas preprocessingu to $O(m+nt)$, zaś czas zapytań jest stały.

Słowa kluczowe: dynamiczne algorytmy grafowe, dynamiczna spójność, dekrementalna spójność, drzewo Steinera

Klasyfikacja tematyczna AMS: 05C85 Graph algorithms, 68P05 Data structures, 68Q25 Analysis of algorithms and problem complexity, 68W40 Analysis of algorithms

Judytce

Contents

Chapter 1

Introduction

What makes us think that an algorithm is efficient? In the theoretical approach, we consider an algorithm efficient if the number of instructions it executes is linear in the size of its input data. This is justified by the fact that usually the algorithm has to read the entire input, or at least a big fraction of it. In fact, once we show that a linear time algorithm has to access a constant fraction of its input data, we may infer that it is optimal.

At the same time, the theoretical optimality of an algorithm may not mean much in real life applications. If we implement a linear time algorithm and run it on data, whose size is measured in terabytes, it may take long to complete. While the running time of a single run may be reasonable, if the algorithm is used repeatedly, our requirements for its running time may be much stricter.

However, it is a common scenario that if the data is being processed repeatedly, between two consecutive computations it only changes slightly. This fact can be exploited to obtain faster algorithms, which, instead of processing the data from scratch after each change, would only process the *changes* that are made. Such algorithms are called *dynamic* and are the main topic of this thesis.

We focus on dynamic graph algorithms, that is dynamic algorithms that can be used to process graphs. A graph algorithm is dynamic if it maintains information about a graph subject to its modifications. Typically the modifications alter the set of vertices or edges in the graph. The sequence of modifications, henceforth called *updates*, is intermixed with a set of *queries*. There are various dynamic graph problems, which differ in the allowed types of queries. For example, the queries may ask about the existence of a path between two given vertices or the weight of the minimum spanning tree of the graph. The algorithm should utilize the maintained information to answer each query faster than in the time needed to recompute the solution

from scratch.

There are three kinds of dynamic graph problems, which differ in the types of updates that can happen. Let us focus on the problems, where the updates alter the set of edges. In an *incremental* problem, edges may only be added, whereas in a *decremental* one, edges can only be deleted. A *fully dynamic* problem is more general than the previous two, as both edge insertions and deletions can take place.

The central problem in the area of dynamic graph algorithm is the dynamic connectivity problem. In this problem, we are given a graph subject to edge insertions and deletions, and our goal is to answer queries about the existence of a path connecting two given vertices. Since connectivity is a fundamental graph property, the study of dynamic connectivity is important both from theoretical and practical points of view.

The dynamic connectivity problem is also a benchmark of the known techniques for tackling dynamic graph problems. While it has a simple formulation and has received much attention, the first algorithm with polylogarithmic update time was given in 1995 [20], and it has taken over 15 more years to develop the first solution with polylogarithmic worst-case update time [26]. However, many questions regarding dynamic connectivity have not been answered yet. The running time of the best algorithm of fully dynamic connectivity is still higher than the lower bound. Similarly, for decremental connectivity in general graphs, no lower bound is known.

Only in few special cases we know that the existing solutions are optimal. In particular, there exists an incremental connectivity algorithm [40], whose running time matches an existing lower bound [16]. In the case of trees there exists a decremental connectivity algorithm that processes all updates in linear time, and answers each query in constant time [3]. Also, for plane graphs[1] there exists a fully dynamic algorithm with amortized update time of $O(\log n)$, which matches the lower bound.

In this thesis we finally settle the connectivity problem in one more setting. We show a decremental connectivity algorithm for planar graphs, which processes updates in linear total time and answers queries in constant time. This improves over an existing algorithm, whose total update time is $O(n \log n)$ and matches the time bound of decremental connectivity in trees.

Moreover, we introduce and solve a new dynamic problem that deals with connectivity in a semi-offline model. We develop algorithms that process a *graph timeline*, that is a sequence of graphs $G_1, \ldots, G_t$, such that G_{i+1} is obtained from G_i by adding or removing a single edge. In this model, an

[1] A graph is *plane* if it is planar, and its embedding remains fixed in the course of the operations.

algorithm may preprocess the entire timeline at the beginning, and after that it should answer queries arriving in online fashion. We consider timelines of undirected graphs and two types of queries.

An `exists`(u, w, a, b) query, where u and w are vertices and $1 \leqslant a \leqslant b \leqslant t$, asks whether vertices u and w are connected in *any* of $G_a, G_{a+1}, \ldots, G_b$. On the other hand, a `forall`(u, w, a, b) query asks whether vertices u and w are connected in *each* of $G_a, G_{a+1}, \ldots, G_b$.

This thesis also deals with another dynamic graph problem, namely dynamic Steiner tree. Let $G = (V, E, d_G)$ be a weighted graph and $S \subseteq V$ be a set of *terminals*. The Steiner tree spanning S in G is a minimum-weight subgraph of G, in which every pair of vertices of S is connected. In the dynamic Steiner tree problem, the set S is *dynamic*, that is, its elements are inserted and deleted. The goal is to maintain a constant-approximate Steiner tree spanning S in G.

The dynamic variant of the Steiner tree problem was introduced by Imase and Waxman [25] in 1991, and while it has been studied since then, no solution with sublinear update time has been given. The existing algorithms for dynamic Steiner tree focus on minimizing the number of changes to the tree, or use a heuristic approach to minimize the running time.

We show the first solution for dynamic Steiner tree with sublinear update time. For any $k \geqslant 2$ it processes updates in $O(kn^{1/k} \log^4 n)$ expected amortized algorithm, and maintains a $(8k-4)$-approximate Steiner tree. Moreover, for planar graphs we show a $(4+\varepsilon)$-approximate algorithm, which processes each update in $\tilde{O}(\varepsilon^{-1} \log^6 n)$ amortized time.

The Steiner tree problem is closely related to the minimum spanning tree (MST) problem. First, the MST problem is a special case of the Steiner tree problem, where every vertex of the graph is a terminal. More importantly, there also exists a reverse relation. We may use an algorithm for computing MST to compute an approximate Steiner tree. This is achieved by building a complete graph on the set of terminals, where the edge weight of an edge uw is the distance between u and w in the original graph. It is well-known that the MST of this complete graph corresponds to a 2-approximate Steiner tree.

The Steiner tree problem and MST are also related in the dynamic setting. The 2-approximate algorithm can be made dynamic, using an algorithm for dynamic MST, which gives an algorithm for dynamic Steiner tree with an update time of $\tilde{O}(n)$. The sublinear time algorithm for dynamic Steiner tree, that we give in this thesis, also uses dynamic MST as a subroutine.

In the following part of this chapter, we describe our results and their relation to the previously obtained algorithm in the area.

Update time	Query time	Type	Authors
$O(\log n(\log\log n)^3)$	$O(\log n/\log\log n)$	Monte Carlo, amrt.	Thorup [42]
$O(\log^2 n/\log\log n)$	$O(\log n/\log\log n)$	deterministic, amrt.	Wulff-Nilsen [47]
$O(\log^5 n)$	$O(\log n/\log\log n)$	Monte Carlo, w-c	Kapron et al. [26]
$O(\sqrt{n})$	$O(1)$	deterministic, w-c	Eppstein et al. [12]

Figure 1.1: Algorithms for fully dynamic connectivity in general graphs. W-c stands for worst-case, whereas amrt. means amortized.

1.1 Dynamic Connectivity

In the dynamic connectivity problem, given a graph subject to edge updates, the goal is to answer queries about the existence of a path connecting two vertices. We first review the previously obtained algorithms for this problem, both for general graphs and some restricted graph classes. Then, we show our results in this area and present their relation to the existing results.

1.1.1 General Graphs

Fully Dynamic Connectivity

There has been a long line of research considering the fully dynamic connectivity in general graphs [15, 12, 20, 23, 42, 26, 47]. The study of this problem was initiated by Frederickson [15] about 30 years ago, but the first polylogarithmic time algorithm has been given over 10 years later [20]. The first algorithm with polylogarithmic worst-case update time was shown in 2013 by Kapron and King [26], but the algorithm is randomized. A deterministic algorithm with polylogarithmic worst-case update time is not known, and obtaining such an algorithm is a major open problem. The best currently known algorithms for fully dynamic connectivity are summarized in Figure 1.1.

Concerning lower bounds, Henzinger and Fredman [21] obtained a lower bound of $\Omega(\log n/\log\log n)$ in the RAM model. This was improved by Demaine and Pătraşcu [38] to a lower bound of $\Omega(\log n)$ in cell-probe model. Both these lower bounds hold also for plane graphs.

Incremental Connectivity

Incremental graph connectivity can be solved using an algorithm for the union-find problem. It follows from the result of Tarjan [40] that a sequence

of t edge insertions and t queries can be handled in $O(t\alpha(t))$ time, where $\alpha(t)$ is the extremely slowly growing inverse Ackermann function. A matching lower bound ($\Omega(\alpha(n))$ time per operation) has been shown by Fredman and Saks [16] in the cell probe model.

Decremental Connectivity

For the decremental variant, Thorup [41] has shown a randomized algorithm, which processes any sequence of edge deletions in $O(m\log(n^2/m) + n(\log n)^3(\log\log n)^2)$ time and answers queries in constant time. Here, m is the initial number of edges in the graph. If $m = \Theta(n^2)$, the update time is $O(m)$, whereas for $m = \Omega(n(\log n \log\log n)^2)$ it is $O(m\log n)$.

1.1.2 Planar Graphs and Trees

The situation is much simpler in the case of planar graphs. Eppstein et. al [14] gave a fully dynamic algorithm, which handles updates and queries in $O(\log n)$ amortized time, but it works only for *plane* graphs, that is, it requires that the graph embedding remains fixed. For the general case (i.e., when the embedding may change) Eppstein et. al [13] gave an algorithm with $O(\log^2 n)$ worst-case update time and $O(\log n)$ query time.

In planar graphs, the best known solution for the incremental connectivity problem is the union-find algorithm. On the other hand, for the decremental problem nothing better than a direct application of the fully dynamic algorithm is known. This is different from both general graphs and trees, where the decremental connectivity problems have better solutions than what could be achieved by a simple application of their fully dynamic counterparts. In the case of general graphs, the best total update time is $O(m\log n)$ [41] (except for very sparse graphs, including planar graphs), compared to $O(m\log n(\log\log n)^3)$ time for the fully dynamic variant. For trees, only $O(n)$ time is necessary to perform all updates in the decremental scenario [3], while in the fully dynamic case one can use dynamic trees that may handle each update in $O(\log n)$ worst-case time.

1.1.3 Dynamic MST

In the dynamic MST problem, the input is a weighted undirected graph $G = (V, E, d_G)$, subject to edge insertions and removals. The goal is to maintain the weight of the MST of G, as the set of edges is modified. The only efficiency parameter is the time needed to process a single update.

Update time	Type	Authors
$O(\sqrt{m})$	worst-case	Frederickson [15]
$O(\sqrt{n})$	worst-case	Eppstein [12]
$O(\sqrt[3]{n}\log n)$	amortized	Henzinger, King [22]
$O(\log^4 n)$	amortized	Holm et al. [23]

Figure 1.2: The history of algorithms for dynamic MST. All the algorithms listed here are deterministic.

This problem is closely related with the dynamic connectivity problem, and some techniques are common for both these problems. While the algorithms for dynamic connectivity maintain a spanning tree of a graph, the algorithms for dynamic MST maintain a *minimum* spanning tree. In some cases, new techniques for dynamic connectivity also implied better algorithms for dynamic MST [15, 23].

The algorithms for dynamic MST are listed in Figure 1.2. The fastest known algorithm processes updates in $O(\log^4 n)$ amortized time [23]. Contrary to dynamic connectivity, no algorithm with polylogarithmic worst-case update time is known. In fact even finding an algorithm with $o(\sqrt{n})$ worst-case update time is an open problem [26]. In addition to that, even though the best dynamic algorithms for connectivity are randomized, this is not the case for dynamic MST.

Dynamic MST has also been considered in the offline model. In this model, the input is a sequence of weighted graphs $G_1, \ldots, G_t$, such that G_{i+1} is obtained from G_i by changing the weight of a single edge. Eppstein [11] has shown an algorithm, which computes the weight of the MST of every G_i in $O(t \log n)$ total time. We use the techniques developed by Eppstein in our algorithms for answering `exists` and `forall` queries.

1.1.4 Our Results

Decremental Connectivity in Planar Graphs

We show an algorithm for the decremental connectivity problem in planar graphs, which processes any sequence of edge deletions in $O(n)$ time and answers queries in constant time. This improves over the previous bound of $O(n \log n)$, which can be obtained by applying the fully dynamic algorithm by Eppstein [14], and matches the running time of decremental connectivity

on trees [3].

In fact, we present a $O(n)$ time reduction from the decremental connectivity problem to a collection of *incremental* problems in graphs of total size $O(n)$. These incremental problems have a specific structure: the set of allowed union operations forms a planar graph and is given in advance. As shown by Gustedt [19], such a problem can be solved in linear time.

Our result shows that in terms of total update time, the decremental connectivity problem in planar graphs is definitely not harder than the incremental one. Though, it should be noted that the union-find algorithm can process any sequence of k query or update operations in $O(k\alpha(n))$ time, while in our algorithm we are only able to bound the time to process any sequence of edge deletions.

Moreover, since fully dynamic connectivity has a lower bound of $\Omega(\log n)$ (even in plane graphs) shown by Demaine and Pătraşcu [38], our results imply that in planar graphs decremental connectivity is strictly easier than the fully dynamic one. We suspect that the same holds for general graphs, and we conjecture that it is possible to break the $\Omega(\log n)$ bound for a single operation of a decremental connectivity algorithm, or the $\Omega(m \log n)$ bound for processing a sequence of m edge deletions.

Our algorithm, unlike the majority of algorithms for maintaining connectivity, does not maintain the spanning tree of the current graph. As a result, it does not have to search for a replacement edge when an edge from the spanning tree is deleted. Our approach is based on a novel and very simple approach for detecting bridges, which alone gives $O(n \log n)$ total update time. We use the fact that a deletion of edge uw in the graph causes some connected component to split if both sides of uw belong to the same face. This condition can in turn be verified by solving an incremental connectivity problem in the dual graph. When we detect a deletion that splits a connected component, we start two parallel DFS searches from u and w to identify the *smaller* of the two new components. Once the first search finishes, the other one is stopped. A simple argument shows that this algorithm runs in $O(n \log n)$ time.

We then show that the DFS searches can be speeded up using an r-division, that is a decomposition of a planar graph into subgraphs of size at most $r = \log^2 n$. This gives an algorithm running in $O(n \log \log n)$ time. For further illustration of this idea we show how to apply it twice in order to obtain an $O(n \log \log \log n)$ time algorithm. Then, we observe that the $O(n \log \log \log n)$ time algorithm reduces the problem of maintaining connectivity in the input graph to maintaining connectivity in a number of graphs of size at most $O(\log^2 \log n)$. The number of all graphs on so few vertices is so small that we can simply precompute the answers for all of them and use

these precomputed answers to obtain the linear-time algorithm. The preprocessing of all graphs of bounded size is again an idea that, to the best of our knowledge, has never been previously used for designing dynamic graph algorithms.

Connectivity in Graph Timelines

We also study graph connectivity in a semi-offline model. We develop algorithms that process a *graph timeline*, that is a sequence of graphs $G_1, \ldots, G_t$, such that G_{i+1} is obtained from G_i by adding or removing a single edge. In this model, an algorithm may preprocess the entire timeline at the beginning, and after that it should answer queries arriving in online fashion. We consider timelines of undirected graphs and two types of queries.

An `exists`(u, w, a, b) query, where u and w are vertices and $1 \leqslant a \leqslant b \leqslant t$, asks whether vertices u and v are connected in *any* of $G_a, G_{a+1}, \ldots, G_b$. We show an algorithm that after preprocessing in $O(m + nt)$ time may answer such queries in $O(1)$ time (assuming $t = O(n^c)$). Moreover, it may compute all indices of the graphs, in which u and w are connected, returning them one by one with constant delay.

We also consider `forall`(u, w, a, b) query, which asks whether vertices u and w are connected in *each* of $G_a, G_{a+1}, \ldots, G_b$. For this problem, we show an algorithm whose expected preprocessing time is $O(m+t \log t(\log n+ \log\log t))$ (m denotes the number of edges in G_1) and the query time is $O(\log n \log\log t)$. The algorithm is randomized and answers queries correctly with high probability.

The algorithms for both types of queries are based on a segment tree over the entire sequence $G_1, \ldots, G_t$. We call this tree a connectivity history tree (CHT). Assume that t is a power of 2. Then, the CHT can be computed recursively as follows. The parameter of the recursion is a fragment of the sequence $G_1, \ldots, G_t$, which can be represented as a discrete interval. For an interval $[a, b]$ (we begin with an interval $[1, t]$) we consider a graph $G_{[a,b]}$ obtained by keeping only the edges that are present in every graph among $G_a, \ldots, G_b$ and compute its connected components. Then, if $a < b$, we recurse on the first and second halves of the interval $[a, b]$. We say that every interval, which is at some point the parameter of the recursion, is an *elementary interval*. It is a well-known fact that the number of elementary intervals is $O(t)$ and every interval $[a, b]$, where $1 \leqslant a \leqslant b \leqslant t$ can be partitioned into $O(\log t)$ elementary intervals.

In the case of `exists` queries, for every elementary interval $[a, b]$ we precompute the answer to every possible `exists`(u, w, a, b) query. We make some observations that allow us to precompute the answers in only $O(m+nt)$ time

(instead of $O(n^2 t)$). Using this information, we could answer an arbitrary query by partitioning the query interval into $O(\log t)$ elementary intervals. However, we show a more involved query algorithm, which answers queries in constant time. Our ideas for speeding up the preprocessing phase follow the techniques used by Eppstein [11].

The algorithm for answering `forall` queries is more involved. For every vertex v we define a sequence $C_v = c_v^1, \ldots, c_v^t$, where c_v^i is the identifier of the connected component of v in G_i. In order to answer a query `forall`(u, w, a, b), we compute and compare the hashes of sequences $c_u^a, \ldots, c_u^b$ and $c_w^a, \ldots, c_w^b$. The computation of hashes requires an initial preprocessing. We use the connectivity history tree to compute connectivity information about every graph $G_1, \ldots, G_t$ in near linear time. Using this information we precompute hashes of some prefixes of C_v, which are then used to efficiently compute the desired hashes. It should be noted that our results in this area have been recently speeded up and simplified by Karczmarz [27].

1.2 Dynamic Steiner Tree

The next dynamic graph problem that we consider is the dynamic Steiner tree problem. The static variant of the Steiner tree problem is NP-complete, and, unless $P = NP$, does not admit a PTAS, even in complete graphs with edge weights restricted to 1 and 2. In general graphs, only a 1.39-approximate algorithm is known [8]. On the other hand, the problem admits a PTAS in geometric graphs, i.e., when the edge weights are the Euclidean distances between the points in finite dimensional geometric space [4, 37] and in planar graphs [7]. The PTAS for planar graphs is asymptotically very efficient, i.e., we can construct an $(1 + \varepsilon)$-approximate Steiner tree in $O(n \log n)$ time.

The dynamic Steiner tree problem was first introduced in the pioneering paper by Imase and Waxman [25] and its study was later continued in [34, 17, 18]. However, all these papers focus on minimizing the number of changes to the tree that are necessary to maintain a good approximation, and ignore the problem of efficiently finding these changes. The efficiency of these online algorithms is measured in terms of the number of *replacements* that are performed after every terminal insertion or deletion. The algorithms for dynamic Steiner tree usually represent the Steiner tree as a set of shortest paths between pairs of vertices, and a replacement is every change made to this set of paths.

The original algorithm of Imase and Waxman [25] made $O(n^{3/2})$ replacements during the processing of a sequence of n update operations. This was improved in the incremental case to $O(\log n)$ per terminal insertion by Megow

et al. [34]. Later, Gu, Gupta and Kumar [17, 18] have shown that after every update only $O(1)$ replacements are needed in amortized sense. Moreover, they showed that if terminals are only deleted, it is possible to maintain a constant approximate Steiner tree making only a single change after every terminal deletion.

The problem of maintaining the Steiner tree is also an important problem in the network community [9], and while it has been studied for many years, the research resulted only in several heuristic approaches [5, 1, 24, 39] none of which has been formally proven to have sublinear running time.

1.2.1 Our Results

We show the first sublinear time algorithm for the dynamic Steiner tree problem. For general graphs and any $k \geqslant 2$, we give a $O(kn^{1/k} \log^4 n)$ time algorithm, which maintains a $(8k-4)$-approximate Steiner tree. The time bound is expected and amortized. Moreover, we show a $(4+\varepsilon)$-approximate algorithm for planar graphs, which processes updates in $\tilde{O}(\varepsilon^{-1} \log^6 n)$ amortized time.

To the best of our knowledge, previously only a simple $\tilde{O}(n)$ time algorithm was known. This algorithm first computes the metric closure $\overline{G}$ of the graph G, and then maintains the MST of $\overline{G}[S]$ using a polylogarithmic dynamic MSF (minimum spanning forest) algorithm [23]. It is a well-known fact that this yields a 2-approximate Steiner tree. In order to update $\overline{G}[S]$ we need to insert and remove terminals together with their incident edges, what requires $\Theta(n)$ calls to the dynamic MSF structure. However, such a linear bound is far from being satisfactory, as it does not lead to any improvement in the running time for sparse networks, where $m = O(n)$.[2] In such networks after each update we can actually compute a 2-approximate Steiner tree in $O(n \log n)$ time from scratch [35].

Our algorithm for dynamic Steiner tree uses an auxiliary graph called a *bipartite emulator*. It is a low-degree bipartite graph, which can be used to approximate distances in the original graph. Roughly speaking, in our algorithm we maintain a subgraph H of the bipartite emulator, which changes with every change to the set of terminals. We show that the MSF of H approximates the Steiner tree spanning the set of terminals in the original graph. To obtain the algorithm for dynamic Steiner tree, we run dynamic MSF algorithm on the graph H.

We construct different bipartite emulators for general and planar graphs, which results in different running times. While our emulators are constructed

[2]It is widely observed that most real-world networks are sparse [10].

using previously known distance oracles [44, 43], our contribution lies in the introduction of the concept of bipartite emulators, whose properties make it possible to solve the dynamic Steiner tree problem in sublinear time using dynamic MSF algorithm.

1.3 Organization of This Thesis

This thesis is organized as follows. In Chapter 2 we review basic concepts related to graph algorithms, introduce notation and review some existing results that we use. In the following three chapters we describe our results. Chapter 3 shows the algorithm for decremental connectivity in planar graphs. In Chapter 4 we describe our algorithms for dynamic Steiner tree problem in general and planar graphs. Then, in Chapter 5 we deal with the algorithms for processing graph timelines. Finally, in Chapter 6 we list some interesting open problems related to the problems we consider.

1.4 Articles Comprising This Thesis

The contents of this thesis have been included in the following papers:

- *Dynamic Steiner tree and subgraph TSP*, joint work with Jakub Oćwieja, Marcin Pilipczuk, Piotr Sankowski, and Anna Zych, preliminary version available in [31].
- *Optimal decremental connectivity in planar graphs*, joint work with Piotr Sankowski, to appear at STACS 2015, preliminary version available in [33].
- *Reachability in graph timelines*, joint work with Piotr Sankowski, published at ITCS 2013 [32].

This thesis contains only some of the results of the papers listed above. Only the results whose main contributor is the author of this thesis are included here.

1.5 Acknowledgments

I would like to thank my supervisor, Piotr Sankowski, for his motivation, numerous fruitful discussions, and patience with answering lots of my questions. I would also like to thank Krzysztof Diks, who helped me whenever it

was needed. I am very grateful to all co-authors of my publications on theoretical computer science: Krishnendu Chatterjee, Tomasz Idziaszek, Tomasz Kulczyński, Yahav Nussbaum, Jakub Oćwieja, Marcin Pilipczuk, Jakub Radoszewski, Christian Wulff-Nilsen and Anna Zych, as well as my friends Łukasz Bieniasz-Krzywiec and Dariusz Leniowski. I would like to express my gratitude to my great teachers: Krzysztof Benedyczak, who taught me programming, and Ryszard Szubartowski, who taught me algorithmics. Finally, I would like to thank my closest relatives, especially my fiancée and my parents for their endless and ongoing support.

During three years of my studies, my research was supported by Google European Doctoral Fellowship in Graph Algorithms, which provided me financial support and saved tons of paperwork.

Chapter 2

Preliminaries

2.1 Graphs

2.1.1 Basic Definitions

An *undirected graph* is a pair $G = (V, E)$, where V is a finite set of vertices and E is a set of *edges*. Each edge is an unordered pair of elements of V, that is $E \subseteq \{\{u, w\} \mid u, w \in V\}$. A *directed graph* is also a pair $G = (V, E)$, where V is a finite set of vertices and E is a set of *edges*. However, edges of a directed graph are *ordered* pairs of elements of V. Unless stated otherwise, when referring to a graph we mean an undirected graph. We typically use the letter n to denote the number of vertices in a graph, and m to denote the number of edges. Moreover, we use $V(G)$ and $E(G)$ to denote the sets of, respectively, vertices and edges of a graph G.

Let $e = \{u, w\}$ be an edge of an undirected graph. We call u and w the *endpoints* of an edge e. In the following, for simplicity, we use uw to denote an edge, whose endpoints are u and w. We say that e is *incident* to u and w, u and w are *adjacent* to e, and u and w are *adjacent* to each other. The *degree* of a vertex is the number of edges incident to it. The neighborhood of a vertex v, denoted $\Gamma(v)$ is the set of vertices adjacent to v.

A *walk* in a graph $G = (V, E)$ is a sequence of vertices $v_1, v_2, \ldots, v_k$, where $k \geqslant 1$, and for $1 \leqslant i < k$, $v_i v_{i+1}$ is an edge of G. The same definition applies to directed graphs. The *endpoints* of a walk $v_1, v_2, \ldots, v_k$ are v_1 and v_k and the *length* of this walk is $k - 1$. A *path* is a walk $v_1, v_2, \ldots, v_k$, where all v_i are distinct.

A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$. Note that since G' is required to be a graph, for every $e' \in E'$, both endpoints of e' belong to V'. With a slight abuse of notation, if $S \subseteq V$, we denote by $G \setminus S$ a subgraph of G obtained by removing vertices of S and

all their incident edges. Similarly, if $v \in V$, we use $G \setminus v$ to denote $G \setminus \{v\}$.

Let $V' \subseteq V$ be a set of vertices. A subgraph of G *induced by* V', denoted $G[V']$ is a subgraph $G' = (V', E')$ of G, where E' is the set of all edges of E, whose both endpoints are in V'. Similarly, for a set $E' \subseteq E$ of edges, we define $G' = (V', E')$ to be an *edge-induced subgraph* of G, if V' is the set of all endpoints of E'.

A graph $G = (V, E)$ is a *bipartite graph* if the set V can be partitioned into two sets V_1, V_2, such that $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, and each edge of G has exactly one endpoint in each of V_1 and V_2. When describing a bipartite graph we sometimes write $G = (V_1 \cup V_2, E)$ to give the aforementioned partition of its vertex set. A *complete graph* is a graph that contains an edge connecting every pair of its vertices. If G is a complete graph over a set of vertices V, we write $G = (V, \binom{V}{2})$. Finally, we say that $G = (V, E)$ is a *multigraph* if V is the set of vertices and E is a *multiset* of pairs of vertices, that is G may have multiple edges between a pair of vertices. The definitions that apply to graphs can be extended to multigraphs in a natural way.

2.1.2 Connectivity

Two vertices u, w of a graph $G = (V, E)$ are *connected* if there is a path in G, whose endpoints are u and w. We say that G is *connected*, if every two vertices of G are connected. A *connected component* of G is a subset $C \subseteq V$, such that every two vertices of C are connected and C is maximal (with respect to inclusion).

Proposition 2.1.1. *Connected components of a graph $G = (V, E)$ form a partition of V.*

Let $G = (V, E)$ be a graph. An edge $e \in E$ is a *bridge*, if $(V, E \setminus \{e\})$ has more connected components than G. A graph $G = (V, E)$ is *biconnected* if it is connected and for every $v \in V$, $G \setminus v$ is connected.

A *separator* of a graph $G = (V, E)$ is a set $S \subseteq V$, such that $G \setminus S$ has more connected components than G. A separator is *balanced* if the size of every connected component of $G \setminus S$ is at most $\alpha|V|$ for some universal constant α (in this thesis we can assume $\alpha = 3/4$).

2.1.3 Weighted Graphs

A graph G is *weighted* if $G = (V, E, d_G)$, and $d_G : E \to \mathbb{R}$ is a function assigning *weights* to edges of G. Throughout this thesis, we assume that the weights are nonnegative. A subgraph of a weighted graph $G = (V, E, d_G)$ is a weighted graph $G' = (V', E', d_{G'})$, such that (V', E') is a subgraph of (V, E)

and $d_{G'}$ is a restriction of d_G to E'. Other definitions for unweighted graphs can be extended to weighted graphs in a similar manner. On the other hand, the definitions for weighted graphs can be used with unweighted ones. In such a case, we assume that the weight of every edge is equal to 1.

Let $G' = (V', E', d_{G'})$ be a subgraph of a weighed graph $G = (V, E, d_G)$. We slightly abuse notation and use $d_G(G')$ to denote $\sum_{e' \in E'} d_{G'}(e')$. We call this value the *weight* of G'. Similarly, for a walk $P = v_1, v_2, \ldots, v_k$ in G we use $d_G(P)$ to denote the *weight* of this walk equal to $\sum_{i=1}^{k-1} d_G(v_i v_{i+1})$.

Let u, w be two vertices of a weighted graph G. The *shortest path* connecting u and w is a minimum weight path whose endpoints are u and w. If u and w are connected, the *distance* between u and w is the weight of the shortest path connecting u and w. Otherwise, the distance between u and w is assumed to be ∞. We denote the distance between u and w by $\delta_G(u, w)$. A *metric closure* of a weighted graph $G = (V, E, d_G)$, denoted $\overline{G}$, is a complete graph $\overline{G} = (V, \binom{V}{2}, d_{\overline{G}})$, where the length of an edge uw is the distance between u and w in G, that is $d_{\overline{G}}(uw) = \delta_G(u, w)$.

2.1.4 Trees

A graph $G = (V, E)$ is a *tree* if for every two vertices $u, w \in V$ there is a unique path connecting u and w. A graph $G = (V, E)$ is a *forest* if for every two vertices $u, w \in V$ there is at most one path connecting u and w.

A vertex v of a tree or forest is called a *leaf* if its degree is equal to 1. A *spanning tree* of a graph $G = (V, E)$ is any subgraph $T = (V, E_T)$ of G which is a tree. A *spanning forest* of a graph $G = (V, E)$ is any subgraph $F = (V, E_F)$ of G which is a forest and has the same number of connected components as G. A *minimum spanning tree* (MST) of a weighted graph $G = (V, E, d_G)$, denoted $MST(G)$, is a spanning tree of G of minimal weight. Similarly, a *minimum spanning forest* (MSF) of a weighted graph $G = (V, E, d_G)$, is a spanning forest of G of minimal weight. For every set $S \subseteq V$, we say that a tree $T = (V, E)$ *spans* S.

Let $G = (V, E, d_G)$ be a weighted graph and $S \subseteq V$ be a subset of vertices. A *Steiner tree* of G, denoted $ST(G)$ is a subgraph $T = (V_T, E_T, d_T)$ of G, such that $S \subseteq V_T$, every two vertices of S are connected in T and T has minimal possible weight. We call S the set of *terminal vertices* or *terminals*.

A *rooted tree* is a tree with a distinguished vertex called the *root*. If $v \in V$ is not a root, we define the *parent* of v, denoted $\textsc{Parent}(v)$ to be the first vertex on the unique path from v to the root. If a vertex u is adjacent to a vertex w and u is not a parent of w, we say that u is a *child* of w. If u and w are two vertices of a rooted tree and u lies on the path from w to the root,

we say that u is an *ancestor* of w and w is a *descendant* of u.

A *binary tree* is a rooted tree, in which every non-leaf vertex has exactly two children. Moreover, we assume that the children of every vertex v are *ordered*, that is there is a distinguished *left child* (denoted $\textsc{Left}(v)$) and a *right child* (denoted $\textsc{Right}(v)$). A *complete binary tree* is a binary tree, in which the distance between the root and every leaf is the same. We call this distance the *height* of the tree.

2.1.5 Planar Graphs

A *plane embedding* of a graph $G = (V, E)$ is a mapping of G into $\mathbb{R}^2$, which maps vertices of G into points and edges of G into simple arcs. Each vertex v is mapped to a distinct point $\pi(v)$ of a plane. An edge uw is mapped to an arc connecting $\pi(u)$ and $\pi(w)$. The arcs corresponding to two distinct edges do not intersect except, possibly, at endpoints. A graph is called *planar* if it admits a plane embedding.

Consider a plane embedding of a planar graph $G = (V, E)$. The arcs of the embedding partition the plane into regions that we call *faces*. Exactly one face is unbounded. We call it the *outer face*. We say that a face f is *adjacent* to the edges corresponding to the arcs bounding f.

Theorem 2.1.2 (Euler's formula)**.** *Let $G = (V, E)$ be a plane embedded graph. Let v be the number of vertices of G, e be the number of edges, f be the number of faces and c be the number of connected components. Then*

$$v - e + f = c + 1.$$

We say that a planar graph is *triangulated* if every face is adjacent to exactly three edges.

A *dual graph* of a planar graph G is a multigraph G^* obtained by embedding a single vertex in every face of G. Let e be an edge of G, which is adjacent to faces f_1 and f_2. For each such edge, we add to G^* the *dual edge* of e, which connects vertices embedded in f_1 and f_2.

Proposition 2.1.3. *A dual graph of a planar graph is planar.*

Note that although we have not defined planar multigraphs (only planar graphs), our definition of planarity can be naturally extended to multigraphs.

A *region* R is an edge-induced subgraph of G. A *boundary vertex* of a region R is a vertex $v \in V(R)$ that is adjacent to an edge $e \notin E(R)$. We denote the set of boundary vertices of a region R by $\partial(R)$. An *r-division* $\mathcal{P}$ of G is a partition of G into $O(n/r)$ edge-disjoint regions (which might

share vertices), such that each region contains at most r vertices and $O(\sqrt{r})$ boundary vertices. The set of boundary vertices of a division $\mathcal{P}$, denoted $\partial(\mathcal{P})$ is the union of the sets $\partial(R)$ over all regions R of $\mathcal{P}$. Note that $|\partial(\mathcal{P})| = O(n/\sqrt{r})$.

Lemma 2.1.4 ([29, 45])**.** *Let $G = (V, E)$ be an n-vertex biconnected triangulated planar graph and $1 \leqslant r \leqslant n$. An r-division of G can be constructed in $O(n)$ time.*

2.2 Segment Trees

Throughout this section we consider *discrete intervals*, that is intervals of integers. The *length* of such an interval is the number of its elements. Let t be a power of 2. We define the set of *elementary intervals* over $1, \ldots, t$ as follows. First, $[1, t]$ is an elementary interval. Second, if $[a, b]$ is an elementary interval and $a < b$, then also $[a, (a + b - 1)/2]$ and $[(a + b + 1)/2, b]$ are elementary intervals. For example, the set of elementary intervals over $1, \ldots, 8$ is $\{[1, 8], [1, 4], [5, 8], [1, 2], [3, 4], [5, 6], [7, 8], [1, 1], [2, 2], [3, 3], [4, 4], [5, 5], [6, 6], [7, 7], [8, 8]\}$. Observe that the elementary intervals can be organized into a complete binary tree, in which the root is $[1, t]$, and for an elementary interval $[a, b]$, where $a < b$, $\textsc{Left}([a, b]) = [a, (a + b - 1)/2]$ and $\textsc{Right}([a, b]) = [(a+b+1)/2, b]$. This tree is called a *segment tree*. In the rest of this section we implicitly assume that elementary intervals we refer to are over $1, \ldots, t$ and a segment tree is a segment tree over this set of elementary intervals.

Proposition 2.2.1. *The height of the segment tree is $O(\log t)$.*

Proof. Consider a path that starts in the root and goes to the left child until a leaf is reached. The height of this tree is the length of this path. Each edge on this path connects an interval with an interval that is half shorter. Since we start in $[1, t]$, the path has length $O(\log t)$. □

Proposition 2.2.2. *There are $2t - 1$ elementary intervals.*

Proof. For $i = 0, \ldots, \log_2 t$, The segment tree has exactly 2^i vertices, whose distance from the root is i. □

From the construction we easily obtain the following.

Proposition 2.2.3. *If $[a_1, b_1]$ and $[a_2, b_2]$ are elementary intervals, then either $[a_1, b_1] \cap [a_2, b_2] = \emptyset$ or one of the intervals is fully contained in the other one.*

The set of elementary intervals can be also characterized as follows.

Lemma 2.2.4. *Let $1 < a \leqslant t$ and $1 \leqslant b \leqslant t$. There exists an elementary interval of length 2^i whose right end is b if and only if b is divisible by 2^i. Similarly, there exists an interval of length 2^i whose left end is a if and only if $a - 1$ is divisible by 2^i.*

Proof. Let $t = 2^d$. We first show that the set of elementary intervals over $1, \ldots, t$ is $A = A_0 \cup A_1 \cup \ldots A_d$, where $A_i = \{[k \cdot 2^i + 1, (k+1) \cdot 2^i] \mid 0 \leqslant k < 2^{d-i}\}$. We have that A_i has exactly 2^{d-i} elements, each being an interval of 2^i elements. In particular, $A_d = \{[1, t]\}$. Moreover, since the sets A_i are disjoint, $|A| = 2t - 1$. By Proposition 2.2.2 there are also $2t - 1$ elementary intervals. Thus the set A and the set of elementary intervals both have size $2t - 1$. To complete the first part of the proof, it suffices to show that every elementary interval is contained in A.

In order to do that we show that for $i > 0$ and any $[a, b] \in A_i$, both $\textsc{Left}([a, b])$ and $\textsc{Right}([a, b])$ belong to A_{i-1}. For simplicity, we only show $\textsc{Left}([a, b]) \in A_{i-1}$. The second claim is similar.

Since $[a, b] \in A_i$, we have that $a = k \cdot 2^i + 1$ and $b = (k+1) \cdot 2^i$ for $0 \leqslant k < 2^{d-i}$. Recall that $\textsc{Left}([a, b]) = [a, (a + b - 1)/2] = [k \cdot 2^i, (2k+1) \cdot 2^{i-1}] = [2k \cdot 2^{i-1}, (2k+1) \cdot 2^{i-1}]$. We set $k' = 2k$. Since $0 \leqslant k < 2^{d-i}$, we have $0 \leqslant k' < 2^{d-(i-1)}$. Moreover, $\textsc{Left}([a, b]) = [k' \cdot 2^{i-1}, (k'+1) \cdot 2^{i-1}]$. Hence, $\textsc{Left}([a, b]) \in A_{i-1}$, so we conclude that A is exactly the set of elementary intervals.

Now, fix a value of $b \leqslant 1$. We have that an interval $[b - 2^i + 1, b] \in A_i$ if and only if $b = (k+1)2^i$ for some $0 \leqslant k < 2^{d-i}$. Since $1 \leqslant b \leqslant t$, we can find a matching k if and only if b is divisible by 2^i. Now, consider $a > 1$. An interval $[a, a + 2^i - 1] \in A_i$ if and only if $a = k \cdot 2^i + 1$ for some $0 \leqslant k < 2^{d-i}$. Since $1 < a \leqslant t$, we can find a matching k if and only if $a - 1$ is divisible by 2^i. The lemma follows. □

We now show that each interval can be partitioned into $O(\log t)$ elementary intervals. Algorithm 1 shows a procedure, which computes such partition.

First, let us note the following property, which follows directly from the pseudocode.

Proposition 2.2.5. *Assume we are computing a decomposition of $[c, d]$ into elementary intervals. In each recursive call $\textsc{Decompose}([c', d'], [a, b])$ we have that $[c', d'] = [c, d] \cap [a, b]$.*

Lemma 2.2.6. *Algorithm 1 produces a decomposition of $[c, d]$ into elementary intervals.*

Algorithm 1

```
1: function DECOMPOSE([c, d], [a, b])   ▷ Decompose [c, d] into elementary
   intervals, which are sub-intervals of [a, b]
   Require: [a, b] is an elementary interval and [c, d] ⊆ [a, b]
2:     if [c, d] = [a, b] then return {[a, b]}
3:     ret := ∅
4:     if [c, d] ∩ LEFT([a, b]) ≠ ∅ then
5:         ret := ret ∪ DECOMPOSE([c, d] ∩ LEFT([a, b]), LEFT([a, b]))
6:     if [c, d] ∩ RIGHT([a, b]) ≠ ∅ then
7:         ret := ret ∪ DECOMPOSE([c, d] ∩ RIGHT([a, b]), RIGHT([a, b]))
       return ret
```

Proof. Consider the first parameter $[c, d]$ of DECOMPOSE. In every call we either return a decomposition that contains solely of $[c, d]$ or call DECOMPOSE recursively. The first parameters of the recursive calls form a partition of $[c, d]$. Thus, every element of $[c, d]$ is either returned in a decomposition or passed to a further recursive call. Consequently, the decomposition we return is a partition of $[c, d]$.

Observe that the algorithm terminates, as in every recursive call the length of the second parameter of DECOMPOSE halves. Once we reach an interval of length 1, that is we call DECOMPOSE$([c', d'], [a, a])$, we know that $[c', d'] \subseteq [a, a]$ and $[c', d']$ is nonempty (this is a necessary condition to execute the call). Thus, the call terminates returning an interval $[a, a]$. The lemma follows. □

Lemma 2.2.7. *A call to* DECOMPOSE$([c, d], [a, b])$*, where* $d = b$ *or* $c = a$*, requires* $O(\log(b - a + 1))$ *time and returns* $O(\log(b - a + 1))$ *elementary intervals.*

Proof. We assume $d = b$, the other case is analogous. If LEFT$([a, b]) \cap [c, d] \neq \emptyset$, then RIGHT$([a, b]) \subseteq [c, d]$. Thus, both LEFT$([a, b])$ and RIGHT$([a, b])$ intersect $[c, d]$, so DECOMPOSE$([c, d], [a, b])$ makes two recursive calls. The second one is DECOMPOSE$([c, d] \cap$ RIGHT$([a, b])),$ RIGHT$([a, b]))$, but since RIGHT$([a, b]) \subseteq [c, d]$, the first parameter is simply RIGHT$([a, b])$. Hence, this call terminates immediately and returns a single interval, so only the other recursive call may trigger further recursive calls.

On the other hand, if LEFT$([a, b]) \cap [c, d] = \emptyset$ we only make one recursive call. In both cases, we spend $O(1)$ time and execute a single recursive call. The second parameter of this recursive call is an interval which is half the size of $[a, b]$. Hence, altogether we spend $O(\log(b-a+1))$ time. Consequently, the length of the produced decomposition is bounded by $O(\log(b - a + 1))$. □

Lemma 2.2.8. *Let $1 \leqslant a \leqslant b \leqslant t$. The interval $[a, b]$ can be partitioned into $O(\log t)$ elementary intervals over $1, \ldots, t$ in $O(\log t)$ time.*

Proof. We use Algorithm 1. By Lemma 2.2.6 the algorithm is correct. It remains to bound the running time. From this, it would follow that the returned decomposition has length $O(\log t)$.

Consider the first recursive call to DECOMPOSE which calls DECOMPOSE twice. We call it a *branching call*. Note that there are at most $O(\log t)$ calls before a branching call (or $O(\log t)$ calls in total, if there is no branching call), as in each call the second parameter is an interval that is two times shorter.

In a branching call we have $[c, d] \cap \textsc{Left}([a, b]) \neq \emptyset$ as well as $[c, d] \cap \textsc{Right}([a, b]) \neq \emptyset$. Hence, we may apply Lemma 2.2.7 to both recursive calls that are made and bound their total running time by $O(\log(b - a + 1)) = O(\log t)$. The lemma follows. □

Note that an interval has multiple possible decompositions into elementary intervals. However, in the following we assume that we use a decomposition produced according to Lemma 2.2.8.

Lemma 2.2.9. *Let $[a, b]$ be an elementary interval, such that $[a, b] \subseteq [c, d]$. Then the decomposition of $[c, d]$ into elementary intervals contains either $[a, b]$ or one of its ancestors.*

Proof. Observe that for every $x \in [a, b]$ the intervals containing x are ancestors of $[a, b]$, $[a, b]$ itself and (a subset of) descendants of $[a, b]$. It suffices to show that no descendants of $[a, b]$ belong to the decomposition.

Consider a recursive call $\textsc{Decompose}([c', d'], [a, b])$. By Proposition 2.2.5, $[c', d'] = [c, d] \cap [a, b]$). Since $[a, b] \subseteq [c, d]$, we have $[c', d'] = [a, b]$, so the call returns immediately. Consequently, DECOMPOSE is never called for any descendant of $[a, b]$ (as a second argument). The lemma follows. □

2.3 Algorithms

2.3.1 Approximation Algorithms

Consider an optimization problem, in which the goal is to compute an object, which satisfies certain properties and has minimal possible weight. An algorithm is called α-approximate (for $\alpha \geqslant 1$), if it computes a feasible object, whose weight is at most α times the optimal weight.

The following Lemma gives a 2-approximate algorithm for computing the Steiner tree.

Lemma 2.3.1. *Let $G = (V, E, d_G)$ be a weighted graph and $S \subseteq V$. Then, $d_G(ST(G)) \leqslant MST(\overline{G}[S]) \leqslant 2d_G(ST(G))$.*

Although no polynomial-time algorithm is known for the Steiner tree problem, both the metric closure and its minimum spanning tree can be computed in polynomial time.

2.3.2 Dynamic Graph Algorithms

A *dynamic graph algorithm* is an algorithm that maintains some information about a graph G, which is undergoing modifications. Typically the modifications, in the following called *updates*, are edge additions or removals. The sequence of updates is intermixed with a set of *queries*, e.g., about the existence of a path between two vertices or about the weight of the minimum spanning tree. The algorithm is supposed to answer queries faster than by computing the answer from scratch.

In this thesis we work with dynamic graph problems, in which the updates change the set of edges in the graph. There are three types of dynamic graph problems, depending on the allowed modifications. In an *incremental* problem, edges may only be added, whereas in a *decremental* one, edges can only be deleted. Finally, a *fully dynamic* problem allows both edge insertions and deletions.

In our algorithms we use fully dynamic algorithm that maintains a minimum spanning forest of a graph. In the following, we call it a dynamic MSF algorithm.

Theorem 2.3.2 ([23])**.** *There exists a fully dynamic MSF algorithm, that for a graph on n vertices supports m edge additions and removals in $O(m \log^4 n)$ total time.*

2.3.3 Connectivity

Let $G = (V, E)$ be a graph with n vertices and m edges. It is well known that, using depth-first search or breadth-first search algorithms we may find the connected components of G.

Proposition 2.3.3. *Connected components of G can be found in $O(n + m)$ time.*

Formally, the algorithm computes for each vertex the unique identifier of its connected component. Two vertices belong to the same connected components if and only if their identifiers are equal.

Disjoint-Set Data Structure

A *disjoint-set data structure* (further denoted by DSDS, also called an *union-find data structure*) maintains a partition of a set of elements into disjoint subsets. In each subset one element is selected as a *representative*. The data structure supports two operations. First, given any element x, it can return the representative of the subset containing x (this is called the `find` operation). It can be used to test whether some two elements belong to the same set of the partition. Moreover, the data structure supports a `union` operation, which, given two elements x and y, merges the subsets containing x and y. A famous result by Tarjan [40] bounds the running time of a previously known union-find algorithm.

Theorem 2.3.4 ([40])**.** *There exists a disjoint-set data structure that supports any sequence of m operations on an universe of size n in $O((n+m)\alpha(n))$ time, where α is the inverse of Ackerman's function.*

In this thesis however, we need an DSDS, in which the running time of every individual operation is bounded. Hence, we use a simple data structure that we now describe. Each subset in the data structure is represented as a rooted tree. The root of the tree is a representative of a subset, and every other element of a subset maintains a pointer to its parent in the tree. We assume that the parent pointer of a representative points to itself. Moreover, each representative maintains the size of its subset.

In order to perform the `find` operation, it suffices to follow the parent pointers, until the representative is reached. To union two subsets we first find its representatives x and y. If $x = y$, nothing has to be done. Otherwise, we compare the sizes of their sets. Assume that the subset of x is not greater than the subset of y. In such a case we merge the sets by making y the parent of x.

Let us list some simple properties of the described DSDS.

Proposition 2.3.5. *The algorithm takes $O(n)$ time to initialize and performs every operation in $O(\log n)$ worst-case time.*

Proposition 2.3.6. *The `find` operation does not change the data structure. Every `union` operation can change at most one parent pointer in the data structure. When a change is performed, one representative of the merged subsets becomes a parent of the other representative.*

The DSDS is closely related to *incremental connectivity problem*, that is a problem in which we work with a dynamic graph, subject to edge insertions. The sequence of insertions is intermixed with queries of the form 'Are

vertices u and w connected?'. It is easy to see that we may use DSDS to solve incremental connectivity problem: a query can be solved by a `find` operation, whereas an update maps naturally to a `union` operation. Let $G = (V, E)$ be a graph. We say that a DSDS D is a DSDS of G if D is obtained by performing a `union` operation for the endpoints of each edge of G.

Proposition 2.3.7. *Let $G = (V, E)$ be a graph, and $n = |V|$. Given a DSDS of a graph G, we can find the connected components of G in $O(n)$ time.*

Proof. We view the forest maintained by the DSDS as an undirected graph F. Observe that the connected components of F are the same as the connected components of G. Thus, by Proposition 2.3.3, we can find connected components of F in $O(n)$ time. □

2.3.4 String Hashing

We use a string hashing scheme based on the fingerprinting technique of Rabin and Karp [28]. This scheme can be used to compute hash values (henceforth called *hashes*) of strings, that may be used for probabilistic equality testing. Throughout this section, let us assume that we work with sequences of length at most n, that consist of positive integers not greater than M. Let $p > \max(M, n)$ be a prime number and $B \in \{0, \ldots, p-1\}$ be chosen uniformly at random. The hash value of a sequence $S = s_1, \ldots, s_k$ is

$$H(S) = \left(\sum_{i=0}^{k-1} B^{k-1-i} s_i \right) \bmod p. \tag{2.1}$$

While the scheme described in [28] chooses p randomly, we modify the scheme slightly, as done, e.g., in [30]. We fix p and then randomly pick B. As shown in [30], this assures that the probability of two distinct sequences having the same hash value is at most n/p. Thus, by choosing a value of p that is suitably large, yet polynomial in n, distinct sequences have distinct hash values with high probability.

We now derive useful properties of this hashing scheme. Note that in order to use these properties, together with each hash we need to store the length of the sequence represented by the hash.

Proposition 2.3.8. *The hash of a sequence $s_1, \ldots, s_k$ can be computed in $O(k)$ time.*

Proof. We use Horner's rule to evaluate Formula 2.1. □

Let $S_1 = a_1, \ldots, a_k$ and $S_2 = b_1, \ldots, b_l$ be two sequences, $h_1 = H(S_1)$ and $h_2 = H(S_2)$. We denote by $h_1 \oplus h_2$ the hash of a sequence obtained by appending S_2 to S_1. Moreover, if S_2 be a prefix of S_1, we denote by $h_1 \ominus h_2$ the hash of $a_{l+1}, a_{l+2}, \ldots a_k$

Lemma 2.3.9. *Let S_1 and S_2 be two sequences. Given $H(S_1)$ and $H(S_2)$, we may compute $H(S_1) \oplus H(S_2)$ in $O(1)$ time. This requires initial preprocessing in $O(n)$ time.*

Proof. Let $S_1 = a_1, \ldots, a_k$ and $S_2 = b_1, \ldots, b_l$. We have that $H(S_1) \oplus H(S_2) = (H(S_1)B^l + H(S_2)) \bmod p$. This can be evaluated in $O(1)$ time, if we preprocess $B^i \bmod p$, for all $1 \leqslant i \leqslant n$. □

Lemma 2.3.10. *Let S_1 be a sequence, and S_2 be a prefix of S_1. Given $H(S_1)$ and $H(S_2)$, we may compute $H(S_1) \ominus H(S_2)$ in $O(1)$ time. This requires initial preprocessing in $O(n)$ time.*

Proof. Let $S_1 = a_1, \ldots, a_k$ and $S_2 = a_1, \ldots, a_l$ for $l \leqslant k$. We have that $H(S_1) \ominus H(S_2) = (H(S_1) - B^{k-l}H(S_2)) \bmod p$. This can be evaluated in $O(1)$ time, if we preprocess $B^i \bmod p$, for all $1 \leqslant i \leqslant n$. □

Lemma 2.3.11. *Let $S = a_1, \ldots, a_k$, where for each $1 \leqslant i \leqslant k$, $a_i = a_1$. Then, we may compute $H(S)$ in constant time. This requires initial preprocessing in $O(n)$ time.*

Proof. We have that $H(S) = \left(s_1 \sum_{i=0}^{k-1} B^i\right) \bmod p$. This can be evaluated in constant time, if we preprocess $\sum_{i=0}^{k-1} B^i$ for all $1 \leqslant i \leqslant n$. □

2.4 Other Remarks

Throughout this thesis we use $\log x$ to denote the binary logarithm of x. Moreover, we use $\log^* n$ to denote the iterated logarithm function. We have $\log^* n = 0$ for $n \leqslant 1$, and $\log^* n = 1 + \log^*(\log n)$ for $n > 1$. We also use the *soft-O* notation and write $\widetilde{O}(f(n))$ to denote $O(f(n)\text{polylog}(n))$. Note that f may have multiple arguments, and the soft-O notation hides factors which are polylogarithmic in each of them.

We assume word-RAM model with standard instructions. This means that the machine word has size $w \geqslant \log n$. Here, n denotes the size of the input data. All basic arithmetic and logical operations (including multiplication and bit shifts) on integers of at most w bits take unit time.

Chapter 3

Decremental Connectivity in Planar Graphs

In this chapter we show an algorithm for decremental connectivity in planar graphs. The algorithm, at any point, given two vertices of the graph may answer whether they belong to the same connected component. The total running time of this algorithm is linear in the size of the graph. By the total running time we denote the total time of handling any sequence of deletions of edges. Each query is answered in constant time.

In the following part of this chapter we first introduce some definitions and deal with minor technical issues (in Section 3.1). Then, we present the algorithm, by introducing our ideas one by one. Each each idea results in a faster algorithm. We describe a simple $O(n \log n)$ algorithm in Section 3.2, and then, in Section 3.3, present how to speed it up to $O(n \log \log n)$ time using r-division. Next, in Section 3.4, we show that our idea can be used recursively, which results in the running time of $O(n \log \log \log n)$. Finally, in Section 3.5 we make the algorithm linear by precomputing connectivity information of all graphs of bounded size.

3.1 Preliminaries

In this chapter we describe multiple distinct connectivity algorithms. Some of them maintain identifiers of connected components. These identifiers (henceforth denoted *cc-identifiers*) are values assigned to vertices, which uniquely identify the connected components. Two vertices have the same cc-identifiers if and only if they belong to the same connected component. We say that an algorithm maintains cc-identifiers *explicitly* if after every deletion it returns the list of changes to the cc-identifiers. We assume that cc-identifiers

are integers that require $\log n + O(1)$ bits.

Proposition 3.1.1. *A dynamic graph algorithm which explicitly maintains cc-identifiers implies a dynamic connectivity algorithm with the same update time and constant query time.*

Let G be a planar graph. In the preprocessing phase of our algorithms, we build an r-division of G (see Section 2.1.5). This r-division is updated in a natural way, as edges are deleted from G. Namely, when an edge is deleted from the graph, we update its r-division by deleting the corresponding edge. However, if we strictly follow the definition, what we obtain may no longer be an r-division.

For that reason, we loosen the definition of an r-division, so that it includes the divisions obtained by deleting edges. Consider an r-division $\mathcal{P}$ built for a graph G. Moreover, let G' be a graph obtained from G by deleting edges, and let $\mathcal{P}'$ be the r-division $\mathcal{P}$ updated in the following way. Let R be a region of $\mathcal{P}$. Then, we define the graph R' in $\mathcal{P}$ obtained by removing edges from R to be a region of $\mathcal{P}'$, although it may no longer be an edge-induced subgraph of G', e.g., it may contain isolated vertices. Similarly, we define the set of boundary vertices of $\mathcal{P}'$ to be the set of boundary vertices of $\mathcal{P}$. Again, according to this definition, a boundary vertex v of $\mathcal{P}'$ may be incident to edges of a single region (because the edges incident to v that belonged to other regions have been deleted). In the following, we say that $\mathcal{P}'$ is an r-division of G'.

In order to compute an r-division, we use Lemma 2.1.4. Since the Lemma requires the graph to be biconnected and triangulated, in order to obtain an r-division for a graph which does not have these properties, we first add edges to G to make it biconnected and triangulated, then compute the r-division of G, and finally delete the added edges both from G and its division.

Without loss of generality, we can assume that each vertex $v \in V$ has degree at most 3. This can be assured by triangulating the dual graph in the very beginning. In particular, this assures that each vertex belongs to a constant number of regions in an r-division.

3.2 $O(n \log n)$ Time Algorithm

Let G be a planar graph subject to edge deletions. We call an edge deletion *critical* if and only if it increases the number of components of G, i.e., the deleted edge is a bridge in G. We first show a dynamic algorithm that for every edge deletion decides, whether it is critical. It is based on a simple relation between the graph G and its dual.

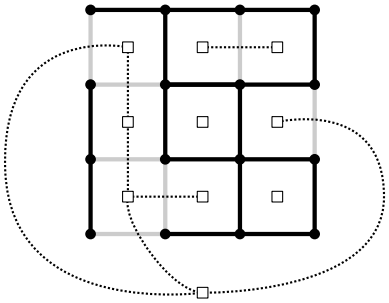

Figure 3.1: The graphs that illustrate the proof of Lemma 3.2.2. Edges of G are drawn with solid black lines, whereas the gray lines depict edges that have been deleted from G. The small squares are vertices of D_G, and the dotted lines are edges of D_G.

Lemma 3.2.1. *Let G be a planar graph subject to edge deletions. There exists an algorithm that for each edge deletion decides whether it is critical. It runs in $O(n)$ total time.*

Proof. The intuition behind the proof is as follows. We maintain the number of faces in G. In order to do that, when an edge e is deleted, we simply merge faces on both sides of e (if they are different from each other). This can be implemented using union-find data structure on the vertices of the dual graph G^*.

More formally, we build and maintain a graph D_G. Initially, this is a graph consisting of vertices of G^* (faces of G). When an edge is deleted from G, we add its dual edge to D_G (see Figure 3.1). Clearly, the connected components of D_G are exactly the faces of G. Since edges are only added to D_G, we can easily maintain the number of connected components in D_G with a union-find data structure.

This allows us to detect critical deletions in G. We use Euler's formula (see Theorem 2.1.2). After every edge deletion, we know the number of edges and vertices of G. Moreover, we know that the number of faces of G is equal to the number of connected components of D_G, which we also maintain. As a result, by Euler's formula, we get the number of connected components of G, so in particular we may check if the deletion causes the number of connected components to increase. The algorithm executes $O(n)$ find and union operations on the union-find data structure.

In addition to that, the sequence of union operations has a certain structure. Let G_1 be the initial version of the graph G (before any edge deletion). Observe that each union operation takes as arguments the endpoints of an

edge of G_1^*. The variant of the union-find problem, in which the set of allowed union operations forms a planar graph given during initialization, was considered by Gustedt [19]. He showed that for this special case of the union-find problem there exists an algorithm that may execute any sequence of $O(n)$ operations in $O(n)$ time (for an n-vertex planar graph). Thus, we infer that our algorithm runs in $O(n)$ time. □

We can now use Lemma 3.2.1 to show a simple decremental connectivity algorithm that runs in $O(n \log n)$ total time.

Lemma 3.2.2. *Let G be a planar graph subject to edge deletions. There exists a decremental connectivity algorithm that for every vertex of G maintains its cc-identifier explicitly. It runs in $O(n \log n)$ total time.*

Proof. We use Lemma 3.2.1 to detect critical deletions. When an edge uw is deleted, and the deletion is not critical, nothing has to be done. Otherwise, after a critical deletion, some connected component C breaks into two components C_u and C_w ($u \in C_u$, $w \in C_w$) and we start two parallel depth-first searches from u and w. We stop both searches once the first of them finishes. W.l.o.g. assume that it is the search started from u. Thus, we know that the size of C_u is at most half of the size of C.[1] We can now iterate through all vertices of C_u and change their cc-identifiers to a new unique number. All these steps require $O(|C_u|)$ time. The running time of the algorithm is proportional to the total number of changes of the cc-identifiers. Since every vertex changes its identifier only when the size of its connected component halves, we infer that the total running time is $O(n \log n)$. □

3.3 $O(n \log \log n)$ Time Algorithm

In order to speed up the $O(n \log n)$ algorithm, we need to speed up the linear depth-first searches that are run after a critical edge deletion. We build an r-division $\mathcal{P}$ of G for $r = \log^2 n$ and use a separate decremental connectivity algorithm to maintain the connectivity information inside each region. On top of that, we maintain a *skeleton graph* that represents connectivity information between the set of boundary vertices (and possibly some other vertices that we consider important). Loosely speaking, since the number of boundary vertices is $O(n/\log n)$ we can pay a cost of $O(\log n)$ for maintaining each cc-identifier.

[1] Since the graph has constant degree, we may assure that both searches are synchronized in terms of number of vertices visited.

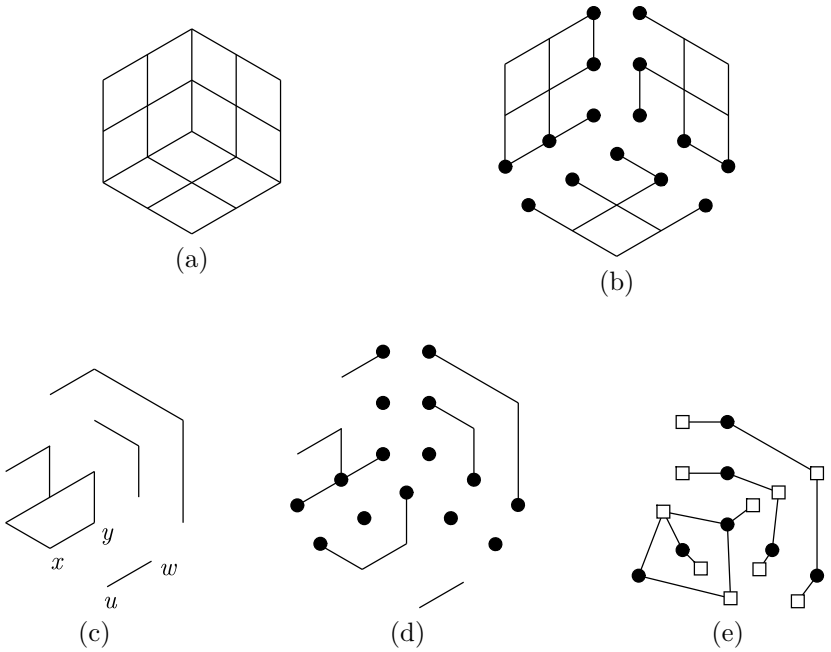

Figure 3.2: Panels 3.2a and 3.2b show a sample graph G and its r-division into three regions (boundary vertices are marked with small circles). In panel 3.2c there is graph G' obtained from G by a sequence of edge deletions. Panel 3.2d shows its r-division obtained from the r-division of G (again, boundary vertices are marked with small circles). Finally, panel 3.2e contains the skeleton graph of G' (for $V_s = \partial(\mathcal{P})$). Auxiliary vertices are marked with squares.

Definition 3.3.1. *Consider an r-division $\mathcal{P}$ of a planar graph $G = (V, E)$ and a set V_s (called a* skeleton set*), such that $\partial(\mathcal{P}) \subseteq V_s \subseteq V$. The* skeleton graph *for $\mathcal{P}$ and V_s is a graph over the skeleton set V_s and some additional auxiliary vertices. Consider a region R of $\mathcal{P}$. Group vertices of $V_s \cap V(R)$ into sets $V_1, \ldots, V_k$, such that two vertices belong to the same set if and only if there is a path in R that connects them. For each set V_i add a new auxiliary vertex w_i and add an edge $w_i x$ for every $x \in V_i$.*

For illustration, see Figure 3.2.

Proposition 3.3.2. *The skeleton graph has $O(|V_s|)$ vertices and edges.*

Proof. For a region R, we add to the skeleton graph at most one vertex and edge per each vertex of $V_s \cap V(R)$. Since each vertex belongs to a constant number of regions, we get the desired bound. □

Proposition 3.3.3. *If $u, w \in V_s$, then u and w are connected in the skeleton graph if and only if they are connected in G.*

Proof. Consider a region R of the r-division. From the construction it follows that two vertices of $V_s \cap V(R)$ are connected in G with a path inside R if and only if they are connected in the part of the skeleton graph built for this region.
($\Longrightarrow$) Follows directly from the above observation.
($\Longleftarrow$) Consider a path P in G between u and w. Break this path into subpaths at each element of V_s. Since $\partial(\mathcal{P}) \subseteq V_s \subseteq V$, each resulting subpath is fully contained in one region of the r-division. Clearly, from the property given at the beginning of the proof, for each subpath there exists a corresponding path in the skeleton graph. □

The skeleton graph is also planar, but our algorithms do not use this property.

In our algorithm we update the skeleton graph of G, as edges are deleted. As in the $O(n \log n)$ time algorithm, we need a way of detecting whether an edge deletion in G increases the number of connected components in the skeleton graph.

Lemma 3.3.4. *Let G be a dynamic planar graph, subject to edge deletions. Assume that we maintain its skeleton graph G_s computed for an r-division $\mathcal{P}$ and a skeleton set V_s. An edge deletion in G causes an increase in the number of connected components in G_s if and only if the deletion is critical in G and there exists a region of $\mathcal{P}$, in which the deletion disconnects some two vertices of V_s.*

Before we proceed with the proof, let us note that all its conditions are necessary. In particular, a critical deletion in G may not disconnect some two vertices of a skeleton set in a region (e.g., edge uw in Figure 3.2c, whose deletion does not affect the skeleton graph at all). It may also happen that the deletion is not critical in G, but inside some region it disconnects some two vertices of V_s (e.g., edge xy in Figure 3.2c).

Proof. By Proposition 3.3.3, two vertices of V_s are connected in G if and only if they are connected in G_s.
($\Longrightarrow$) If two vertices of V_s become disconnected in G_s, they also become disconnected in G, so the edge deletion is critical. The deletion has to disconnect some two vertices in a region, because otherwise the graph G_s would not change at all.
($\Longleftarrow$) Assume that the deletion disconnected vertices $u, w \in V_s$ in a region R. Thus, the deleted edge was on some path from u to w. Since the edge deletion is critical in G, the deleted edge was a bridge in G. After the deletion there is no path from u to w in G and consequently also in G_s. □

Lemma 3.3.5. *Let $G = (V, E)$ be a planar graph and let $X \subseteq V$. Assume there exists a decremental connectivity algorithm that maintains cc-identifiers of a set $X \subseteq V$ explicitly and processes updates in $\Omega(n)$ total time. Then, we can extend the algorithm, so that:*

- *after every edge deletion, if the deletion disconnects some two vertices of X, it reports a pair of vertices that become disconnected,*
- *given a cc-identifier, it returns a vertex $v \in X$ with the same cc-identifier (or reports that such a vertex does not exist).*

The extended algorithm has the same asymptotic running time.

Proof. Since each cc-identifier can be encoded in $\log n + O(1)$ bits, there are $O(n)$ possible cc-identifiers. Thus, for each possible cc-identifier c, we maintain a list L_c of vertices of X with this cc-identifier. Observe that maintaining these lists takes time that is linear in the number of changes of cc-identifiers. Moreover, we need $O(n)$ time to initialize the lists L_c.

Observe that the lists allow us to find a vertex of X of given cc-identifier in constant time, so the second claim follows. To show the first claim, consider a case when after an edge deletion some (but not all) elements from a list L_c are removed. All this elements have to be added to a single list $L_{c'}$ and $L_{c'}$ must have been empty before the new elements were added (because an edge deletion may not cause two vertices to become connected). This means that the number of distinct cc-identifiers has increased, and some elements of X

became disconnected. We can now take any $u \in L_c$ and $w \in L_{c'}$ and report that u and w became disconnected. □

We are ready to show the main building block of our $O(n \log\log n)$ time algorithm.

Lemma 3.3.6. *Let G be a planar graph. Assume there exists a decremental connectivity algorithm that runs in $f(n)$ time, where f is a nondecreasing function, and maintains cc-identifiers explicitly. Then, there exists a decremental connectivity algorithm that runs in $O(n + n \cdot f(\log^2 n)/\log^2 n)$ time and answers queries in $O(1)$ time.*

Proof. We build an r-division $\mathcal{P}$ of G for $r = \log^2 n$. By Lemma 2.1.4, this takes $O(n)$ time. For each region R of the division, we run the assumed decremental algorithm to handle edge deletions. We use A_R to denote the algorithm run for region R. A_R maintains cc-identifiers of $V(R)$ explicitly. We call these cc-identifiers *local* cc-identifiers. We also extend each A_R according to Lemma 3.3.5, taking $X = \partial(\mathcal{P}) \cap V(R)$. Moreover, we use Lemma 3.2.1 to detect critical deletions in G.

We build the skeleton graph G_s of G, for an r-division $\mathcal{P}$ and a skeleton set $V_s = \partial(\mathcal{P})$. We maintain G_s as edges are deleted, that is the deletions in G are reflected in G_s. This can be done using the algorithms A_R. By Lemma 3.3.5, A_R can report that some two vertices of V_s become disconnected inside R. This means that G_s needs to be updated. Observe that the part of G_s inside a region R can be implicitly represented as a partition of $V_s \cap V(R)$, where two vertices belong to the same element of the partition, if they are connected in R. Thus, if a deletion causes t local cc-identifiers to change, we may update G_s in $O(t)$ time. As a result, the time for updating G_s is linear in the number of local cc-identifiers that are changed.

For every vertex of G_s, we maintain its cc-identifier (called a *global* cc-identifier). Once G_s is updated after an edge deletion, we use Lemma 3.3.4 to check whether the number of connected components of G_s increased. According to the lemma, it suffices to check whether the deletion is critical in G (this is reported by the algorithm of Lemma 3.2.1), and whether some two elements of the skeleton set became disconnected within some region (using Lemma 3.3.5).

When we detect that the number of connected components of the skeleton graph G_s has increased, similarly to the $O(n \log n)$ algorithm, we run two parallel DFS searches to identify the smaller of the two new connected components, and update the global cc-identifiers.

In order to answer a query regarding two vertices u and w, we perform two checks. First, if the vertices belong to the same region, we check whether

there exists a path connecting them that does not contain any boundary vertices. This can be done by querying algorithm A_R for the appropriate region.

Then, we check whether there is a path from u to w that contains some boundary vertex. For each of the two vertices, we find two arbitrary boundary vertices b_u and b_w that u and w are connected to (using Lemma 3.3.5). Then, we check whether b_u and b_w have the same global cc-identifier.

Let us now analyze the running time. The algorithm of Lemma 3.2.1 requires $O(n)$ time. The total running time of algorithms A_R is $O(n \cdot f(r)/r) = O(n \cdot f(\log^2 n)/\log^2 n)$. Lastly, we bound the running time of the DFS searches performed to update the global cc-identifiers. We use an argument similar to the one in the proof of Lemma 3.2.2. The skeleton graph has $O(n/\log n)$ vertices, and each global cc-identifier can change at most $O(\log(n/\log n)) = O(\log n)$ times. Hence, the DFS searches require $O((n/\log n)\log n) = O(n)$ time. The lemma follows. □

By applying Lemma 3.3.6 to Lemma 3.2.2, we obtain the following.

Lemma 3.3.7. *There exists a decremental connectivity algorithm for planar graphs that runs in $O(n \log\log n)$ total time.*

Proof. The total update time of the algorithm of Lemma 3.2.2 is $f(n) = O(n \log n)$. Thus, the running time is $O(n + n \cdot f(\log^2 n)/\log^2 n) = O(n + n \log^2 n \log\log n/\log^2 n) = O(n \log\log n)$. □

3.4 $O(n \log\log\log n)$ Time Algorithm

In order to obtain an even faster algorithm, we would like to use Lemma 3.3.6 multiple times, starting from the $O(n \log n)$ algorithm, and each time applying the lemma to the algorithm obtained in the previous step. This, however, cannot be done directly. While the lemma requires an algorithm that maintains all cc-identifiers explicitly, it does not produce an algorithm with this property. We deal with this problem in this section.

Observe that in the proof of Lemma 3.3.6 we only needed the assumed decremental algorithm to maintain the cc-identifiers of the vertices of the skeleton set. This fact can be exploited in the following way. We show that if we have an algorithm that maintains cc-identifiers of some vertices, we may construct another (possibly faster) algorithm with the same property.

Lemma 3.4.1. *Assume there exists a decremental connectivity algorithm for planar graphs that, given a graph $G = (V, E)$ and a set $V_e \subseteq V$ (called an* explicit set*):*

- *maintains cc-identifiers of the vertices of V_e explicitly,*
- *processes updates in $f(n) + O(|V_e| \log n)$ time,*
- *may return the cc-identifier of any vertex in $g(n)$ time,*

where $f(n)$ and $g(n)$ are nondecreasing functions.

Then, there exists a decremental connectivity algorithm for planar graphs, which, given a graph $G = (V, E)$ and a set $V_e \subseteq V$:

- *maintains cc-identifiers of the vertices of V_e explicitly,*
- *processes updates in $O(n + |V_e| \log n + n \cdot f(\log^2 n)/\log^2 n)$ time,*
- *may return the cc-identifier of any vertex in $g(\log^2 n) + O(1)$ time.*

Proof. We build an r-division $\mathcal{P}$ of G for $r = \log^2 n$. By Lemma 2.1.4, this takes $O(n)$ time. We also build a skeleton graph G_s, by taking a skeleton set $V_s := V_e \cup \partial(\mathcal{P})$. Hence, $|V_s| = |V_e| + O(n/\log n)$.

For each region R of $\mathcal{P}$, we run a copy A_R of the assumed decremental connectivity algorithm, extended according to Lemma 3.3.5. Observe that in the proof of Lemma 3.3.6, we only need A_R to explicitly maintain cc-identifiers of $V_s \cap V(R)$. Thus, the set of explicit vertices for algorithm A_R is $V_s \cap V(R)$. Hence, A_R maintains local cc-identifiers of these vertices.

We maintain the graph G_s and its global cc-identifiers in the same way as in the proof of Lemma 3.3.6. The only difference is that now the skeleton set V_s is bigger. Let us bound the running time. Algorithm A_R uses $f(\log^2 n) + O(|V_s \cap V(R)| \log n)$ time. Summing this over all regions, we obtain

$$\begin{aligned}\sum_{R \in \mathcal{P}} & f(\log^2 n) + O(|V_s \cap V(R)| \log n) \\ &= O(n \cdot f(\log^2 n)/\log^2 n + |V_s| \log n) \\ &= O(n \cdot f(\log^2 n)/\log^2 n + |V_e| \log n + n/\log n \cdot \log n) \\ &= O(n \cdot f(\log^2 n)/\log^2 n + |V_e| \log n + n).\end{aligned}$$

Note that we use the fact that each vertex is contained in a constant number of regions. The running time of depth-first searches used to update the global cc-identifiers is

$$O(|V_s| \log n) = O(n/\log n \cdot \log n + |V_e| \log n) = O(n + |V_e| \log n).$$

Thus, the total update time is $O(n + |V_e| \log n + n \cdot f(\log^2 n)/\log^2 n)$.

Since the cc-identifiers of vertices of G_s are maintained explicitly, in particular we explicitly maintain the cc-identifiers of vertices of V_e. It remains

to describe the process of computing the global cc-identifier of an arbitrary vertex $v \in V$. Assume that v belongs to a region R (if v is a boundary vertex, we may use an arbitrary region containing it). We first query A_R to obtain the local cc-identifier of v. We use Lemma 3.3.5 to check whether there exists a vertex b_v in $V_s \cap V(R)$ that has the same local cc-identifier as v. If this is the case, since b_v belongs to the skeleton set, we return its global cc-identifier (maintained explicitly). Otherwise, we return a new cc-identifier by encoding as an integer a pair consisting of the identifier of the region containing v (this requires $\log O(n/\log^2 n) = \log n + O(1) - 2\log\log n$ bits) and the local cc-identifier of v (which requires $\log\log^2 n + O(1) = 2\log\log n + O(1)$ bits). Overall, the resulting cc-identifier requires $\log n + O(1)$ bits. Thus, obtaining a cc-identifier of an arbitrary vertex requires $g(\log^2 n) + O(1)$ time. □

The main advantage of Lemma 3.4.1 over Lemma 3.3.6 is that we may apply Lemma 3.4.1 recursively to obtain better algorithms. We can view applying Lemma 3.4.1 as reducing connectivity in a graph of size n to connectivity in a collection of graphs of size $\log^2 n$. If we apply Lemma 3.4.1 to itself, we obtain the following.

Lemma 3.4.2. *Assume there exists a decremental connectivity algorithm for planar graphs that, given a graph $G = (V, E)$ and a set $V_e \subseteq V$:*

- *maintains cc-identifiers of the vertices of V_e explicitly,*
- *processes updates in $f(n) + O(|V_e| \log n)$ time,*
- *may return the cc-identifier of any vertex in $g(n)$ time,*

where $f(n)$ and $g(n)$ are nondecreasing functions.

Then, there exists a decremental connectivity algorithm for planar graphs, which, given a graph $G = (V, E)$ and a set $V_e \subseteq V$:

- *maintains cc-identifiers of the vertices of V_e explicitly,*
- *processes updates in $O(n+|V_e|\log n+n\cdot f(\log^2\log^2 n)/\log^2\log^2 n)$ time,*
- *may return the cc-identifier of any vertex in $g(\log^2\log^2 n)+O(1)$ time.*

Proof. We apply Lemma 3.4.1 to the assumed algorithm and obtain an algorithm with total update time $f_1(n) + O(|V_e| \log n)$, where $f_1(n) = O(n + n \cdot f(\log^2 n)/\log^2 n)$ and query time $g_1(n) = g(\log^2 n) + O(1)$. Then, we apply

the lemma to the obtained algorithm and get a new algorithm, whose total update time is

$$\begin{aligned} &O(n + |V_e| \log n + n \cdot f_1(\log^2 n)/\log^2 n) = \\ &\quad = O(n + |V_e| \log n + n(\log^2 n + \log^2 n \cdot f(\log^2 \log^2 n)/\log^2 \log^2 n)/\log^2 n) \\ &\quad = O(n + |V_e| \log n + n \cdot f(\log^2 \log^2 n)/\log^2 \log^2 n). \end{aligned}$$

It answers queries in $g(\log^2 \log^2 n) + O(1)$ time. □

We may now apply Lemma 3.4.2 to the simple $O(n \log n)$ algorithm (see Lemma 3.2.2) to obtain the following.

Lemma 3.4.3. *There exists a decremental connectivity algorithm, which processes any sequence of updates in $O(n \log \log \log n)$ time.*

Proof. We have $f(n) = O(n \log n)$ and $g(n) = O(1)$. Thus, $f(\log^2 \log^2 n) = O((\log^2 \log^2 n) \log(\log^2 \log^2 n)) = O((\log^2 \log^2 n) \log \log \log n)$. Thus, the total update time is $O(n \log \log \log n)$, and the query time is constant. □

3.5 $O(n)$ Time Algorithm

In this section we finally show an algorithm that runs in $O(n)$ time. Observe that in Lemma 3.4.2, we run the assumed decremental algorithm on graphs of size $\log^2 \log^2 n$. However, the number of all such graphs is so small, that we may precompute all necessary connectivity information for all of them.

Lemma 3.5.1. *Let w be the word size and $\log n \leqslant w$. After preprocessing in $o(n)$ time, we may repeatedly initialize and run algorithms for decremental maintenance of connected components in graphs of size $t = O(\log^2 \log n)$. These algorithms may be given a set of vertices V_e, and maintain the cc-identifiers of vertices of V_e explicitly. An algorithm for a graph of size t runs in $O(t + |V_e| \log t)$ time and may return the cc-identifier of every vertex in $O(1)$ time.*

Proof. As in the previous sections, we say that V_e is an explicit set. The state of the algorithm is uniquely described by the current set of edges in the graph and the explicit set. There are $2^{t(t-1)/2}$ labeled undirected graphs on t vertices (including non-planar graphs) and $O(2^t)$ possible explicit sets. Thus, there are $O(2^{t^2})$ possible states, which, for $t = O(\log^2 \log n)$ gives $2^{O(\log^4 \log n)} = 2^{o(\log n)} = o(n)$. In particular, each state can be encoded as a binary string of length $O(\log^4 \log n)$ which fits in a single machine word.

For each state, we precompute the cc-identifiers. Moreover, for each pair of state and an edge to be deleted, we compute the changes to the cc-identifiers of vertices in the explicit set. Observe that if the edge deletion is critical, we simply need to compute the set of vertices in the smaller out of the two connected components that are created and store the intersection of this set and V_e. These vertices should be assigned new, unique cc-identifiers.

We encode the graph by a binary word of length $O(\log^4 \log n)$, where each bit represents an edge between some pair of vertices. Thus, when an edge is deleted, we may compute the new state of the algorithm in constant time by switching off a single bit. For any planar graph and any sequence of deletions, the total number of changes of cc-identifiers of vertices of V_e is $O(|V_e| \log t)$ (using the analysis similar to the one from the proof of Lemma 3.2.2). The query time is constant, since the cc-identifiers are precomputed. For each of the $2^{O(\log^4 \log n)}$ states, we require $O(\log^4 \log n)$ preprocessing time. Thus, the preprocessing time is $o(n)$. □

We may now apply Lemma 3.4.2 to the algorithm of Lemma 3.5.1 to obtain the main result of this chapter.

Theorem 3.5.2. *There exists a decremental connectivity algorithm for planar graphs that supports updates in $O(n)$ total time and answers queries in constant time.*

Chapter 4

Dynamic Steiner Tree

In this chapter we consider the dynamic Steiner tree problem. We are given a graph $G = (V, E, d_G)$ with positive edge weights $d_G : E \to \mathbb{R}_+$. The goal is to maintain information about constant approximate Steiner tree in G for a dynamically changing set $S \subseteq V$ of terminals.

Our construction is based on the notion of *bipartite emulator*. The bipartite emulator of a graph G is a low-degree bipartite graph, which can be used to approximate distances in G. Moreover, it has some additional properties which assure that, roughly speaking, maintaining the MST of some subgraph of the emulator corresponds to maintaining an approximate Steiner tree in G. Once we know the (approximate) distances in G, we use Lemma 2.3.1 to approximate the Steiner tree.

The algorithm we give has a modular construction. In Section 4.1, we show how to maintain a Steiner tree spanning a dynamic set of vertices in a graph G, given a bipartite emulator of G. Then, in Section 4.2, we present the constructions of bipartite emulators for general and planar graphs. Since these two bipartite emulators have distinct characteristics (maximum degree and approximation ratio) we obtain two different algorithms for planar and general graphs. Finally, in Section 4.3 we mention other algorithms for dynamic Steiner tree that where given in [31], but are not described in this thesis.

4.1 Bipartite Emulators

We introduce the notion of *bipartite emulator*, which is essential for our dynamic Steiner tree algorithms, and show how to use it to maintain a good approximation of a Steiner tree in G.

Definition 4.1.1. *Let $G = (V, E, d_G)$ be a graph and $\alpha \geqslant 1$. A* bipartite

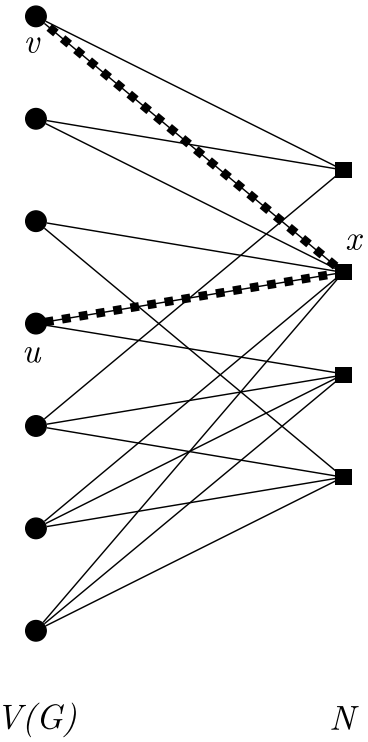

Figure 4.1: An illustration of a bipartite emulator. The thick dashed path corresponds to the distance $\widetilde{\delta}_B(u,v) = d_B(ux) + d_B(xv)$.

emulator *of G is a bipartite graph $B = (V \cup N, E_B, d_B, p)$ that satisfies the following properties:*

- *for every $u, w \in V$, there exists a vertex $x \in N$ in B, such that $d_B(ux) + d_B(xw) \leqslant \alpha \cdot \delta_G(u, w)$,*
- *p is a function $p : N \to V$, such that for every edge $uw \in E_B$, where $u \in V, w \in N$, there is a walk in G from u to $p(w)$ of length $d_B(uw)$.*

We say that α is the stretch *of emulator B.*

Let us first discuss the definition. For illustration, see Figure 4.1. We later formally prove the observations we make here. The bipartite emulator is a graph over the set of vertices of G, and the set of auxiliary vertices, denoted N. The first condition states, that the distance between vertices u and w in G can be approximated with a two-edge path connecting u and w in B. The second condition says that the auxiliary vertices actually correspond to vertices of G, and each edge of the emulator corresponds to a walk, which exists in G. This implies that the bipartite emulator cannot underestimate the distances. Throughout this section we consider a graph G and its bipartite emulator B of stretch α.

Let T be a subgraph of B. Since every edge uw of the bipartite emulator corresponds to a path from u to $p(w)$ in G, we can construct a subgraph T' of G by adding to T' paths corresponding to edges of T. We say that T *maps to* T'.

Proposition 4.1.2. *Let $u, w \in V$ and let P be a path connecting u and w in B. Then P maps to a walk in G, which connects u and w and has weight $d_B(P)$.*

We show that B can be used to obtain approximate distances in G.

Proposition 4.1.3. *For every $u, w \in V$, $\delta_G(u, w) \leqslant \delta_B(u, w) \leqslant \alpha \cdot \delta_G(u, w)$,*

Proof. The first inequality follows directly from Proposition 4.1.2. By the first property of a bipartite emulator, there exists $x \in N$, such that $d_B(ux) + d_B(xw) \leqslant \alpha \cdot \delta_G(u, w)$. Thus, $\delta_B(u, w) \leqslant \alpha \cdot \delta_G(u, w)$. □

Our goal is to maintain some tree in the bipartite emulator that spans the set of terminals. The following lemma says that a tree in B that spans $S \subseteq V$ corresponds to a tree spanning S in G.

Lemma 4.1.4. *Let T be a tree in B that spans (a superset of) a set $S \subseteq V$. Then, T maps to a subgraph T' in G, such that T' that spans S, and $d_G(T') \leqslant d_B(T)$.*

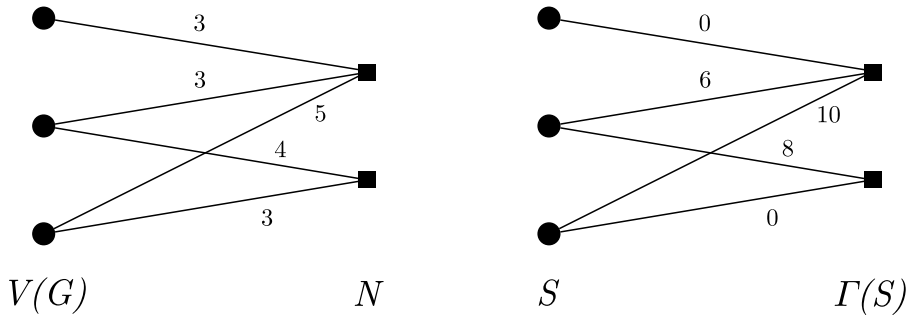

Figure 4.2: An example bipartite emulator of a graph $G = (V, E, d_G)$ (on the left) and its corresponding graph B_S for $S = V(G)$ (on the right).

Proof. Fix $u, w \in S$. Every edge xy of T maps to a walk in G of weight $d_B(xy)$. Since T' is a union of all edges of these walks, $d_G(T') \leqslant d_B(T)$. Moreover, by Proposition 4.1.2, every path P in B connecting u and w maps to a walk between u and w in G. Thus, T' spans S. □

We define $\widetilde{G} = (V, \binom{V}{2}, \widetilde{\delta}_B)$, where $\widetilde{\delta}(u, w) = \min_{x \in N} d_B(ux) + d_B(xw)$ to be the approximation of the metric closure of G given by the bipartite emulator. Observe that $\widetilde{G}$ is an α-approximation of $\overline{G}$, that is:

Proposition 4.1.5. *For every $u, w \in V$, $\delta_G(u, w) \leqslant \widetilde{\delta}_B(u, w) \leqslant \alpha \cdot \delta_G(u, w)$.*

This immediately gives the following.

Proposition 4.1.6. *For every $S \subseteq V$, $\delta_G(MST(\overline{G}[S])) \leqslant \widetilde{\delta}_B(MST(\widetilde{G}[S])) \leqslant \alpha \cdot \delta_G(MST(\overline{G}[S]))$.*

Let $S \subseteq V$ be a set of terminals. In order to approximate $ST(G, S)$, we use Lemma 2.3.1 and approximate $MST(\overline{G}[S])$. By Proposition 4.1.6, this can be achieved by approximating $MST(\widetilde{G}[S])$. However, maintaining $\widetilde{G}[S]$ under changes to S would require too much time. Instead of that, we use an auxiliary graph based on B and maintain its MST.

We define a graph $B_S = (V_S, E_S, d_{B_S})$ as follows (see Figure 4.2). Let $\Gamma(S) \subseteq N$ be the neighborhood of S in B. The edge and vertex set of B_S is the same as in $B[S \cup \Gamma(S)]$. Formally, $V_S = S \cup \Gamma(S)$, $E_S = \{uw \in E \mid u \in S, w \in \Gamma(S)\}$. The edge weights in B_S, however, are different from the corresponding weights in B. Fix $v \in N$. Let $vw_1, \ldots, vw_k$ be the edges incident to v in B sorted in nondecreasing order of weight, that is $d_B(vw_1) \leqslant \ldots \leqslant d_B(vw_k)$. Set $d_{B_S}(vw_1) = 0$ and $d_{B_S}(vw_i) = 2d_B(vw_i)$ for $i = 2, \ldots, k$.

Let us now provide some intuition behind this construction. As we later show, this choice of edge weights assures that the distances between vertices of S in B_S are at most twice the distances in B, and at the same time, they are not smaller than the distances in B. Moreover, in B_S, every vertex of $\Gamma(S)$ is connected to a vertex of S with an edge of weight 0. This means that any tree spanning S in B_S can be extended to a spanning tree of B_S of the same weight. Thus, $MST(B_S)$ and $ST(B_S, S)$ have the same weight, and because the distances in B_S approximate distances in G, we may use $MST(B_S)$ to approximate $ST(G, S)$.

In our algorithm we maintain the MST of B_S. Our entire construction can be described by the following sequence of trees. Each object in the sequence is used to approximate the previous one: Steiner tree in G, $MST(\overline{G}[S])$, $MST(\widetilde{G}[S])$, tree spanning S in B, $MST(B_S)$. We first show how to relate weight of $MST(B_S)$ to a weight of a tree spanning S in B.

Lemma 4.1.7. *Let T be a tree in B_S, which does not contain any leaves in $\Gamma(S)$. Then $d_B(T) \leqslant d_{B_S}(T)$.*

Proof. Let us group the edges of T by their endpoints in N and show the inequality for each group. Fix $v \in \Gamma(S)$, such that v belongs to T. Let $vw_1, \ldots, vw_k$ be the edges of T incident to v, ordered such that $d_{B_S}(vw_1) \leqslant \ldots \leqslant d_{B_S}(vw_k)$. We assume that if T contains the edge that is assigned zero weight during the construction of G_S, it comes first in this order. Thus, $d_{B_S}(vw_i) = 2d_G(vw_i)$ for all $2 \leqslant i \leqslant k$. In addition, since no vertex of N is a leaf, $k \geqslant 2$. Hence,

$$\begin{aligned} d_{B_S}(vw_1) + \ldots + d_{B_S}(vw_k) &\geqslant d_{B_S}(vw_2) + \ldots + d_{B_S}(vw_k) \\ &= 2(d_G(vw_2) + \ldots + d_G(vw_k)) \\ &\geqslant (d_G(vw_1) + d_G(vw_2)) + 2(d_G(vw_3) + \ldots + d_G(vw_k)) \\ &\geqslant d_G(vw_1) + \ldots + d_G(vw_k). \end{aligned}$$

□

Lemma 4.1.8. $d_{B_S}(MST(B_S)) \leqslant 2\widetilde{\delta}_B(MST(\widetilde{G}[S]))$.

Proof. We map each edge of $MST(\widetilde{G}[S])$ to a corresponding two-edge path in B_S, thus obtaining a tree T_n. Note that $d_B(T_n) \leqslant \widetilde{\delta}_B(MST(\widetilde{G}[S]))$, as multiple paths can contain the same edge of B. Observe that vertices of T_n are contained in $S \cup \Gamma(S)$, so T_n is also a tree in B_S. Since for any $uw \in E(B_S)$, $d_{B_S}(uw) \leqslant 2d_B(uw)$, we have that $d_{B_S}(T_n) \leqslant 2d_B(T_n) \leqslant 2\widetilde{\delta}_B(MST(\widetilde{G}[S]))$. Moreover, every $v \in N$ in B_S is connected to a vertex of S with a zero-weight edge, so by adding zero-weight edges to T_n, we can obtain a tree T such that $d_{B_S}(T) = d_{B_S}(T_n)$ and T is a spanning tree of B_S. We have $d_{B_S}(MST(B_S)) \leqslant d_{B_S}(T) = d_{B_S}(T_n) \leqslant 2\widetilde{\delta}_B(MST(\widetilde{G}[S]))$. □

The following lemma shows that the MST of B_S can be used to approximate $ST(G, S)$. The weight of $MST(B_S)$ is low and it maps to a tree spanning S in G of weight at most $d_{B_S}(MST(B_S))$.

Lemma 4.1.9. *Let T be a tree obtained from $MST(B_S)$ by removing all leaves that are vertices of $\Gamma(S)$. Then, T maps to a subgraph T' spanning S in G, and $d_G(T') \leqslant d_{B_S}(MST(B_S)) \leqslant 4\alpha \cdot d_G(ST(G, S))$.*

Proof. Let $v \in \Gamma(S)$ be a leaf of $MST(B_S)$. Since every vertex of $\Gamma(S)$ is connected to a vertex of S with an edge of weight 0, the weight of the only edge incident to v is also 0. Thus, $d_{B_S}(T) = d_{B_S}(MST(B_S))$. Clearly, T is a tree that spans S in B_S and in B. By Lemma 4.1.4, T maps to a subgraph T' of G spanning S. We have

$$\begin{aligned} d_{B_S}(MST(B_S)) &= d_{B_S}(T) \\ &\geqslant d_B(T) && \text{by Lemma 4.1.7} \\ &\geqslant d_G(T') && \text{by Lemma 4.1.4} \end{aligned}$$

It remains to show the last inequality:

$$\begin{aligned} d_{B_S}(MST(B_S)) &\leqslant 2\widetilde{\delta}_B(MST(\widetilde{G}[S])) && \text{by Lemma 4.1.8} \\ &\leqslant 2\alpha \cdot \delta_G(MST(\overline{G}[S])) && \text{by Proposition 4.1.6} \\ &\leqslant 4\alpha \cdot d_G(ST(G, S)) && \text{by Lemma 2.3.1} \end{aligned}$$

□

Lemma 4.1.10. *Let $G = (V, E, d_G)$ be a graph and let $B = (V \cup N, E_B, d_B, p)$ be its bipartite emulator with stretch α. Denote by Δ the maximum degree of a vertex from V in B. Let S be a set subject to insertions and deletions, such that at any time $S \subseteq V$. Then, we can maintain a 4α-approximate Steiner tree of G that spans S, handling each update to S in $O(\Delta \log^4 n)$ time.*

Proof. From Lemma 4.1.9 it follows that it suffices to maintain $MST(B_S)$. In order to do that, as the elements are inserted to or removed from S, we maintain the graph B_S and run the decremental MSF algorithm (see Theorem 2.3.2) on top of it. This algorithm requires $O(\log^4 n)$ amortized time for any edge addition or removal.

Recall that the edge and vertex sets of B_S are the same as in $B[S \cup \Gamma(S)]$. Thus, each time a vertex is added to/removed from S, we need to

insert/remove a single vertex from B_S and all its incident edges. For simplicity, instead of adding/removing a vertex, we may add/remove all its incident edges.

It remains to describe how to maintain the edge weights of B_S. Recall that for each $v \in N$, the edge incident to v in B_S that has the smallest weight in B, has weight 0 in B_S. Other edges incident to v have weight which is two times bigger than their weight in B. It is easy to see that adding/removing a single edge to B_S may trigger one edge weight update. In order to detect these updates, for every $v \in N$ we maintain a heap containing the weights of all its incident edges.

It follows that handling an update to S requires $O(\Delta)$ edge updates in the graph B_S that we maintain. Thus, updating $MST(G_S)$ requires $O(\Delta \log^4 n)$ amortized time. We also need $O(\log n)$ per each change to detect edge weight updates, but this time is dominated by the time needed to maintain the MST. □

4.2 Constructing Bipartite Emulators

In this section we show how to construct emulators that can be plugged into Lemma 4.1.10 to obtain dynamic algorithms for maintaining the Steiner tree.

4.2.1 General Graphs

Our bipartite emulator for general graphs is based on an approximate distance oracle by Thorup and Zwick [44]. Let $G = (V, E, d_G)$ be an undirected, weighted graph, and $k \geqslant 1$ be an integer. The algorithm first constructs a sequence of sets $A_0 \supseteq A_1 \supseteq \ldots \supseteq A_{k-1} \supseteq A_k$. We set $A_0 = V$. For $1 \leqslant i < k$, A_i is obtained by taking each element of A_i independently with probability $n^{-1/k}$, and $A_k = \emptyset$. We assume that $A_{k-1} \neq \emptyset$. This happens with big probability, so we can assure it by repeating the sampling, if necessary.

For a $v \in V$, we define a *bunch* of v to be a set $\mathcal{B}(v) = \bigcup_{i=0}^{k-1}\{w \in A_i \setminus A_{i+1} \mid \delta_G(w, v) < \delta_G(A_{i+1}, v)\}$. Here, $\delta_G(A_{i+1}, v)$ denotes the distance between v and the nearest vertex in A_{i+1}. Moreover, for $v \in V$, and $0 \leqslant i < k$, we define $p_i(v)$ to be the vertex in A_i, which is nearest to v.

The initialization of the oracle consists in computing:

- the sets $A_0, \ldots, A_k$,
- the bunch $\mathcal{B}(v)$ for every $v \in V$,
- $p_i(v)$ for every $v \in V$ and $0 \leqslant i < k$,

- the distance $\delta_G(v, p_i(v))$, for every $v \in V$ and $0 \leqslant i < k$,
- the distance $\delta_G(u, w)$ for every $w \in \mathcal{B}(u)$.

Let us now list some properties of the oracle that we use.

Lemma 4.2.1 ([44]). *Let $G = (V, E, d)$ be a graph, $n = |V|$, $m = |E|$, and $k \geqslant 1$ be an integer. Then, the approximate distance oracle can be initialized in $O(kmn^{1/k})$ expected time.*

Lemma 4.2.2 ([44]). *Let $v_1, v_2 \in V$. For some $s \in \{1, 2\}$, there exists a vertex $x \in \mathcal{B}(v_s)$, such that $x = p_j(v_{3-s})$ for some $0 \leqslant j < k$ and $\delta_G(v_1, x) + \delta_G(x, v_2) \leqslant (2k-1)\delta_G(v_1, v_2)$.*

Lemma 4.2.3 ([44]). *For every $v \in V$, the expected size of $\mathcal{B}(v)$ is $O(kn^{1/k})$.*

Observe that the distances $\delta_G(v_1, x)$ and $\delta_G(x, v_2)$ are computed during the initialization of the oracle.

Lemma 4.2.4. *Let $G = (V, E, d)$ be a graph, $n = |V|$, $m = |E|$, and $k \geqslant 1$ be an integer. We can compute a bipartite emulator $B = (V \cup N, E_B, d_B, p)$ of G of stretch $2k-1$ in $O(kmn^{1/k})$ expected time. The degree of every vertex of V in B is $O(n^{1/k})$.*

Proof. By Lemma 4.2.1, we build a TZ oracle for G in $O(kmn^{1/k})$ expected time. Let V' be a copy of V. For each vertex $v \in V$, V' contains its copy v'. We set the vertex set of B to be $V \cup V'$.

Now, for every u, w, such that $u \in V$ and $w \in \mathcal{B}(u)$, we add to B an edge uw' of weight $\delta_G(u, w)$. Moreover, for every $u \in V$ and $0 \leqslant i < k$, we add to B an edge $up_i(u)'$ of weight $\delta_G(u, p_i(u))$. Note that the distances we need are computed during the initialization of the oracle.

We now prove that we obtain a bipartite emulator. Consider $u, w \in V$. By Lemma 4.2.2, there exists $x \in \mathcal{B}(w)$, such that (possibly after we swap u and w) $x = p_j(u)$ for some $0 \leqslant j < k$ and $\delta_G(u, x) + \delta_G(x, w) \leqslant (2k-1)\delta_G(u, w)$. Observe that B contains edges ux' and wx' of lengths $\delta_G(u, x)$ and $\delta_G(w, x)$, respectively. Thus, there is a two-edge path in B between u and w of length at most $(2k-1)\delta_G(u, w)$. It is easy to see that for every $v' \in V'$, we set $p(v') = v$, to obtain the desired mapping between V' and V.

The expected degree of every $v \in V$ in B is $O(|\mathcal{B}(v)| + k)$, which, by Lemma 4.2.3 is $O(kn^{1/k})$ The initialization time is clearly dominated by the time needed to construct the TZ oracle. □

By combining the above Lemma with Lemma 4.1.10 we obtain the following result.

Theorem 4.2.5. *Let $G = (V, E, d)$ be a graph, $n = |V|$, $m = |E|$ and $k \geqslant 1$ be an integer. Let $S \subseteq V$ be a dynamic set, subject to vertex insertions and removals (initially $S = \emptyset$). Then, after preprocessing in $O(kmn^{1/k})$ expected time, we may maintain a $(8k - 4)$-approximate Steiner tree that spans S, handling each update to S in $O(kn^{1/k} \log^4 n)$ expected amortized time.*

4.2.2 Planar Graphs

In this section we show a construction of a bipartite emulator for planar graphs. As a result we obtain an algorithm which maintains a $(4 + \varepsilon)$-approximate Steiner tree in polylogarithmic time per update. In order to reach this goal, we use a construction by Thorup (Section 3.8 in [43]) that we extend in order to construct a bipartite emulator (see Lemma 4.2.7).

Let $G = (V, E, d_G)$ be an undirected weighted planar graph. The overall idea uses recursive division of G using balanced separators. We find a balanced separator of G that consists of a constant number of shortest paths $P_1, \ldots, P_k$ (the separator consists of vertices contained in these paths). For a shortest path P_i, we build an emulator that approximates all the shortest paths in G that intersect P_i. Then, we recurse on each of the connected components of $G \setminus (P_1 \cup \ldots \cup P_k)$. Hence, we now focus on the following problem. Given a planar graph G and a shortest path P, build an emulator that approximates all shortest paths intersecting P.

We define a *connection* to be an edge that connects a vertex $v \in V$ with a vertex $a \in P$ and has length $d_G(va)$, which is at least $\delta_G(v, a)$ (it would be convenient to assume that $d_G(va) = \delta_G(v, a)$, but the algorithm we use may sometimes give longer connections). A connection vb *ε-covers* x if $d_G(vb) + \delta_G(b, x) \leqslant (1 + \varepsilon)\delta_G(v, x)$. Observe that the distance $\delta_G(b, x)$ can be measured along the path P. A set of connections $C(v, P)$ between $v \in V$ and P is *ε-covering* if it ε-covers every $x \in P$.

Lemma 4.2.6. *Let $G = (V, E, d_G)$ be a planar graph, $n = |V|$, and $0 < \varepsilon \leqslant 1$. Let P be a shortest path in G. For each $v \in V(G)$ we can construct an ε-covering set $C(v, P)$ of size $O(\varepsilon^{-1})$ in $O(\varepsilon^{-1} n \log n)$ total time.*

A very similar fact is shown in [43], but the definition of ε-covering used there is slightly different, so, for the sake of completeness, we rewrite the proof.

Proof. Let $\varepsilon_0 = \varepsilon/2$. We say that a connection vb *strongly-ε-covers* a if $d_G(vb) + (1 + \varepsilon)\delta_G(b, a) \leqslant (1 + \varepsilon)\delta_G(v, a)$. By Lemma 3.18 in [43], for each $v \in V(G)$ we can construct a strongly-$(\varepsilon_0/2)$-covering set $D(v, P)$ of size

$O(\varepsilon_0^{-1} \log n)$ in $O(\varepsilon_0^{-1} n \log n)$ time.[1] We now show that we can use it to construct an ε-covering set $C(v, P) \subseteq D(v, P)$ of size $O(\varepsilon^{-1})$.

Let vc be the shortest connection from $D(v, P)$ and s be one of the two endpoints of P. We add vc to $C(v, P)$. Now, iterate through connections in $D(v, P)$ starting from vc and going towards s. Let vb be the connection that was most recently added to $C(v, P)$. If for the current connection va we have $d_G(vb) + \delta_G(b, a) > (1 + \varepsilon_0) d_G(va)$, we add va to $C(v, P)$. Then, we run a similar procedure using the other endpoint of P.

To prove that $C(v, P)$ covers every vertex between c and s, consider some vertex $x \in P$. There exists a connection $va \in D(v, P)$ that strongly-$(\varepsilon_0/2)$-covers vx, so $d_G(va) + (1+\varepsilon_0/2)\delta_G(a, x) \leqslant (1+\varepsilon_0/2)\delta_G(v, x)$. If this connection is in $C(v, P)$ then vx is strongly-$\varepsilon_0/2$-covered and obviously also ε-covered. Otherwise, there exists a connection vb such that $d_G(vb) + \delta_G(b, a) \leqslant (1 + \varepsilon_0) d_G(va)$. We have

$$\begin{aligned}
d_G(vb) + \delta_G(b, x) &\leqslant d_G(vb) + \delta_G(b, a) + \delta_G(a, x) \\
&\leqslant (1 + \varepsilon_0) d_G(va) + \delta_G(a, x) \\
&\leqslant (1 + \varepsilon_0) d_G(va) + (1 + \varepsilon_0)(1 + \varepsilon_0/2)\delta_G(a, x) \\
&= (1 + \varepsilon_0)(d_G(va) + (1 + \varepsilon_0/2)\delta_G(a, x)) \\
&\leqslant (1 + \varepsilon_0)(1 + \varepsilon_0/2)\delta_G(v, x) \\
&\leqslant (1 + \varepsilon)\delta_G(v, x).
\end{aligned}$$

The last inequality follows from $\varepsilon_0 = \varepsilon/2 \leqslant 1$. It remains to bound the size of $C(v, P)$. Let $f(vb) = d_G(vb) + \delta_G(b, s)$. As connections are added to $C(v, P)$ we trace the value of $f(vb)$, where vb is the last connection that we have added. Every time we add a connection va we reduce the value of f by

$$\begin{aligned}
f(vb) - f(va) &= d_G(vb) + \delta_G(b, s) - d_G(va) - \delta_G(a, s) \\
&= d_G(vb) - d_G(va) + \delta_G(b, a) \\
&> \varepsilon_0 d_G(va) \\
&\geqslant \varepsilon_0 \delta_G(v, c).
\end{aligned}$$

However, the total change equals

$$\begin{aligned}
f(vc) - f(vs) &= d_G(vc) + \delta_G(c, s) - d_G(vs) \\
&\leqslant (1 + \varepsilon_0)\delta_G(v, c) + \delta_G(c, s) - \delta_G(v, s) \\
&\leqslant (2 + \varepsilon_0)\delta_G(v, c).
\end{aligned}$$

[1] Strictly speaking, the strongly-ε-covering set according to our definition is an $\varepsilon/(\varepsilon+1)$-covering set according to the definition used in the statement of Lemma 3.18 of [43].

Thus, at most $O(\varepsilon_0^{-1}) = O(\varepsilon^{-1})$ connections are added to $C(v, P)$.

The same procedure is then repeated for the other endpoint of P, so we get a total of $O(\varepsilon^{-1})$ connections. □

Lemma 4.2.7. *Let $G = (V, E, d_G)$ be a planar graph, $n = |V|$, $0 < \varepsilon \leqslant 1$. Let P be a shortest path in G. For each $v \in V$ we can construct a set of connections $C'(v, P)$ of size $O(\varepsilon^{-1} \log n)$, which satisfies the following property. For any two vertices $u, w \in V$, if the shortest path between u and w intersects P, then for some $x \in P$ there exist connections $ux \in C'(u, P)$ and $wx \in C'(w, P)$, such that $\delta(u, w) \leqslant d_G(ux) + d_G(wx) \leqslant (1 + \varepsilon)\delta(u, w)$. The sets $C'(v, P)$ can be constructed in $O(\varepsilon^{-1} n \log n)$ time.*

Proof. First, using Lemma 4.2.6, for every $v \in V$ we construct an ε-covering sets of connections $C(v, P)$. Consider a shortest path Q between u and w, which intersects P in $x \in P$. There exists a path Q' between u and w which consists of a connection, subpath of P, denoted henceforth Q'_P, and another connection. Moreover, $d_G(Q') \leqslant (1+\varepsilon)d_G(Q)$. We call each path of this form an *approximating path*. Our goal is to substitute every approximating path with an approximating path that consists solely of two connections from $C'(v, P)$.

The construction is done recursively. The parameter of the recursion is a subpath P' of P. Consider a single step, with a parameter $P' = p_1 p_2 \dots p_k$. Let $p_m = p_{\lfloor k/2 \rfloor}$ be the middle vertex of P'. For any $v \in V$ and $p_i \in P'$, if there is an connection $vp_i \in C(v, P)$, we add a connection vp_m of length $d_G(vp_i) + \delta(p_i, p_m)$ to $C'(v, P)$. Then, we recurse on $P_1 = p_1 p_2 \dots p_{m-1}$ and $P_2 = p_{m+1} \dots p_k$. Lastly, for each $p_i \in P$ we add a connection $p_i p_i$ of length 0.

To prove the correctness of the procedure, consider now the aforementioned approximating path Q', and recall that $Q'_P = P \cap Q'$. Let p be the vertex that is taken as p_m in the closest to root node in the recursion tree of the algorithm, among all vertices of Q'_P. Observe that in the single recursive step when $p_m = p$, we add to $C'(v, P)$ the connections up and wp of length exactly equal to the length of the part of Q' between u and p, and the part of Q' between p and w, respectively. Also, the connections we add clearly do not cause any distances to be underestimated.

The running time of each step is proportional to the length of the subpath we consider and the number of connections incident to this subpath. Moreover, every connection may be considered in at most $O(\log n)$ recursive calls, so we we add to $C'(v, P)$ at most $O(\varepsilon^{-1} \log n)$ connections. It follows that the total running time of the procedure is $O(\varepsilon^{-1} n \log n)$. □

Lemma 4.2.8. *Let $G = (V, E, d)$ be a planar graph, $n = |V|$, $0 < \varepsilon \leqslant 1$. We can construct a bipartite emulator $B = (V \cup N, E_B, d_B, p)$ of G of stretch $1 + \varepsilon$. The degree of every $v \in V$ in B is $O(\varepsilon^{-1} \log^2 n)$. The graph B can be constructed in $O(\varepsilon^{-1} n \log^2 n)$ time.*

Proof. We begin with B being a graph with vertex set V and no edges. The construction is done recursively. As it is shown, e.g., in [43], each planar graph admits a balanced separator that consists of a constant number of shortest paths $P_1, \ldots, P_k$, and, moreover, such a separator can be found in $O(n)$ time. For each path P_i we use Lemma 4.2.7 to construct a set of connections $C'(v, P_i)$ for every $v \in V$. Next, we iterate through the vertices of the paths P_i. For each vertex $w \in P_i$ we add a new auxiliary vertex w' to B and add an edge uw' for each connection uw from G (the length of the edge is the length of the connection). After that, we recurse on each connected component of $G \setminus (P_1 \cup \ldots \cup P_k)$.

Let us now prove the correctness of the construction. Consider any two $v_1, v_2 \in V$ and the shortest path Q between them. At some step of the recursion, some vertex of Q belongs to the separator that we have found. From the construction, it follows that in this step we have added to B a vertex w' and edges v_1w' and v_2w' to B of total length at most $(1 + \varepsilon)\delta_G(v_1, v_2)$. It remains to prove that B has the second property of the bipartite emulator. It follows from the fact that each time we add a vertex w' and connection uw', w' corresponds to some vertex of V and the connection has length equal to the length of some walk between u and the corresponding vertex of w'.

Since every vertex $v \in V$ takes part in $O(\log n)$ recursive steps and in every step we add $O(\varepsilon^{-1} \log n)$ edges incident to any $v \in V$, we have that the degree of any vertex of V in B is $O(\varepsilon^{-1} \log^2 n)$. As shown in [43], finding the separators requires $O(n \log n)$ total time. The running time of every recursive step is dominated by the time from Lemma 4.2.7. Summing this over all recursive steps, we get that the construction can be done in $O(\varepsilon^{-1} n \log^2 n)$ time. □

By constructing B according to Lemma 4.2.8 and applying Lemma 4.1.10 we obtain the following.

Theorem 4.2.9. *Let $G = (V, E, d)$ be a planar graph and $0 < \varepsilon \leqslant 1$. Let $S \subseteq V$ be a dynamic set, subject to vertex insertions and removals (initially $S = \emptyset$). Then, after preprocessing in $O(\varepsilon^{-1} n \log^2 n)$ time, we may maintain a $(4 + \varepsilon)$-approximate Steiner tree that spans S, handling each update to S in $O(\varepsilon^{-1} \log^6 n)$ amortized time.*

4.3 Related Results

In the paper by Łącki et al. [31], in addition to the algorithm for dynamic Steiner tree, which we present in this chapter, other results are obtained. Namely, the paper shows that it is possible to obtain better approximation ratios, if we allow higher running time. In particular, it presents a $(6+\varepsilon)$-approximate algorithm, which for a weighted graph $G = (V, E, d_G)$ processes updates in $\tilde{O}(\varepsilon^{-5}\sqrt{n}\log D)$ amortized time. Here, D is the *stretch* of $\overline{G}$, that is the ratio between the weights of edges with maximal and minimal weights. Moreover, for planar graphs, two $(2+\varepsilon)$-approximate algorithms are shown. The first one processes updates in $\tilde{O}(\varepsilon^{-5.5}\sqrt{n}\log D)$ amortized time. The second one handles each update in $\tilde{O}(\varepsilon^{-2}\log^3 n \log D)$ amortized time, but it only supports adding terminals. While this approximation ratios are two times smaller than the ones we have obtained in this chapter, they are still far from what can be computed in polynomial time in static case. In general graphs, a 1.39-approximate algorithm is known [8], whereas for planar graphs there exists a PTAS [7]. We believe that there exist dynamic algorithms with lower approximation ratios, and designing them is an interesting open problem.

Chapter 5

Connectivity in Graph Timelines

In this section we consider dynamic connectivity in undirected graphs in a semi-offline model. We develop algorithms that process a *graph timeline*, that is a sequence of graphs $G_1, \ldots, G_t$, such that G_{i+1} is obtained from G_i by adding or removing a single edge. In this model, an algorithm may preprocess the entire timeline at the beginning, and after that it should answer queries arriving in online fashion. We consider timelines of undirected graphs and two types of queries. Moreover, in order to obtain simpler running time bounds we assume $t = O(n^c)$.

An `exists`(u, w, a, b) query, where u and w are vertices and $1 \leqslant a \leqslant b \leqslant t$, asks whether vertices u and v are connected in *any* of $G_a, G_{a+1}, \ldots, G_b$. We show an algorithm that after preprocessing in $O(m + nt)$ time may answer such queries in $O(1)$ time.

We also consider a `forall`(u, w, a, b) query, which asks whether vertices u and v are connected in *all* graphs among $G_a, G_{a+1}, \ldots, G_b$. For this problem, we show an algorithm whose preprocessing time is $O(m + t \log t \log\log t)$ (m denotes the number of edges in G_1) and the query time is $O(\log n \log\log t)$. The algorithm is randomized and answers queries correctly with high probability.

In the following part of this chapter, we first introduce a data structure, which is common to the algorithms handling both types of queries (Section 5.1). In Section 5.2 we present an algorithm answering `exists` queries. Then, in Section 5.3 we deal with `forall` queries. Finally, in Section 5.4, we discuss the recent improvements to our algorithms that were made by Karczmarz [27].

5.1 Connectivity History Tree

In this section we introduce a data structure, which represents connectivity information of the entire timeline. We begin by introducing some basic notation related to graph timelines.

Let $G_1 = (V, E_1)$, $G_2 = (V, E_2)$ be two undirected graphs on the same vertex set. We define their intersection $G_1 \cap G_2$ to be a graph G' obtained by intersecting the sets of their edges, that is $G' = (V, E_1 \cap E_2)$. Let G^t be a graph timeline. For an interval[1] $[a, b]$, we define $G_{[a,b]} := G_a \cap G_{a+1} \cap \ldots \cap G_b$. We say that an edge is *permanent* if it is an edge of $G_{[1,t]}$, that is, it is present in every graph in the timeline. Other edges of the timeline are called *temporary*. If an edge connecting two vertices is removed from the timeline, and then added again, we consider these two edges different. This allows us to define a *lifetime* of an edge to be the maximal (w.r.t. inclusion) interval $[a, b]$, such that the edge is present in all graphs $G_a, \ldots, G_b$.

We assume that the input timeline is represented in $O(m + t)$ space, where m denotes the number of edges of G_1. The representation consists of the representation of G_1, and, for each $2 \leqslant i \leqslant t$, the information about the edge that is added or removed to obtain G_i.

Proposition 5.1.1. *There are at most m permanent edges and t temporary edges. We can compute the lists of permanent and temporary edges in $O(m+t)$ time.*

We first show that we may reduce our problems to the case when the graph timeline does not contain any permanent edges. The reduction takes only linear time.

Lemma 5.1.2. *Assume there exists an algorithm that, given a graph timeline G^t with no permanent edges, after preprocessing in $f(n, t)$ time answers* `exists` / `forall` *queries in $g(n, t)$ time, where $g(n, t)$ and $f(n, t)$ are nondecreasing in both parameters. Then, there exists an algorithm that, given a graph timeline that may contain permanent edges, after preprocessing in $f(\min(n, 2t), t) + O(m + t)$ time answers* `exists` / `forall` *queries in $g(\min(n, 2t), t) + O(1)$ time.*

Proof. We build a graph timeline $\bar{G}^t$, which does not contain permanent edges and can be used to answer queries regarding G^t. Observe that two vertices u and w connected with a path consisting of permanent edges are equivalent from the point of view of `exists` and `forall` queries. Thus, we can contract all permanent edges.

[1]Throughout this chapter we assume that the intervals are intervals of integers.

By Proposition 5.1.1, we can find all permanent edges in $O(m+t)$ time. Let V be the set of vertices of the graphs in the timeline. We build a graph H consisting a vertex set V and all permanent edges. Then, we compute connected components of H in $O(m+t)$ time. Denote by $cc[v]$ the connected component of a vertex $v \in V$.

Consider a connected component C of H. We say that C is *volatile*, if some temporary edge is incident to a vertex of C. Assume v is a vertex of a non-volatile component C. Then, v is connected to other vertices of C in all graphs in the timeline, and is not connected to the remaining vertices in every graph in the timeline. Thus answering queries about the timeline, where at least one of the parameters is a vertex of a non-volatile component is simple, if we store the non-volatile components of H

To answer the remaining queries, we build a new timeline $\bar{G}^t$. Each vertex of each graph of $\bar{G}^t$ is a volatile connected component of H. For every temporary edge uw of G^t, whose both endpoints belong to volatile components, we add an edge $cc[u]cc[w]$ to $\bar{G}^t$. In order to answer a query regarding two vertices u and w in G^t, that both belong to volatile components, we issue the same query about vertices $cc[u]$ and $cc[w]$ in $\bar{G}^t$.

It remains to show that $\bar{G}^t$ consists of graphs on at most $2t$ vertices. This follows from the fact that each volatile component contains an endpoint of a temporary edge, and the temporary edges have $2t$ endpoints in total. □

Hence, in the following part of this chapter, we assume that we work with a timeline with no permanent edges, and $t = \Omega(n)$. Both these properties can be assured in linear time, by applying Lemma 5.1.2.

We now define a data structure, which is used in the algorithms for both `exists` and `forall` queries. Throughout the following part of this chapter we heavily use the properties of segment trees (see Section 2.2) and disjoint-set data structures (see Section 2.3.3).

Definition 5.1.3. *Let t be a power of 2 and let G^t be graph timeline. A* connectivity history tree *(CHT) for G^t is a is a segment tree with t leaves. For each elementary interval $[a, b]$ of the segment tree, it contains a DSDS $C_{[a,b]}$ of the graph $G_{[a,b]}$.*

Figure 5.1 shows an example of graphs $G_{[a,b]}$ for all elementary intervals. In Figure 5.2, there is the corresponding CHT containing a DSDS $C_{[a,b]}$ for every elementary interval $[a, b]$. For an elementary interval $[a, b]$, we define $D_{[a,b]}$ to be the list of edges such that $G_{[a,b]}$ is obtained from $G_{\textsc{Parent}([a,b])}$ by adding edges of $D_{[a,b]}$. If $[a, b]$ has no parent, we let $D_{[a,b]}$ be the list of edges of $G_{[a,b]}$.

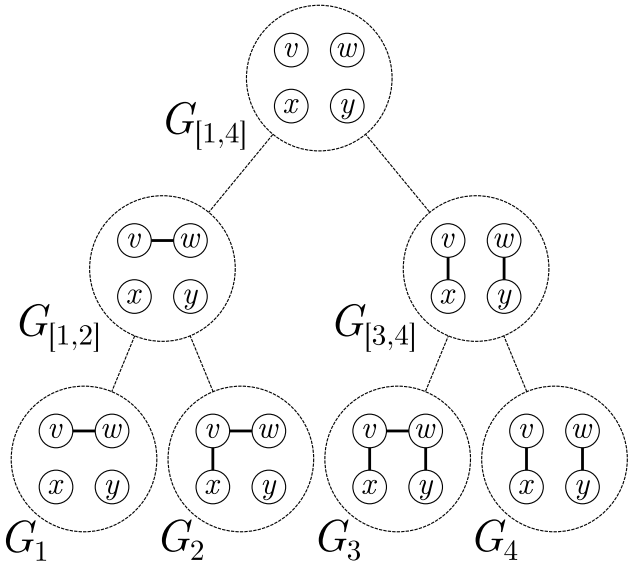

Figure 5.1: The timeline G_1, G_2, G_3, G_4 and the corresponding graphs $G_{[a,b]}$ for all elementary intervals $[a, b]$, arranged into a tree. Note that for $i = 1, 2, 3, 4$, $G_i = G_{[i,i]}$.

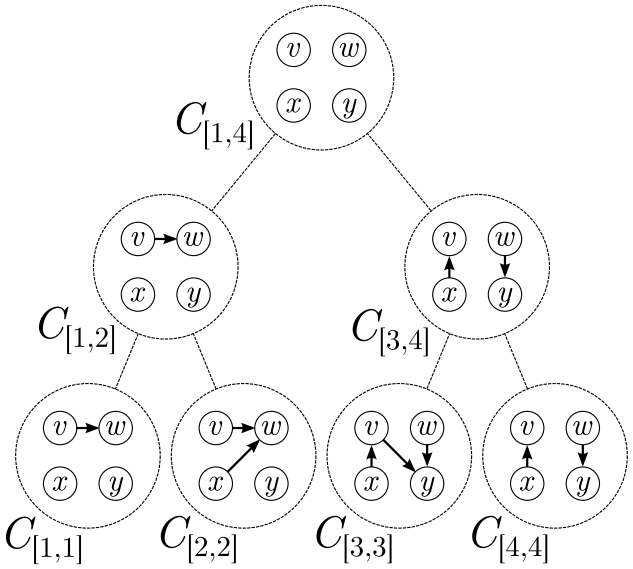

Figure 5.2: The CHT corresponding to the timeline from Figure 5.1. With each elementary interval we associate a DSDS. The arrows in each node point from a vertex to its parent in the DSDS. Parent pointers of the representatives have been omitted.

In the example from Figure 5.1, we have $D_{[1,2]} = vw$, $D_{[2,2]} = vx$, $D_{[3,4]} = vx, wy$, $D_{[3,3]} = vw$. The remaining lists, $D_{[1,4]}, D_{[1,1]}$ and $D_{[4,4]}$, are empty.

Lemma 5.1.4. *Let G^t be a graph timeline with no permanent edges. We can compute the lists $D_{[a,b]}$ for all elementary intervals $[a, b]$ in $O(t \log t)$ time. The total length of the lists is $O(t \log t)$.*

Proof. The algorithm is very simple. Consider a temporary edge e, whose lifetime is $[c, d]$. We use Lemma 2.2.8 to decompose $[c, d]$ into $k = O(\log t)$ elementary intervals $[a_1, b_1], \ldots, [a_k, b_k]$ and add e to $D_{[a_i,b_i]}$ for $1 \leqslant i \leqslant k$. Since the number of temporary edges is $O(t)$, this takes $O(t \log t)$ time and the total length of the resulting lists is $O(t \log t)$. It remains to show the correctness of our construction.

First, we need to show that every edge belonging to the list $D_{[a,b]}$ computed this way is an edge of $G_{[a,b]} = G_a \cap G_{a+1} \cap \ldots \cap G_b$. This follows directly from the construction. An edge is added to $D_{[a,b]}$ only if its lifetime is an interval $[c, d] \supseteq [a, b]$. Second, we need to show that every edge of $G_{[a,b]}$ is contained either in $D_{[a,b]}$ or $G_{\textsc{Parent}([a,b])}$. Consider an edge $e \in E(G_{[a,b]})$. Let the lifetime of e be $[c, d]$. Clearly $[a, b] \subseteq [c, d]$. By Lemma 2.2.9, the decomposition of $[c, d]$ into elementary intervals either contains $[a, b]$ or one of its ancestors. In the first case, e is added to $D_{[a,b]}$, whereas in the second case $\textsc{Parent}([a, b]) \subseteq [c, d]$, so $e \in G_{\textsc{Parent}([a,b])}$. The lemma follows. □

5.2 `exists` Queries

We now describe an algorithm for answering `exists` queries. We first compute an explicit representation of the CHT.

Lemma 5.2.1. *Let G^t be a graph timeline with no permanent edges. The CHT of G^t can be computed in $O(nt)$ time. It uses $O(nt)$ space.*

Proof. We first use Lemma 5.1.4 to compute lists $D_{[a,b]}$ for all elementary intervals $[a, b]$ in $O(t \log t)$ time. Then, we can compute the DSDSes $C_{[a,b]}$ in a top-down fashion. Since there are no permanent edges, $G_{[1,t]}$ consists solely of isolated vertices, so computing $C_{[1,t]}$ is trivial.

Then, we compute $C_{[a,b]}$ by first creating a copy of $C_{\textsc{Parent}([a,b])}$ and then adding edges of $D_{[a,b]}$ to it. It takes $O(n)$ time to create each of the $O(t)$ copies. Moreover, we add $O(t \log t)$ edges from lists $D_{[a,b]}$, each in time $O(\log n)$. This gives $O(t \log t \log n)$ total time, which is $O(nt)$, since we assume $t = O(n^c)$. The lemma follows. □

In order to answer queries, our algorithm precomputes, for all elementary intervals $[a, b]$, and all pairs of vertices u and v, the answer to an $\texttt{exists}(u, v, a, b)$ query. Thus, by Lemma 2.2.8, the answer to a query regarding an arbitrary interval $[p, q]$ can be computed in $O(\log t)$ time by decomposing $[p, q]$ into $O(\log t)$ elementary intervals and combining the answers for these intervals. However, as we later show, this can be improved to constant time.

For each elementary interval $[a, b]$, we compute a two-dimensional Boolean matrix $M_{[a,b]}$ which contains answers to $\texttt{exists}(u, v, a, b)$ queries for all pairs of vertices u, v. The matrix is indexed by vertices of the graph. We set $[M_{[a,b]}]_{u,v} = 1$ if and only if u and v are connected in *any* of $G_a, \ldots, G_b$. Additionally, for each node in the CHT, we define matrices $F_{[a,b]}$ and $L_{[a,b]}$. If $[M_{[a,b]}]_{u,v} = 1$, then $[F_{[a,b]}]_{u,v}$ is equal to the index of the first graph among $G_a, \ldots, G_b$, in which u and v are connected. Otherwise, $[F_{[a,b]}]_{u,v}$ is set to ∞. The matrix $L_{[a,b]}$ is defined similarly, but it contains indices of the last graph, in which the vertices are connected (thus, we use $-\infty$ instead of ∞).

We now show that these matrices can be computed and stored efficiently in a way that allows constant-time access to each of their cells.

Lemma 5.2.2. *Let G^t be a graph timeline with no permanent edges. We can compute implicit representations of the matrices $M_{[a,b]}$, $F_{[a,b]}$ and $L_{[a,b]}$ for all elementary intervals $[a, b]$ in $O(nt)$ time. The representations use $O(nt)$ space.*

Proof. For each elementary interval $[a, b]$, CHT contains $C_{[a,b]}$, which is a DSDS representing $G_{[a,b]}$. By Proposition 2.3.7, given a DSDS of an n-vertex graph G, we may compute its connected components in $O(n)$ time. This allows us to compute, for every elementary interval $[a, b]$, the connected components of $G_{[a,b]}$ in $O(nt)$ total time.

We now describe the implicit representation of matrices $M_{[a,b]}$. If $b-a > n$, $M_{[a,b]}$ is stored explicitly as a $n \times n$ matrix. However, if $b - a \leqslant n$, then the matrix has a more compact implicit representation, as some of its rows or columns are equal to each other. Consider a connected component C of $G_{[a,b]}$. Vertices belonging to C are connected in each of $G_a, G_{a+1}, \ldots, G_b$. Hence, it suffices only to store one row and column of $M_{[a,b]}$ for each connected component C.

Furthermore, within $G_a, \ldots, G_b$ at most $b-a$ edges are added or deleted. The endpoints of these edges are contained in at most $2(b - a) = O(b - a)$ components of $G_{[a,b]}$. We call these components *volatile*. Denote the number of volatile components by s and choose a single *representative* vertex from each of them, thus obtaining a list $v_1, \ldots, v_s$.

Thus, it suffices to store:

- an $s \times s$ submatrix of $M_{[a,b]}$ containing only rows and columns corresponding to $v_1, \ldots, v_s$,
- for each $v \in V$, the identifier of its connected component in $G_{[a,b]}$,
- for each $v \in V$, whether it belongs to a volatile component,
- for each $v \in V$ belonging to a volatile component, the representative vertex from this component.

Using this representation, we may obtain the value of any cell of $M_{[a,b]}$ in the following way. Consider two vertices u and w. First, assume that at least one of them (say u) does not belong to a volatile component. Let C be the connected component of u in $G_{[a,b]}$. Since C is not volatile, the connected component of u is equal to C in every element of $G_a, \ldots, G_b$. Hence, it suffices to check whether u and w are connected in $G_{[a,b]}$.

Otherwise, if u and w both belong to volatile components, we find the representative vertices v_i and v_j of u and w. Then, we simply check the value of $[M_{[a,b]}]_{v_i,v_j}$, which is stored explicitly. Overall, the implicit representation of $M_{[a,b]}$ takes $O(\min((b-a)^2, n^2) + n)$ space.

The implicit representation of all matrices $M_{[a,b]}$ can be simply computed in time that is linear in their size. We proceed in a bottom-up fashion on the segment tree. For a leaf interval $[a, a]$, there are no volatile components, so we simply store identifiers of connected components, which we have already computed. Now consider a non-leaf interval $[a, b]$, and $b - a \leqslant n$. First, note that we have computed the identifiers of connected components of $G_{[a,b]}$ before. In order to find volatile components, we simply iterate through all edges that are added or deleted in the interval $[a, b]$, which takes $O(b-a) = O(n)$ time. Then, we compute representative vertices in $O(n)$ time. Finally, we compute the $s \times s$ submatrix of $M_{[a,b]}$. Observe that

$$[M_{[a,b]}]_{x,y} = \max([M_{\textsc{Left}([a,b])}]_{x,y}, [M_{\textsc{Right}([a,b])}]_{x,y}), \tag{5.1}$$

and this expression can be evaluated in constant time (since the implicit representations of $M_{\textsc{Left}([a,b])}$ and $M_{\textsc{Right}([a,b])}$, which allow constant-time access, have already been computed). Thus, the implicit representation of $M_{[a,b]}$ is computed in $O(\min((b-a)^2, n^2) + n)$ time. It remains to consider the case when $b - a > n$, but then we simply compute an $n \times n$ matrix using Formula 5.1.

Observe that the implicit representations of $F_{[a,b]}$ and $L_{[a,b]}$ can be computed in an analogous way. Let us focus on $F_{[a,b]}$. If two vertices u and w are connected in $G_{[a,b]}$, we set $[F_{[a,b]}]_{u,w} = a$. If they stay disconnected in

every graph among $G_a, \ldots, G_b$, we set $[F_{[a,b]}]_{u,w} = \infty$. Otherwise we can use a formula $[F_{[a,b]}]_{x,y} = \min([F_{\textsc{Left}([a,b])}]_{x,y}, [F_{\textsc{Right}([a,b])}]_{x,y})$.

It remains to bound the total running time and space usage. Assume that $t = 2^d$. There are 2^{d-i} elementary intervals containing 2^i elements. For each such interval, the running time is $O(\min(2^{2i}, n^2) + n)$. Thus, we have

$$\begin{aligned}\sum_{i=0}^{d} 2^{d-i}(\min(2^{2i}, n^2) + n) =& \sum_{i=0}^{d} 2^{d-i} n + \sum_{i=0}^{\lfloor \log n \rfloor} 2^{d-i} 2^{2i} + \sum_{\lfloor \log n \rfloor + 1}^{d} 2^{d-i} n^2 \\ =& nt + \sum_{i=0}^{\lfloor \log n \rfloor} 2^{d+i} + 2^d n^2 \sum_{\lfloor \log n \rfloor + 1}^{d} 2^{-i} \\ =& nt + 2^d O(n) + 2^d n^2 O(1/n) = O(nt).\end{aligned}$$

Note that we have used an assumption that $\log n \leqslant d$, which follows from $n \leqslant t$, but this was for convenience only. We can derive the same bound in the case when $t = \Omega(n)$. □

5.2.1 Answering Queries

After the preprocessing phase, a query regarding an interval $[a, b]$ can be answered by decomposing it into $O(\log t)$ elementary intervals. The matrices $M_{[a,b]}$ allow us to obtain the answer for every elementary interval in $O(1)$ time, so each query requires $O(\log t)$ total time. However, this can be also done faster. We show how to use matrices $F_{[a,b]}$ and $L_{[a,b]}$ to improve the query time to $O(1)$.

Moreover, it is possible to extend the algorithm, so that it reports the graphs in which the two given vertices are connected. This can be done optimally, i.e., the graphs can be returned one by one, each with constant delay.

Let $\textsc{Parity}(x) := \max_i 2^i \mid x$ and $\textsc{MaxParity}(a, b)$ be the number from the interval $[a, b]$ that maximizes the value of $\textsc{Parity}$.

Proposition 5.2.3. *For any $1 \leqslant a \leqslant b$, the value of $\textsc{MaxParity}(a, b)$ is uniquely defined.*

Proof. Assume that $\textsc{Parity}(x) = \textsc{Parity}(y)$ and $a \leqslant x \leqslant y \leqslant b$. Let $z := x + 2^{\textsc{Parity}(x)}$. Clearly, $\textsc{Parity}(z) > \textsc{Parity}(x)$. Moreover, z is the smallest element greater than x, such that $\textsc{Parity}(z) \geqslant \textsc{Parity}(x)$. Thus, $z < y$, so $a \leqslant z \leqslant b$. Hence, if $\textsc{Parity}(x) = \textsc{Parity}(y)$, there exists z, such that $x < z < y$ and $\textsc{Parity}(z) > \textsc{Parity}(x)$. □

Lemma 5.2.4. *Let k be a positive integer. After preprocessing in $O(k)$ time, it is possible to compute* MaxParity(a, b) *for any $1 \leqslant a \leqslant b \leqslant k$ in constant time.*

Proof. The pseudocode of an algorithm for computing MaxParity is given as Algorithm 2. We use `xor`, `and` and `not` to denote the standard bitwise operations, whereas MaxPower(x) is the largest power of 2 that is not greater than x. In other words, MaxPower(x) returns the most significant bit of x.

Algorithm 2 MaxParity(a, b)

Require: $1 \leqslant a \leqslant b$

```
1: if a = b then
2:     return a
3: else if a ⩽ MaxPower(b) then
4:     return MaxPower(b)
5: else
6:     d := MaxPower(a xor b)
7:     c := b and (not (d − 1))
8:     if Parity(c) > Parity(a) then
9:         return c
10:    else
11:        return a
```

We now prove its correctness. Let us denote the correct result of the function by M. There are three cases to consider, one for every branch of the **if** statement from the first line. The correctness of the first case is trivial. Consider the second case. Observe that MaxParity(b) is the number with the largest value of Parity in the interval $[1, b]$. Moreover, in this case, this number lies in the interval $[a, b]$, from which we infer the desired.

Let us move to the third case. We have that MaxPower$(b) < a < b$, so the binary representations of a and b have the same length. In the first step, d is computed. It is a number with just one bit – the most significant bit of all the bits that are different in a and b. Assume that $d = 2^{p-1}$, that is only p-th bit is on in d. Because a is smaller than b, p-th bit is on in b, but not in a. It remains to show that $M \in \{a, c\}$. There are now two cases to consider.

1. M is divisible by $2d = 2^p$. Then p trailing bits of M are equal to 0. However, in $[a, b]$ there can be at most one number with p trailing zeroes, as other bits are common to a and b. It follows easily, that $M = a$.

2. M is not divisible by $2d$. In this case, we show that there exists a number in $[a, b]$ that is divisible by d, which clearly maximizes the value of MaxParity. We claim that it is a number obtained from b by setting its $p - 1$ trailing bits to zeroes. Observe that this value is computed and assigned to c in the 7^{th} line ($d-1$ is a number consisting of p consecutive ones). Obviously c is divisible by $2^{p-1} = d$. Moreover $a < c$, as the most significant bit, in which they differ is the p-th one, which is on in c.

Since our algorithm chooses the better from among a and c, it correctly computes M. All operations can be performed in constant time after $O(k)$ preprocessing, as the values of Parity and MaxParity can be precomputed. □

Observe that it is possible to obtain the result of the above lemma, by using a data structure that performs range minimum queries in constant time [6]. However, such approach leads to a much more complicated algorithm.

Lemma 5.2.5. *Let $1 \leqslant a \leqslant b \leqslant t$. After preprocessing in $O(t)$ time, in $O(1)$ time we can compute two elementary intervals $[r, p]$, $[p + 1, s]$, such that $p \in [a, b]$ and $[a, b] \subseteq [r, s] \subseteq [1, t]$, or a single interval $[r, s]$, such that $1 \leqslant r \leqslant a$ and $s = b$.*

Proof. If $b = t$, we simply return $[1, t]$. Let us now assume $b < t$.

We use Lemma 5.2.4, which results in $O(t)$ preprocessing time. We first compute $p = \text{MaxParity}(a, b)$. Then, we find elementary intervals $[r, p]$ and $[p+1, s]$ with the minimal possible r and maximal possible s. Such intervals can be precomputed for all values of p in the beginning in $O(t)$ time.

Clearly, $p \in [a, b]$. We claim that the interval $[a, b]$ is contained in $[r, s]$. It suffices to show that neither $r-1$ nor s is contained in $[a, b]$. By Lemma 2.2.4, $r = p - 2^{\text{Parity}(p)} + 1$. We have $\text{Parity}(r - 1) = \text{Parity}(p - 2^{\text{Parity}(p)}) > \text{Parity}(p)$. Since p maximizes the value of Parity among elements of $[a, b]$, $r - 1 \notin [a, b]$. Similarly, by Lemma 2.2.4, $s = p + 2^{\text{Parity}(p)}$, so a similar argument shows that $s \notin [a, b]$. Observe that $1 \leqslant r$ (trivially) and $s \leqslant t$, as $\text{Parity}(t) > \text{Parity}(p)$ and s is the smallest number, such that $p < s$ and $\text{Parity}(s) \geqslant \text{Parity}(p)$. The lemma follows. □

Theorem 5.2.6. *There exists an $O(nt)$ size data structure that, given a graph timeline G^t, consisting of graphs on n vertices, after preprocessing in $O(m + nt)$ time can answer* `exists` *queries in constant time, assuming that $t = O(n^c)$. Moreover, the data structure can report indices of the graphs, in which u and v are connected, each with constant delay.*

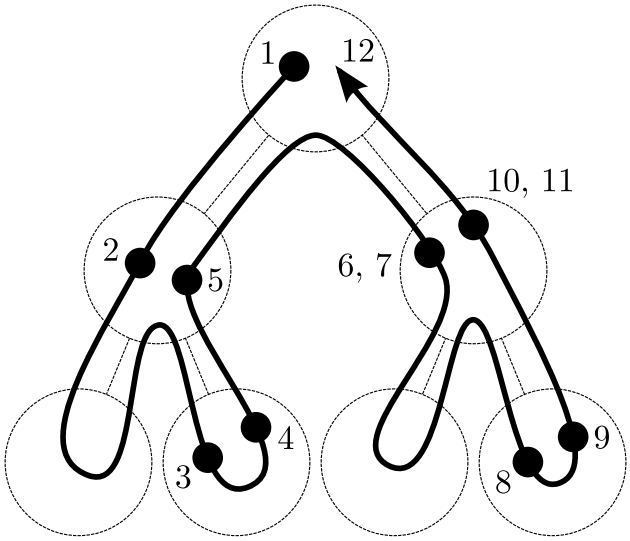

Figure 5.3: The in-order traversal of the CHT. The black circles denote the steps of the traversal when new graphs $\tilde{G}_i$ are created. Their indices are marked next to each circle. Note that during both the first and last visits of a node $[a, b]$, we create $|D_{[a,b]}|$ new graphs.

Proof. We first use Lemma 5.1.2 to reduce the problem to a timeline with no permanent edges in $O(m + t)$ time. Then, we use Lemma 5.2.2 to compute matrices $M_{[a,b]}$, $F_{[a,b]}$ and $L_{[a,b]}$ for all elementary intervals $[a, b]$ in $O(nt)$ time.

Consider an `exists`(u, v, a, b) query. We use Lemma 5.2.5 to find two elementary intervals $[r, p]$ and $[p + 1, s]$, such that $p \in [a, b]$ and $[a, b] \subseteq [r, s]$ (the case when the lemma computes a single interval is only easier). Using matrices $F_{[a,b]}$ and $L_{[a,b]}$ we can find the largest $x \in [r, p]$, such that u and w are connected in G_x, as well as the smallest $y \in [p + 1, s]$, such that u and w are connected in G_y. If either x or y (say x) belongs to $[a, b]$, we know that u and w are connected in G_x and $a \leqslant x \leqslant b$. We can then recurse on $[a, x - 1]$ and $[x + 1, b]$ to report further values. Otherwise, u and w are not connected in any of $G_a, \ldots, G_b$. □

5.3 `forall` Queries

In this section we show our algorithm for answering `forall` queries. Let G^t be a graph timeline consisting of graphs, whose vertex set is V. Assume we have computed lists $D_{[a,b]}$ for each elementary interval $[a, b]$ (see Lemma 5.1.4). Consider an inorder traversal of the CHT. We first traverse the root, then

recursively traverse its left subtree, visit the root again, recursively traverse the right subtree and finish in the root. We use the traversal to build a sequence of T graphs. As we traverse the tree, we maintain a graph G' on the vertex set V. The graph G' is modified as follows during the traversal. When we first visit an elementary interval $[a, b]$, we add all elements of $D_{[a,b]}$ one by one to G'. Moreover, after we last visit an elementary interval $[a, b]$, we remove all elements of $D_{[a,b]}$ from G' in reverse order (i.e., we first delete the last edge on the list). Now, we create a sequence of graphs, in which every time we modify G', we append the updated graph to the sequence. Denote the resulting sequence by $\tilde{G}_1, \tilde{G}_2, \ldots, \tilde{G}_T$.

The entire process is illustrated in Figure 5.3. We have $T = 12$, $\tilde{G}_2 = G_1, \tilde{G}_3 = G_2, \tilde{G}_7 = G_3$ and $\tilde{G}_8 = G_4$.

Proposition 5.3.1. *During the first visit to an elementary interval, once we add to G' all edges of $D_{[a,b]}$, we have $G' = G_{[a,b]}$.*

Proof. This follows from the following fact, which can be easily proven inductively. Assume the traversal is currently in elementary interval $[a, b]$. Then the edges of G' are the union of $D_{[c,d]}$ for all ancestors $[c, d]$ of $[a, b]$ and some prefix of $D_{[a,b]}$. □

Corollary 5.3.2. *For every elementary interval $[a, b]$, at some point G' is equal to $G_{[a,b]}$.*

For our algorithm, the following property is crucial.

Proposition 5.3.3. *$G_1, \ldots, G_t$ is a subsequence of $\tilde{G}_1, \tilde{G}_2, \ldots, \tilde{G}_T$.*

Moreover, we can bound the length of $\tilde{G}_1, \tilde{G}_2, \ldots, \tilde{G}_T$.

Proposition 5.3.4. *$T = O(t \log t)$.*

Proof. By Lemma 5.1.4, the total length of all lists $D_{[a,b]}$ is $O(t \log t)$. Each edge of each list $D_{[a,b]}$ is added to G' and removed from it only once. Thus, $T = O(t \log t)$. □

In our algorithm, we actually do not need to compute the sequence $\tilde{G}_1, \ldots, \tilde{G}_T$. Instead we would work with a DSDS.

Lemma 5.3.5. *We can maintain a DSDS C' of G' during the traversal in $O(t \log t \log n)$ time. In total, $O(t \log t)$ changes are made to C' during the traversal.*

Proof. The traversal algorithm works as follows:

1. Add to G' all edges of $D_{[a,b]}$.
2. Recursively traverse $\textsc{Left}([a,b])$ (if it exists).
3. Recursively traverse $\textsc{Right}([a,b])$ (if it exists).
4. Remove from G' all edges of $D_{[a,b]}$ in reverse order.

Thus, in order to maintain a DSDS of G', it suffices to use a DSDS which supports two operations: adding an edge, or *undoing* the last performed edge addition. We modify the DSDS we use, so that it records every change made to the data structure, which enables us to undo them. By Proposition 2.3.6, every `union` operation makes only $O(1)$ changes to the data structure, so each undo operation requires only $O(1)$ time. Every `union` operation requires $O(\log n)$ time. By Proposition 5.3.4, the traversal consists of $O(t \log t)$, so performing all operations requires $O(t \log t \log n)$ time. □

Thus, after the i-th step of the traversal, C' is a DSDS of $\tilde{G}_i$. Denote this version of C' by $\tilde{C}_i$. Each $\tilde{C}_i$ has a corresponding parent array. By $\tilde{p}(v)_i$ we denote the parent of a vertex v in $\tilde{C}_i$ and by $\tilde{r}(x)_i$ – the representative of x in $\tilde{C}_i$. We use $\tilde{p}(v)$ to denote the entire sequence $\tilde{p}(v)_1, \ldots, \tilde{p}(v)_T$ (and similarly for $\tilde{r}(v)$). Moreover, if $A = a_1, \ldots, a_k$ is a sequence, we define $A[c \ldots d]$ to be the sequence $a_c, a_{c+1}, \ldots, a_d$.

In our algorithm, we trace how the parent of a vertex changes, as C' is modified. In the first step of the preprocessing phase of our algorithm the ultimate goal is to compute the sequences $\tilde{p}(v)$. For efficiency, we use *run-length encoding* (RLE) to store each sequence.

Run-length encoding is used to store sequences, in which many consecutive elements are the same. In run-length encoding of a sequence s, every fragment of a sequence s consisting of equal elements is replaced by just one pair. The fragment consisting of k repetitions of an element x is replaced with a pair (x, k).

Proposition 5.3.6. *Let $s = s_1, \ldots, s_n$ be a sequence, such that n is an integer, which fits in a machine word. Assume there are exactly k indices $2 \leqslant i \leqslant n$, such that $s_i \neq s_{i-1}$. Then, run-length encoding of sequence s requires $O(k)$ space.*

Thus, to produce the encodings, we need to detect the pairs of consecutive elements, which are different. In the following, we say that i is a *critical moment* for x if $\tilde{p}(x)_i \neq \tilde{p}(x)_{i+1}$ or $i = 0$.

Lemma 5.3.7. *Let G^t be a graph timeline without permanent edges. Then, we can compute RLE encoding of all sequences $\tilde{p}(v)$ in $O(n + t \log t \log n)$ time. The representation uses $O(t \log t)$ space.*

Proof. We use Lemma 5.3.5 to maintain C' during the traversal. In order to produce the RLE encodings of a sequence $\tilde{p}(v)$, we need to compute the indices i, such that $\tilde{p}(v)_i \neq \tilde{p}(v)_{i+1}$. We obtain exactly one such index for every change made to C'. By Lemma 5.3.5, in total $O(t \log t)$ changes are made. In addition, if some sequence is never modified during the traversal, we can produce its encoding in $O(1)$ time. Handling all such sequences takes $O(n) = O(t)$ time and space. □

Note that the sequences $\tilde{p}(v)$ carry all connectivity information about all graphs $\tilde{G}_1, \ldots, \tilde{G}_T$. In particular, for each $1 \leqslant i \leqslant T$, we can find the representatives of every vertex v, that is the value of $\tilde{r}(v)_i$, which gives the connected components of $\tilde{G}_i$. In particular, we have the following.

Proposition 5.3.8. *Let $u, w \in V$ and $1 \leqslant a \leqslant b \leqslant T$. Then u and w are connected in each of $\tilde{G}[a \ldots b]$ if and only if the sequences $\tilde{r}(u)[a \ldots b]$ and $\tilde{r}(w)[a \ldots b]$ are equal.*

However, our ultimate goal is to answer connectivity queries regarding the timeline $G_1, \ldots, G_t$, which is a subsequence of $\tilde{G}_1, \tilde{G}_2, \ldots, \tilde{G}_T$. Let $s_1, \ldots, s_t$ be the sequence such that $\tilde{G}_{s_i} = G_i$. Our goal is to use the following property.

Proposition 5.3.9. *Let $u, w \in V$ and $1 \leqslant a \leqslant b \leqslant t$. Then, u and w are connected in each of $G[a \ldots b]$ if and only if the sequences $r(u)[a \ldots b]$ and $r(w)[a \ldots b]$ are equal. This is is turn equivalent to checking whether the sequences $\tilde{r}(u)_{s_a}, \tilde{r}(u)_{s_{a+1}}, \ldots, \tilde{r}(u)_{s_b}$ and $\tilde{r}(w)_{s_a}, \tilde{r}(w)_{s_{a+1}}, \ldots, \tilde{r}(w)_{s_b}$ are equal.*

From now on, the only information of our algorithm about the timeline are the RLE encoding of the sequences $\tilde{p}(v)$ and the sequence $s_1, \ldots, s_t$. In order to answer query regarding vertices u and v, we need to compare some fragments of sequences $r(u)$ and $r(v)$. We achieve that by computing and comparing their hashes. However, we first describe how to compute hashes of fragments of sequences $\tilde{r}(v)$, and then show a simple modification to the algorithm, which would yield the desired hashes.

Let us now describe the process of computing hashes. Let x and y be two sequences. We use the hashing scheme described in Section 2.3.4. Recall that we denote by $H(x)$ the hash of sequence x and by $H(x) \oplus H(y)$ the a hash of a sequence obtained by appending y to x. If y is a prefix of x, we use $H(x) \ominus H(y)$ to denote the hash of the sequence obtained from x by removing the prefix equal to y.

In order to be able to obtain hashes of some fragments of $\tilde{r}(x)_1 \ldots \tilde{r}(x)_T$, we precompute selected prefix hashes. A *prefix hash* of vertex x of length k, denoted by $ph(x, k)$, is equal to $H(\tilde{r}(x)[1 \ldots k])$.

The prefix hashes $ph(x,k)$ are computed only for each critical moment k of vertex x. To perform the computation, we use the following lemmas.

Lemma 5.3.10. *Let $k > 0$ be a critical moment for a vertex x. Then $\tilde{p}(x)_k$ is a representative in $\tilde{C}_k$. Moreover, x is a representative in $\tilde{C}_k$ if and only if it is* not *a representative in $\tilde{C}_{k+1}$.*

Proof. Our goal is to prove that x is either a representative in $\tilde{C}_k$ or a child (direct descendant) of the representative. In the sequence $\tilde{C}_1, \ldots, \tilde{C}_T$, $\tilde{C}_{i+1}$ is obtained from $\tilde{C}_i$ by performing or undoing a single `union` operation. By Proposition 2.3.6 an `union` operation changes at most one parent pointer in a DSDS, and if some pointer does change, it is the parent pointer of one a representatives of one of the merged subsets. We consider two cases. First, assume that $\tilde{C}_{k+1}$ is obtained form $\tilde{C}_k$ by performing an `union` operation. Then, since $\tilde{p}(x)_k \neq \tilde{p}(x)_{k+1}$, we infer that x is a representative in $\tilde{C}_k$ (which implies that $\tilde{p}(x)_k = x$ is also a representative), and it is a child of a representative in $\tilde{C}_{k+1}$. On the other hand, if $\tilde{C}_{k+1}$ is obtained form $\tilde{C}_k$ by undoing an `union` operation, by a reverse reasoning we infer that x is a not representative in $\tilde{C}_k$, but it is a representative in $\tilde{C}_{k+1}$. □

In the next lemma we use the fact that if k' is the largest critical moment for x such that $k' < k$ then the elements $\tilde{p}(x)_{k'+1}, \tilde{p}(x)_{k'+2}, \ldots, \tilde{p}(x)_k$ are all equal.

Lemma 5.3.11. *Let x be a representative in $\tilde{C}_k$, $k > 0$, and k' be the largest critical moment of x lower than x. Then, x is a representative in $\tilde{C}_{k'+1}, \ldots, \tilde{C}_k$.*

Proof. From the definition of k' it follows that all elements of $\tilde{p}(x)[k'+1 \ldots k]$ are equal. Since x is a representative in $\tilde{C}_k$, they are all equal to $\tilde{p}(x)_k = x$. The lemma follows. □

We now show a formula, which is used to prove the following lemmas. It is an easy consequence of the definition of prefix hashes.

Proposition 5.3.12. *Let x be a vertex, $k > 0$ and let k' be the largest critical moment of x lower than k. Then, $ph(x,k) = ph(x,k') \oplus H(\tilde{r}(x)[k'+1 \ldots k])$*

Lemma 5.3.13. *If x is a representative in $\tilde{C}_k$ then the value of $ph(x,k)$ can be computed in $O(\log\log t)$ time, given $ph(y,k')$ for all pairs (y,k') such that $k' < k$ and k' is a critical moment for y. This requires initial preprocessing in $O(t \log t \log\log t)$ expected time.*

Proof. We first show how to compute $ph(x,k)$ in $O(\log t)$ time and then we speed the algorithm up. Let k' be the largest critical moment of x lower than k. The value of k' can be found in $O(\log t)$ time using binary search. By Proposition 5.3.12, $ph(x,k) = ph(x,k') \oplus H(\tilde{r}(x)[k'+1 \dots k])$.

Since x is a representative in $\tilde{C}_k$, by Lemma 5.3.11, it is a representative in all of $\tilde{C}_{k'+1}, \dots, \tilde{C}_k$, so all elements of $\tilde{r}(x)[k'+1 \dots k]$ are equal to x. Thus, we can simply compute the hash of $x^{k-k'}$ (x repeated $k-k'$ times) and combine it with $ph(x,k')$. In this way we obtain $ph(x,k)$.

We can speed up the computation of k', by using Y-fast tries [46]. It is a data structure, which can be initialized in $O(n \log\log M)$ expected time for a set of n integers belonging to $\{0, \dots, M-1\}$, and afterwards supports predecessor queries in deterministic $O(\log\log M)$ worst-case time. In our case, for each vertex v we can build an Y-fast trie containing its critical moments. There are $O(t \log t)$ critical moments, and each of them is an integer less than or equal to t. Thus, building all Y-fast tries takes $O(t \log t \log\log t)$ expected time. Once we do that, given a vertex v and an integer k, we can find the largest critical moment k' lower than k in $O(\log\log t)$ worst-case time. □

By joining together the above two lemmas one can obtain the following lemma.

Lemma 5.3.14. *For any vertex x and any critical moment k, the value of $ph(x,k)$ can be computed in $O(\log\log t)$ time, given $ph(\tilde{p}(x)_k, k)$ and $ph(y,k')$ for all pairs (y,k') such that $k' < k$ and k' is a critical moment for y. This requires initial preprocessing in $O(t \log t \log\log t)$ expected time.*

Proof. If $\tilde{p}(x)_k = x$, we can simply use Lemma 5.3.13, so from now on we assume that $\tilde{p}(x)_k \neq x$.

Let k' be the largest critical moment lower than k. By Proposition 5.3.12, $ph(x,k) = ph(x,k') \oplus H(S)$, where $S = \tilde{r}(x)[k'+1 \dots k]$. By the definition of the parent and representative, $S = \tilde{r}(\tilde{p}(x)_{k'+1})_{k'+1} \dots \tilde{r}(\tilde{p}(x)_k)_k$. Since there are no critical moments between k' and k, we infer that elements $\tilde{p}(x)_{k'+1}, \tilde{p}(x)_{k'+2}, \dots, \tilde{p}(x)_k$ are all equal to $\tilde{p}(x)_k$. Thus S is equal to $\tilde{r}(\tilde{p}(x)_k)[k'+1 \dots k]$. This means that $H(S) = ph(\tilde{p}(x)_k, k) \ominus ph(\tilde{p}(x)_k, k')$. By our assumption $ph(\tilde{p}(x)_k, k)$ is given, so it remains to obtain $ph(\tilde{p}(x)_k, k')$.

Since x is not a representative in $\tilde{C}_k$, it is also not a representative in $\tilde{C}_{k'+1}$ (because $\tilde{p}(x)_k \neq x$ and $\tilde{p}(x)_k = \tilde{p}(x)_{k'+1}$), but x *is* a representative in $\tilde{C}_{k'}$ (by Lemma 5.3.10). Thus, by Proposition 2.3.6, $\tilde{p}(x)_{k'+1}$ is a representative in $\tilde{C}_{k'}$. This means that $ph(\tilde{p}(x)_{k'+1}, k')$ can be computed in $O(\log\log t)$ time, by Lemma 5.3.13. From $\tilde{p}(x)_{k'+1} = \tilde{p}(x)_k$ it follows that $ph(\tilde{p}(x)_{k'+1}, k') = ph(\tilde{p}(x)_k, k')$, so we obtain $ph(\tilde{p}(x)_k, k')$, as needed. □

Lemma 5.3.15. *Computing all prefix hashes takes $O(t \log t(\log n + \log\log t))$ expected time.*

Proof. We first use $O(t \log t \log n)$ time to compute the sequences $\tilde{p}(v)$. We compute $ph(x,k)$ in the order of increasing values of k. For $k = 0$ and for each x, $ph(x,0)$ is a hash of an empty sequence, so all of such hashes can be computed in $O(n)$ time. For $k > 0$, by Lemma 5.3.14, to compute $ph(x,k)$, we need to know $ph(\tilde{p}(x)_k, k)$. As k is a critical moment, $\tilde{p}(x)_k$ is a representative in $\tilde{C}_k$ (by Lemma 5.3.10), so we can use Lemma 5.3.13 to compute $ph(\tilde{p}(x)_k, k)$ in $O(\log\log t)$ time. As a result, every $ph(x,k)$, where k is a critical moment for x, can be computed in $O(\log\log t)$ time. Therefore, computing all $O(t \log t)$ prefix hashes requires $O(n + t \log t \log\log t)$ expected time. The total expected time is $O(t \log t(\log n + \log\log t))$. The time bound is in expectation, because the preprocessing of Lemma 5.3.13 takes $O(t \log t \log\log t)$ expected time. □

5.3.1 Answering Queries

We now show how to use the preprocessed information to compute hashes of arbitrary fragments of $\tilde{r}(x)_1 \ldots \tilde{r}(x)_T$.

Lemma 5.3.16. *Once we preprocess prefix hashes as in Lemma 5.3.15, for each vertex v the hash of an arbitrary fragment of $\tilde{r}(v)_1 \ldots \tilde{r}(v)_T$ can be computed in $O(\log n \log\log t)$ time.*

Proof. Observe that it suffices to describe how to compute $ph(x,k)$ for arbitrary k, since $H(\tilde{r}(x)[a \ldots b]) = ph(x,b) \ominus ph(x,a-1)$. We use Lemma 5.3.14, that reduces our problem to the problem to computing $h(x, \tilde{p}(x)_k)$ (with an $O(\log\log t)$ overhead). In other words, computing a hash for x can be reduced to computing some hash for a parent of x. This value is then computed recursively. Since the depth of all trees in $\tilde{C}_k$ is bounded by $O(\log n)$, the recursion has $O(\log n)$ levels. On each level of the recursion we use Lemma 5.3.14, which requires $O(\log\log t)$ time. Thus, computing an arbitrary prefix hash requires $O(\log n \log\log t)$ time. □

This allows us to hash arbitrary fragment of $\tilde{r}(x)$ for every vertex x. However, as mentioned before, what we need are the hashes of fragments of $r(x)$. We now describe how to modify the algorithm, so that it computes the right hashes.

The only moment when we actually hash a fragment of a sequence is in the proof of Lemma 5.3.13, when we hash a word of the form x^l. In such case we only want to include the elements that are present in $r(x)$. Hence,

given some fragment of $\tilde{r}(x)$, we need to know how many of its elements belong to $r(x)$. Denote by s_i the indices of $\tilde{r}(v)$, which correspond to $r(v)$, i.e., $r(x)_i = \tilde{r}(x)_{s_i}$. Thus, to count the number of elements in $\tilde{r}(x)[a \dots b]$ that correspond to elements in $r(x)$, it suffices to count the number of elements of s_i that are between a and b. As s_i is an increasing sequence, whose values are bounded by $T = O(t \log t)$, we simply compute an array that for each j stores the number of elements of s_i that are smaller than j. This array allows to count the number of elements of s_i in any interval in constant time. As a result we obtain the following theorem. Note that the running time depends on m, as we use Lemma 5.1.2.

Theorem 5.3.17. *There exists a randomized Monte Carlo data structure that, given a graph timeline G^t, after preprocessing in $O(m + t \log t(\log n + \log\log t))$ time can answer* `forall` *queries in $O(\log n \log\log t)$ time. It requires $O(n + t \log t)$ space.*

5.3.2 Deterministic Algorithm

In this section we show a deterministic variant of the algorithm for answering `forall` queries. It uses a data structure for maintaining a family of dynamic sequences. This problem has been first solved by Mehlhorn et al [36], and later the solution has been improved in [2]. Let us now describe this data structure briefly.

The operations performed on the data structure involve a pair of sequences. Let l be the length of the sequences involved in a single operation. Two sequences in the family can be joined or split in $O(\log l \log^* l)$ time, tested for equality in $O(1)$ time, and for any letter a and integer $k > 0$ the sequence a^k can be created in $O(1)$ time. The data structure can be used in place of prefix hashes. It allows us to maintain each desired prefix in the data structure.

During the process of computing the prefixes for all critical moments, an existing prefix is joined with a sequence in which all elements are equal. This is performed $O(t \log t)$ times and the length of the sequences involved in one operation is $O(t)$. Hence, by using the data structure it takes $O(\log t \log^* t)$ time to compute a single prefix, which gives $O(t \log^2 t \log^* t)$ time overhead for the whole process of computing hashes. This dominates the running time of the preprocessing phase.

However, the data structure allows us to answer queries in a simpler and more effective way. After computing all prefixes, the data structure contains a representation of sequences $r(v)_1, \dots, r(v)_t$ for each v. To answer a query, we need to compare $r(v)_a \dots r(v)_b$ to $r(u)_a \dots r(u)_b$, but both these sequences

can be created with two split operations, each involving sequences of length $O(t)$. This requires $O(\log t \log^* t)$ time.

Moreover, we may use a simpler version of Lemma 5.3.13, which achieves a running time of $O(\log t)$, but does not require preprocessing (see the proof for details). We obtain the following theorem.

Theorem 5.3.18. *There exists a deterministic data structure that, given a graph timeline G^t, after preprocessing in $O(m+t \log^2 t \log^* t)$ time can answer* `forall` *queries in $O(\log t \log^* t)$ time. It requires $O(t \log^2 t \log^* t)$ space.*

5.4 Subsequent Results

The results of this chapter have been improved and simplified by Karczmarz [27]. One of the observation he makes, is that we may compute a data structure similar to the connectivity history tree using observations of Lemma 5.1.2 at every level. Moreover, he slightly improves Lemma 5.1.4 and bounds the length of all lists by $O(t \log n)$.

As a result, he obtained a deterministic algorithm that after preprocessing in $O(t \log n)$ time may answer `forall` queries in $O(\log n)$ time. Moreover, he has removed the requirement that $t = O(n^c)$ from the algorithm for `exists` queries. Finally, he showed a lower bound for answering `exists` queries of the following form. Assume there exists a data structure for answering `exists` queries, which uses $O(t^{1.41-\varepsilon}\text{polylog}(t))$ time to preprocess and answer $O(t)$ queries. Then, there exists an algorithm for finding triangles in an t-edge graph, which runs in $O(t^{1.41-\varepsilon}\text{polylog}(t))$ time.

Chapter 6

Open Problems

We have presented new dynamic graph algorithms for maintaining approximate Steiner tree over a dynamic set of terminals, decremental connectivity in planar graphs, and connectivity in graph timelines. In each of these areas there are some interesting open problems to consider.

1. Is it possible to make our time bounds for the maintenance of dynamic Steiner tree *worst-case*? This could be achieved, e.g., by showing a polylogarithmic worst-case time algorithm for dynamic MST.

2. Is it possible to improve the approximation ratios of our algorithms for dynamic Steiner tree? In [31], a $(6+\varepsilon)$-approximate algorithm for general graphs, and a $(2+\varepsilon)$-approximate algorithm for planar graphs is shown. However, in static case it is known how to obtain 1.39 approximation for general graphs and a PTAS for planar graphs. Can we improve the dynamic algorithms, so that they come closer to the static case?

3. Concerning decremental connectivity, is it possible to solve decremental connectivity in general graphs in $o(n \log n)$ time? Our result shows that decremental connectivity in planar graphs is strictly easier than fully dynamic one. Moreover, the existing lower bound for dynamic connectivity in general graphs is only for the fully dynamic variant. In the case of decremental connectivity in general graphs no lower bounds have been shown.

4. Finally, a lower bound of $\Omega(n^{1.41})$ (conditional on triangle detection) for answering n `exists` queries on a timeline consisting of n graphs on n vertices have been shown in [27]. At the same time the preprocessing time of our algorithm in this case is $O(n^2)$. Is it possible to reduce the running time to $O(n^{2-\varepsilon})$?

Bibliography

[1] Ehud Aharoni and Reuven Cohen. Restricted dynamic Steiner trees for scalable multicast in datagram networks. *IEEE/ACM Trans. Netw.*, 6(3):286–297, 1998.

[2] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, SODA '00, pages 819–828, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[3] Stephen Alstrup, Jens P. Secher, and Maz Spork. Optimal on-line decremental connectivity in trees. *Inf. Process. Lett.*, 64(4):161–164, 1997.

[4] Sanjeev Arora. Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. *J. ACM*, 45(5):753–782, 1998.

[5] Fred Bauer and Anujan Varma. ARIES: A rearrangeable inexpensive edge-based on-line Steiner algorithm. *IEEE Journal of Selected Areas in Communications*, 15:382–397, 1995.

[6] Michael A. Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005.

[7] Glencora Borradaile, Philip N. Klein, and Claire Mathieu. An O(n log n) approximation scheme for Steiner tree in planar graphs. *ACM Transactions on Algorithms*, 5(3), 2009.

[8] Jaroslaw Byrka, Fabrizio Grandoni, Thomas Rothvoß, and Laura Sanità. An improved LP-based approximation for Steiner tree. In Leonard J. Schulman, editor, *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 583–592. ACM, 2010.

[9] Xiuzhen Cheng, Yingshu Li, Ding-Zhu Du, and HungQ. Ngo. Steiner trees in industry. In Ding-Zhu Du and PanosM. Pardalos, editors, *Handbook of Combinatorial Optimization*, pages 193–216. Springer US, 2005.

[10] Fan Chung. Graph theory in the information age. *Notices of the American Mathematrical Society*, 57(06):726.

[11] David Eppstein. Offline algorithms for dynamic minimum spanning tree problems. *J. Algorithms*, 17(2):237–250, 1994.

[12] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. *J. ACM*, 44:669–696, 1997.

[13] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. Separator based sparsification: I. Planarity testing and minimum spanning trees. *J. Comput. Syst. Sci.*, 52(1):3–27, 1996.

[14] David Eppstein, Giuseppe F. Italiano, Roberto Tamassia, Robert Endre Tarjan, Jeffery Westbrook, and Moti Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. *J. Algorithms*, 13(1):33–54, 1992.

[15] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985.

[16] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In David S. Johnson, editor, *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*, pages 345–354. ACM, 1989.

[17] Albert Gu, Anupam Gupta, and Amit Kumar. The power of deferral: maintaining a constant-competitive steiner tree online. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 525–534. ACM, 2013.

[18] Anupam Gupta and Amit Kumar. Online steiner tree with deletions. 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 455–467. SIAM, 2014.

[19] Jens Gustedt. Efficient union-find for planar graphs and other sparse graph classes. *Theoretical Computer Science*, 203(1):123 – 141, 1998.

[20] Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, July 1999.

[21] Monika Rauch Henzinger and Michael L. Fredman. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica*, 22(3):351–362, 1998.

[22] Monika Rauch Henzinger and Valerie King. Maintaining minimum spanning forests in dynamic graphs. *SIAM J. Comput.*, 31(2):364–374, 2001.

[23] 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, 2001.

[24] Sung-Pil Hong, Heesang Lee, and Bum Hwan Park. An efficient multicast routing algorithm for delay-sensitive applications with dynamic membership. In *INFOCOM '98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1433–1440 vol.3, Mar 1998.

[25] Makoto Imase and Bernard M. Waxman. Dynamic Steiner tree problem. *SIAM J. Discrete Math.*, 4(3):369–384, 1991.

[26] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, pages 1131–1142. SIAM, 2013.

[27] Adam Karczmarz. Algorytmy dla problemów spójności w grafach nieskierowanych z historią. Master's thesis, University of Warsaw, Warsaw, Poland, 2014.

[28] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[29] Philip N. Klein, Shay Mozes, and Christian Sommer. Structured recursive separator decompositions for planar graphs in linear time. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium*

on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013, pages 505–514. ACM, 2013.

[30] Tomasz Kociumaka, Jakub Radoszewski, and Wojciech Rytter. Fast algorithms for abelian periods in words and greatest common divisor queries. In Natacha Portier and Thomas Wilke, editors, *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, February 27 - March 2, 2013, Kiel, Germany*, volume 20 of *LIPIcs*, pages 245–256. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[31] Jakub Łącki and Jakub Oćwieja and Marcin Pilipczuk and Piotr Sankowski and Anna Zych. Dynamic Steiner tree and subgraph TSP. *CoRR*, abs/1308.3336, 2013.

[32] Jakub Łącki and Piotr Sankowski. Reachability in graph timelines. In Robert D. Kleinberg, editor, *Innovations in Theoretical Computer Science, ITCS '13, Berkeley, CA, USA, January 9-12, 2013*, pages 257–268. ACM, 2013.

[33] Jakub Łącki and Piotr Sankowski. Optimal decremental connectivity in planar graphs. *CoRR*, abs/1409.7240, 2014.

[34] Nicole Megow, Martin Skutella, José Verschae, and Andreas Wiese. The power of recourse for online MST and TSP. In Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts, and Roger Wattenhofer, editors, *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part I*, volume 7391 of *Lecture Notes in Computer Science*, pages 689–700. Springer, 2012.

[35] Kurt Mehlhorn. A faster approximation algorithm for the Steiner problem in graphs. *Inf. Process. Lett.*, 27(3):125–128, 1988.

[36] Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining Dynamic Sequences under Equality Tests in Polylogarithmic Time. *Algorithmica*, 17(2):183–198, 1997.

[37] Joseph S. B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems. *SIAM J. Comput*, 28:402–408, 1996.

[38] Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006.

[39] Sriram Raghavan, G. Manimaran, C. Siva, and Ram Murthy. A rearrangeable algorithm for the construction of delay-constrained dynamic multicast trees. *IEEE/ACM Transactions on Networking*, 7:514–529, 1999.

[40] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[41] Mikkel Thorup. Decremental dynamic connectivity. *J. Algorithms*, 33(2):229–243, 1999.

[42] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In F. Frances Yao and Eugene M. Luks, editors, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 343–350. ACM, 2000.

[43] Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *J. ACM*, 51:993–1024, 2004.

[44] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.

[45] Freek van Walderveen, Norbert Zeh, and Lars Arge. Multiway simple cycle separators and I/O-efficient algorithms for planar graphs. 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 901–918. SIAM, 2013.

[46] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space Θ(n). *Inf. Process. Lett.*, 17(2):81–84, 1983.

[47] Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. 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 1757–1769. SIAM, 2013.