



PHD DISSERTATION

UNIVERSITY OF WARSAW

FACULTY OF MATHEMATICS, INFORMATICS AND MECHANICS

Interesting Graph Problems

Author:

Marcin Smulewicz

Supervisor:

Dr hab. Marcin Mucha

September 29, 2025

Author's declaration:

I hereby declare that this dissertation is my own work.

September 29, 2025

Date

Marcin Smulewicz

Supervisor's declaration:

This dissertation is ready to be reviewed.

September 29, 2025

Date

dr hab. Marcin Mucha

Abstract

This thesis presents several results in the field of graph theory.

This thesis discusses the following papers:

- **Many-Visits TSP Revisited** [33]

where we show how to solve the many-visits traveling salesman problem:

- in $\mathcal{O}^*(2^n \cdot D)$ time by randomized algorithm (D is the maximum distance between cities).
- in $\mathcal{O}^*(4^n)$ time.
- and in $\mathcal{O}(7.88^n)$ time using only polynomial space.

- **Determining 4-Edge-Connected Components in Linear Time** [36, 37]

where we present the first $\mathcal{O}(m)$ algorithm for finding 4-edge-connected components.

- **On Fully Dynamic Strongly Connected Components**[27]

where we provide:

- a conditionally optimal data structure for fully dynamic strongly connected components with $\mathcal{O}(n^{1.528})$ worst-case update time.
- and a conditionally optimal data structure for fully dynamic strong connectivity with $\mathcal{O}(n^{1.406})$ worst-case update time.

- **Decreasing the maximum average degree by deleting an independent set or a d-degenerate subgraph** [38]

where we study the notion of the maximum average degree $\text{mad}(G)$ of a graph G , defined as the largest average degree among all subgraphs of G . We prove that for every graph G and positive integer k such that $\text{mad}(G) \geq k$, there exists $S \subseteq V(G)$ such that $\text{mad}(G[V \setminus S]) \leq \text{mad}(G) - k$ and $G[S]$ is $(k - 1)$ -degenerate.

These problems come from diverse areas of graph theory. To solve them, we employ a mixture of algebraic techniques and flow-based methods, complemented by several carefully crafted ad hoc ideas.

2012 ACM Subject Classification: Theory of computation $\rightarrow$ Design and analysis of algorithms $\rightarrow$ Graph algorithms analysis

Keywords: Algorithm, Data Structure, Graph, Algebraic Methods, Flow, Traveling Salesman Problem, 4-Edge-Connectivity, Strong Connectivity, Maximum Average Degree

Streszczenie

Niniejsza rozprawa przedstawia szereg wyników z dziedziny teorii grafów.

Omówione zostały następujące prace naukowe:

- **Many-Visits TSP Revisited** [33],

w której pokazujemy, jak rozwiązać problem komiwojażera z wieloma wizytami:

- w czasie $\mathcal{O}^*(2^n \cdot D)$ za pomocą randomizowanego algorytmu (D oznacza maksymalną odległość pomiędzy miastami).
- w czasie $\mathcal{O}^*(4^n)$.
- i w czasie $\mathcal{O}(7.88^n)$ przy użyciu jedynie wielomianowej pamięci.

- **Determining 4-Edge-Connected Components in Linear Time** [36, 37],

w której prezentujemy pierwszy algorytm znajdujący 4-krawędziowo-spójne składowe w czasie $\mathcal{O}(m)$.

- **On Fully Dynamic Strongly Connected Components**[27],

w której przedstawiamy:

- warunkowo optymalną strukturę danych dla w pełni dynamicznych silnie spójnych składowych z pesymistycznym czasem aktualizacji $\mathcal{O}(n^{1.528})$.
- warunkowo optymalną strukturę danych dla w pełni dynamicznej silnej spójności z pesymistycznym czasem aktualizacji $\mathcal{O}(n^{1.406})$.

- **Decreasing the maximum average degree by deleting an independent set or a d-degenerate subgraph** [38],

w której badamy pojęcie maksymalnego średniego stopnia $\text{mad}(G)$ grafu G , definiowanego jako największy średni stopień wśród wszystkich podgrafów G . Dowodzimy, że dla każdego grafu G i każdej dodatniej liczby całkowitej k takiej, że $\text{mad}(G) \geq k$, istnieje zbiór $S \subseteq V(G)$ taki, że $\text{mad}(G[V \setminus S]) \leq \text{mad}(G) - k$ oraz $G[S]$ jest $(k - 1)$ -zdegenerowany.

Problemy pochodzą z różnych obszarów teorii grafów. Do ich rozwiązania wykorzystujemy połączenie technik algebraicznych i metod opartych na przepływach, uzupełnione kilkoma starannie opracowanymi doraźnymi pomysłami.

Tytuł rozprawy w języku polskim: Ciekawe algorytmy grafowe

Słowa kluczowe: algorytm, struktura danych, graf, metody algebraiczne, przepływ, problem komiwojażera, 4-krawędziowa spójność, silna spójność, maksymalny średni stopień

Contents

1	Introduction	4
2	Preliminary	7
3	Many-Visits TSP	9
3.1	Our Work	10
3.1.1	$\mathcal{O}^*(2^n D)$ algorithm	10
3.1.2	$\mathcal{O}^*(4^n)$	11
3.1.3	Polynomial Space $\mathcal{O}^*(7.88^n)$	11
3.1.4	My contribution	12
4	4-Edge-Connected Components	13
4.1	Our Work	14
4.1.1	Motivation	14
4.1.2	The Algorithm	14
4.1.3	My Contribution	16
5	Dynamic Strongly Connected Components	17
5.1	Our Work	18
5.1.1	Fully Dynamic Strongly Connected Components	18
5.1.2	Fully Dynamic Strong Connectivity	19
5.1.3	Planar Strongly Connected Components	19

5.1.4	My Contribution	20
5.2	Future Research	20
6	Maximum Average Degree	21
6.1	Our Work	22
6.1.1	The Main Theorem	22
6.1.2	Consequences	22
6.1.3	My Contribution	23
A	Many Visits TSP Revisited	28
B	Determining 4-Edge-Connected Components in Linear Time (Short Version)	50
C	Determining 4-Edge-Connected Components in Linear Time (Full Version)	65
D	On Fully Dynamic Strongly Connected Components	92
E	Decreasing the maximum average degree by deleting an independent set or a d-degenerate subgraph	107

Chapter 1

Introduction

Graphs are fundamental combinatorial structures that describe ways in which pairs of objects can be related. They can represent friendship networks, road networks, computer networks, possible states in games, or even more advanced mathematical concepts such as Markov chains or automata.

Formally, a graph is a pair of sets, one containing vertices, and the other containing pairs of vertices representing edges. This simple notion allows us to formalize a wide variety of algorithmic problems.

This work does not focus on a single family of graph problems. Instead, it demonstrates how certain algorithmic techniques can be applied to solve different challenging problems arising in various areas of graph theory.

The main techniques explored in this thesis are:

- Flow-based methods

For example, applying the max-flow min-cut theorem, or computing maximum flows and minimum-cost maximum flows in carefully designed networks.

- Algebraic techniques

For example, using the DeMillo–Lipton–Schwartz–Zippel Lemma [11, 47, 42], methods based on fast matrix multiplication [2] or the Cut & Count technique [10].

- Other standard algorithmic tools

Such as dynamic programming, disjoint-set unions, properties of the `low` function [23] or linear-time recursions in the style of median of medians [5] or Kärkkäinen-Sanders algorithm [29].

In this thesis, we apply these techniques to the following problems:

- **Many-visits traveling salesmen problem**

This problem is NP-hard, and in such cases the goal is usually to optimize the constant c in $\mathcal{O}(c^n)$ time algorithms.

In [33], we improved the best known algorithms for this problem, achieving:

- $\mathcal{O}^*(4^n)$ time,
- $\mathcal{O}(7.88^n)$ time using only polynomial space,
- and last but not least $\mathcal{O}^*(2^n \cdot D)$ time in a parameterized setting, where D is the maximum edge length in the graph.

If we do not enforce the connectivity of the solution, the problem becomes solvable in polynomial time by using a minimum-cost flow algorithm. In the actual problem, we also use a minimum-cost flow algorithm, but the main challenge is to ensure connectivity. We do this in various ways:

- In the $\mathcal{O}^*(4^n)$ algorithm, we use dynamic programming.
- In the $\mathcal{O}(7.88^n)$ algorithm, we iterate over certain skeletons of connected substructures and enforce connectivity using a modified flow network.
- In the $\mathcal{O}^*(2^n \cdot D)$ algorithm, we use an algebraic framework based on Lagrange interpolation and the DeMillo–Lipton–Schwartz–Zippel Lemma [11, 47, 42], which allows us to use the Cut & Count technique [10].

- **Fully dynamic strongly connected components and strong connectivity**

These problems are solvable in polynomial time, so the goal is to decrease the exponent c in $\mathcal{O}(n^c)$ time complexity.

In [27], we developed data structures with worst-case update time guarantees:

- $\mathcal{O}(n^{1.528})$ for maintaining fully dynamic strongly connected components,
- $\mathcal{O}(n^{1.406})$ for maintaining fully dynamic strong connectivity.

We used reachability data structures [41], [46] based on algebraic techniques, mainly fast matrix multiplication [2]. We also applied the DeMillo–Lipton–Schwartz–Zippel Lemma [11, 47, 42] and the unique witness trick [43] in a novel way.

- **4-edge-connected components**

This problem was known to be solvable in near-linear time. For such problems, the goal is to shave off remaining $\mathcal{O}(\log(n))$ factors from the time complexity.

In [36, 37], we achieved the first $\mathcal{O}(m)$ time algorithm for this problem.

This problem is inherently flow-related, since two vertices are 4-edge-connected if and only if the maximum flow between them is at least 4. Our algorithm uses ideas such as the max-flow min-cut theorem. The main idea of our algorithm relies on a linear time recursion on a geometrically shrinking sequence of instances.

We first presented a simple randomized $\mathcal{O}(m)$ time algorithm, which uses mainly DFS-like procedures supplemented with some algebraic techniques, and then derandomized it to obtain a deterministic algorithm without relying on those algebraic techniques.

Ideas from this algorithm can be used in practice, as the initial version of the algorithm was designed and implemented for competitive programming purposes.

- **Partitionability of maximum average degree**

This problem is purely combinatorial, the goal is to prove that the maximum average degree (mad) is a partitionable graph parameter.

In [38], we made an important step toward this goal. We showed how to remove an independent set from a graph, reducing mad by one, or how to remove an induced forest so that the remaining graph has mad reduced by two.

More generally, we gave a polynomial time algorithm which, given a graph G and a positive integer k , finds a subset S inducing a $(k - 1)$ -degenerated subgraph such that the remaining graph has mad reduced by at least k .

The existence of such partitions has direct combinatorial consequences in graph partitioning and in the theory of reconfiguration of colourings.

Our algorithm is based on flow methods.

Chapter 2

Preliminary

Here we present a list of standard definitions from graph theory, which will be used throughout the rest of this dissertation. Readers familiar with the standard nomenclature may freely omit this chapter.

A graph $G(V, E)$ is a pair consisting of a set V of vertices and a set E of edges. For a given graph H , we may refer to $V(H)$ and $E(H)$ to denote its set of vertices and edges, respectively. If the graph is clear from context, we may simply write V or E .

By default, all considered graphs are undirected, meaning that E is a set of unordered pairs of vertices. Directed graphs will be called digraphs, in this case, E is a set of ordered pairs of vertices. A digraph with no directed cycles is called a DAG.

The number of vertices $|V|$ will be denoted by n . The number of edges $|E|$ will be denoted by m .

For $A \subseteq V$, the subgraph induced by A is denoted by $G[A]$, and defined as $V(G[A]) = A$ and $E(G[A]) = \{(u, v) \in E(G) | u \in A \wedge v \in A\}$.

A set $A \subseteq V$ is called independent if there are no edges connecting vertices from A , i.e., $G[A]$ is edgeless.

For a graph G , vertices $u, v \in V(G)$ are called k -edge-connected if there exist k edge-disjoint paths between u, v . It is well known that k -edge-connectivity is an equivalence relation for any fixed k , the equivalence classes of that relation are called k -edge-connected components.

For a digraph G , vertices $u, v \in V(G)$ are strongly connected, if there exists a path from u to v and a path from v to u . Strong connectivity is also an equivalence relation, its equivalence classes are called strongly connected components.

We will use standard $\mathcal{O}$ and Ω notations, as well as $\mathcal{O}^*$, which hides polynomial factors and $\tilde{\mathcal{O}}$ which hides polylogarithmic factors.

We will consider dynamic graph problems, in which edges of a graph or digraph may be modified online. Such problems are called

- incremental, if edges are only inserted.
- decremental, if edges are only deleted.
- fully dynamic, if edges are inserted and deleted arbitrarily.

If time spent in each individual update is bounded by $\mathcal{O}(F(n, m))$, we call this worst-case update time. This contrasts with amortized update time, which bounds the average time per update over a sequence of updates.

The maximum average degree of a graph, abbreviated by $\mathbf{mad}$, is defined as:

$$\mathbf{mad}(G) := \max_{H \subseteq G, H \neq \emptyset} \frac{2|E(H)|}{|V(H)|}.$$

The degeneracy of a graph G is the smallest value d such that each subgraph $H \subseteq G$ has a vertex with degree at most d .

Chapter 3

Many-Visits TSP

Problem 3.1: Many-Visits Traveling Salesman Problem

Input:

- set V – the vertices of a graph.
- function $d : V^2 \rightarrow \mathbb{Z}_{\geq 0} \cup \{\infty\}$ – pairwise distances (or costs).
- function $k : V \rightarrow \mathbb{Z}_+$ – demands.

Output: Compressed representation of a sequence of vertices $(x_1, \dots, x_\ell)$ of length $\ell := \sum_{v \in V} k(v)$, such that:

- each $v \in V$ appears in the sequence exactly $k(v)$ times.
- the total distance (cost) $\sum_{i=1}^{\ell-1} d(x_i, x_{i+1}) + d(x_\ell, x_1)$ is minimized.

When $k \equiv 1$, the problem becomes the classical TSP, in which each city must be visited exactly once. The Many-Visits TSP (MVTSP) is a natural generalization, where some cities must be visited multiple times.

Since ℓ can be arbitrarily large, the output has to be in compressed form. Usually, it is represented as a multiplicity function $m : V^2 \rightarrow \mathbb{Z}_{\geq 0}$, describing how many times each edge is used. From this representation, one can reconstruct the whole sequence in $\mathcal{O}(\ell)$ time using a standard Eulerian tour algorithm.

The output multiplicity function $m : V^2 \rightarrow \mathbb{Z}_{\geq 0}$ must satisfy two conditions:

- Each vertex must have exactly $k(v)$ ingoing and exactly $k(v)$ outgoing edges.
- Described multigraph must be connected.

Without the connectivity constraint, the problem is solvable in polynomial time because it reduces to a minimum-cost flow problem. The difficulty lies in enforcing the connectivity.

Best existing approaches first build a connected substructure of the solution and then complete it to a full solution using a minimum-cost flow algorithm. This idea was first introduced by Cosmadakis and Papadimitriou [9]. They used an Eulerian subgraph as the connected substructure and introduced an algorithm working in time $2^{\mathcal{O}(n \log n)} + \mathcal{O}(n^3 \log \ell)$. Berger et al. [4, 3] improved that idea by using a smaller connected structure, namely outbranching (directed spanning tree, with all edges directed out of the root), and based on this, they designed two algorithms:

- $\mathcal{O}^*(5^n)$, using exponential space.
- $\mathcal{O}^*(16^n)$, using polynomial space.

3.1 Our Work

In [33] we presented the following results:

- A randomized polynomial space algorithm running in time $\mathcal{O}^*(2^n D)$, where D is the maximum distance between two cities. By using standard methods, this results in a $(1 + \epsilon)$ -approximation in time $\mathcal{O}^*(2^n \epsilon^{-1})$.
- A tight analysis of Berger et al. algorithm [4, 3] resulting in an $\mathcal{O}^*(4^n)$ running time bound.
- A new polynomial space algorithm, running in time $\mathcal{O}(7.88^n)$.

3.1.1 $\mathcal{O}^*(2^n D)$ algorithm

This algorithm is based on algebraic methods. First, by a polynomial time preprocessing step, demands $k(v)$ can be reduced to be polynomially bounded. This follows from the observation that the optimal solution cannot differ too much from the solution to the problem without the connectivity constraint. In particular, if some edge is used sufficiently many times in that solution, we can greedily include it in the connected one as well.

Then each vertex v is replaced with $k(v)$ copies $v_1, v_2, \dots, v_{k(v)}$, and each edge (u, v) is also replicated accordingly $k(u) \cdot k(v)$ times. For each copy of edge e , we introduce a unique variable x_e , and one special variable y to track the length of the solution. Now we create a slightly modified Edmond's matrix where each entry $x_{(u_i, v_j)}$ is multiplied by $y^{d(u, v)}$. Permanent of that matrix encodes all possible solutions (connected or not). Note that each solution is expressed in many monomials due to copying of the vertices, but no two monomials are the same because each entry of the matrix contains a unique variable. Moreover, the exponent of y in each monomial describes the total length of the corresponding solution. Over a field of characteristic 2, the permanent equals the determinant and can be evaluated for given values of all the variables in polynomial time. To eliminate disconnected solutions, we use the Cut & Count technique [10]. It requires considering polynomials corresponding to each subset of vertices and combining them into a single polynomial, whose evaluation takes $\mathcal{O}(2^n)$ time. The algorithm is randomized, since we assign random values to all variables x_e . The DeMillo–Lipton–Schwartz–Zippel lemma [11, 47, 42] ensures that with high

probability the monomials of minimum degree in terms of variable y do not cancel out. Finally, for fixed variables x_e , the polynomial in terms of y is reconstructed using Lagrange interpolation, which adds an additional $\mathcal{O}^*(D)$ factor to the running time. Altogether, this yields an $\mathcal{O}^*(2^n D)$ randomized algorithm.

One can drop the D term at the cost of the accuracy of the solution, namely by rounding distances, one can get an $(1 + \epsilon)$ -approximation algorithm working in time $\mathcal{O}^*(2^n \epsilon^{-1})$.

Improving the constant 2 in those results would be a major breakthrough, as it would result in improving the $\mathcal{O}^*(2^n)$ -time algorithm for DIRECTED HAMILTONIAN CYCLE, which is a 55-years-old open problem.

3.1.2 $\mathcal{O}^*(4^n)$

In this section, we do not introduce any new algorithm. We revisit the $\mathcal{O}^*(5^n)$ algorithm introduced by Berger et al. [4, 3]. Their approach iterates over all possible outdegree sequences of an outbranching. For each such sequence, the cheapest outbranching is found via dynamic programming, and the residual degree constraints are optimally satisfied in polynomial time using the minimum-cost flow algorithm. As subproblems in the dynamic programming, they consider finding the cheapest outbranching spanning a given subset of vertices with a given outdegree sequence. To solve one subproblem, the algorithm picks an arbitrary leaf (a vertex of outdegree zero) and considers all possible vertices it could be connected to. For each such choice, it uses the solution of a smaller subproblem on the subset without that leaf and with the outdegree sequence adjusted accordingly. The authors correctly infer there are $\mathcal{O}^*(4^k)$ feasible outdegree sequences for outbranching spanning a k vertex graph, and summing it up across all subsets they get $\sum_{A \subset V} \mathcal{O}^*(4^{|A|}) = \mathcal{O}^*(5^n)$ bound for the number of subproblems, and consequently time complexity is also bounded by $\mathcal{O}^*(5^n)$.

However, the algorithm is implemented in a recursive manner and may not visit all such subproblems. The main detail exploited by our refined analysis is that the algorithm always chooses to remove the leaf with the smallest index. Thanks to this, one can only consider subproblems in which the second-smallest-index leaf has an index larger than every vertex outside the current subproblem (the smallest-index leaf may arise anywhere as vertices turn into leaves). One can prove that the number of such reachable subsets and consequently the time complexity is bounded by only $\mathcal{O}^*(4^n)$.

This analysis is tight since the algorithm iterates $\Omega(4^n)$ feasible outdegree sequences on the entire vertex set V .

3.1.3 Polynomial Space $\mathcal{O}^*(7.88^n)$

In the case of polynomially bounded space, we cannot afford dynamic programming similar to the previous algorithm. Here we use Berger et al. [4, 3] polynomial space divide and conquer subroutine, which can find the cheapest outbranching for a given outdegree sequence in time $\mathcal{O}^*(4^n)$. It is worth noting that for an outdegree sequence consisting of $n - 1$ ones, finding the cheapest outbranching is essentially regular TSP, for which the best known polynomial space algorithm by Gurevich and Shelah [22] works in time $\mathcal{O}^*(4^n)$, so it is probably hard to improve the cheapest outbranching subroutine in general. Berger et al. [4, 3] algorithm simply iterates $\mathcal{O}^*(4^n)$ feasible outdegree sequences and for each of them constructs the cheapest

outbranching independently. As in the previous algorithm, the resulting partial solutions are then completed via a polynomial time minimum-cost flow routine. The total runtime is $\mathcal{O}^*(4^n) \cdot \mathcal{O}^*(4^n) = \mathcal{O}^*(16^n)$.

Our improvement comes from designing a stronger flow network that takes over part of the task of enforcing connectivity. Consequently, the cheapest outbranching subroutine will be invoked for smaller subsets of vertices and for fewer feasible outdegree sequences.

For example, consider an optimal outbranching (a subgraph of the optimal solution), and denote the number of its leaves by l . One can determine how to optimally arrange its inner vertices using the cheapest outbranching subroutine in time $\mathcal{O}^*(4^{n-l})$. Then build a flow network so that any solution recovered from its minimum-cost flow necessarily assigns at least one edge from the inner vertices to each leaf, therefore ensuring the connectivity of the entire solution. We can notice that after such trimming of one layer of leaves, the remaining part has fewer leaves. In either case, we benefit:

- For small l , the number of possible outdegree sequences decreases.
- For large l , the time spent to find the cheapest outbranching is reduced.

The exact complexity of that algorithm turns out to be

$$\mathcal{O}^*\left(\sum_{l=1}^n \binom{n}{l} \sum_{c=0}^l \binom{n-l}{c}^2 4^{n-l}\right) = \mathcal{O}^*(12.92^n).$$

We extend this idea to trimming K layers of leaves simultaneously. The downside is that we must specify the exact layer of each trimmed vertex, resulting in K^l possible configurations. Our flow network, given that division, ensures that each vertex has at least one edge coming from the previous layer, while still computing the best augmentation to the full solution.

The complexity becomes

$$\mathcal{O}^*\left(\sum_{l=1}^n \binom{n}{l} \sum_{c=0}^{\frac{l}{K}} \binom{n-l}{c}^2 (4^{n-l} + K^l)\right).$$

Analyzing this complexity shows the optimal choice is $K = 4$, leading to a polynomial space algorithm with running time $\mathcal{O}^*(7.88^n)$.

3.1.4 My contribution

- I observed that the Berger et al. algorithm works in $\mathcal{O}^*(4^n)$ time.
- I had a significant contribution in developing $\mathcal{O}(7.88^n)$ algorithm, by introducing the idea of trimming leaves from an outbranching.
- I also finalized the details of the $\mathcal{O}^*(2^n/\epsilon)$ approximation algorithm, while that part is fairly standard, we include it in the paper for completeness.
- I have written chapters 5 and 7.

Chapter 4

4-Edge-Connected Components

Problem 4.1: 4-Edge-Connected Components

Input: An undirected graph $G(V, E)$.

Output: The partition of V into 4-edge-connected components.

We can consider the problem of k -edge-connected components for any fixed k :

- For $k = 1$, the problem reduces to the standard connected components, which can be found trivially by DFS.
- For $k = 2$, it is solvable in linear time using properties of the classic `low` function defined by Hopcroft and Tarjan[23].
- For $k = 3$, the first linear time algorithm was discovered by Galil and Italiano[20] via a reduction to 3-vertex-connected components, which are computable in linear time by Hopcroft and Tarjan [24].
- For $k = 4$, prior to our work the best achieved complexity was $\mathcal{O}(m + n\alpha(n))$, combining the incremental algorithm for 4-vertex-connected components of Kanevsky et al. [25] with the reduction from 4-edge-connected components by Galil and Italiano[20].
- For $k > 4$, i.e., the general case, there was also a lot of work, most notably Dinitz and Westbrook[14] gave a data structure that maintains the k -edge-connected components of an incremental $(k - 1)$ -edge-connected graph. It processes any sequence of q online queries, each either connecting a given pair of vertices or asking whether a given pair is k -edge-connected, in total $\mathcal{O}(q + m + k^2 n \log(n/k))$ time.

Dinitz et al. [13] proved that the structure of k -edge-connected components forms a cactus graph. Moreover, for even k – and thus, for $k = 4$ – this cactus simplifies to a tree.

In [36, 37] we presented the first linear time algorithm for 4-edge-connected components. Our algorithm finds all 3-edge-cuts of the graph and constructs the corresponding tree of 4-edge-connected components.

The same result was provided independently by Georgiadis et al. [21].

Both algorithms rely on the static tree disjoint-set-union data structure of Gabow and Tarjan [19], which assumes the RAM model to perform arithmetic and bitwise operations on $\mathcal{O}(\log(n))$ -bit words in constant time. However, this assumption does not hold in weaker computation models, such as the pointer machine [45]. Georgiadis et al. [21] observed that by combining both algorithms, one can obtain a linear time algorithm that works even in the pointer machine computation model, as each of them uses the disjoint-set-union data structure in a different part of the computation.

Following these breakthroughs, the area has seen renewed activity. A linear time algorithm for 5-edge-connected components was given by Kosinas [32], suggesting the feasibility of a linear algorithm for general constant k . Finally, Korhonen [31, 30] settled the question by giving a linear time algorithm for k -edge-connected components for any fixed constant k .

4.1 Our Work

4.1.1 Motivation

Our interest in this problem came from the competitive programming world. We were looking for a challenging problem suitable for the hardest spot in online qualifications rounds of *Algorithmic Engagements* (pl. *Potyczki Algorytmiczne*). We ended up with the following one: “How many triples of edges disconnect the given graph?” After the contest, we thought that our ideas might also be of scientific interest. In the contest setting, a deterministic linear time algorithm was not required, a randomized $\mathcal{O}(n \cdot \text{polylog}(n))$ algorithm sufficed for full score, but in terms of less practical scientific perspective we were able to derandomize the algorithm, and shave all the polylogarithmic factors using, for example, the static tree disjoint-set-union data structure by Gabow and Tarjan [19]. Our paper [36] received the Best Student Paper Award at the European Symposium on Algorithms 2021, providing evidence that competitive programming insights can be valuable also from a scientific perspective.

4.1.2 The Algorithm

We begin by decomposing the graph into 3-edge-connected components. Each such component can be processed independently after replacing the remainder of the graph by a linear number of additional edges, such that the 4-edge-connected components are not affected. Hence, we can assume the input graph is 3-edge-connected.

By the max-flow min-cut theorem, if two vertices are not 4-edge-connected, there exists a 3-edge-cut separating them. Our algorithm finds all 3-edge-cuts and then reconstructs from them the partition into 4-edge-connected components.

Before presenting the deterministic algorithm, we describe a simpler randomized linear time approach. Fix an arbitrary DFS tree of the graph. For every non-tree edge, sample an independent hash value from a

field of characteristic 2. For each tree edge e , define its hash to be the sum of the hashes of all non-tree edges leaping over e . Equivalently, each non-tree edge contributes to the hash of each tree edge on the unique cycle it forms together with the DFS tree.

One can prove that edges forming any 3-edge-cut have hashes that sum to 0, and if the field is sufficiently large, the probability that there exist three edges not forming a cut and having hashes summing to 0 is bounded by a constant.

We handle 3-edge-cuts by case analysis depending on the number of tree edges they contain:

- **One tree edge.** We iterate over tree edges. Using the `low` function [23], we identify a unique candidate for one of the two remaining edges. Then we determine the potential hash of the third edge by summing the hashes of two known edges, and look it up in a hashtable storing all edge hashes (actually using a hashtable can be omitted by instead sorting all such queries using radix sort and answering them offline).
- **Two tree edges.** This case is more complicated, but DFS-style procedures still yield a constant number of candidates to check per tree edge, and these candidates provably cover all 3-edge-cuts of this type.
- **Three tree edges.** Despite considerable effort, we could not obtain an approach in the same style as above. Instead, we handle this case in a recursive manner, without deteriorating the time complexity.

We contract all non-tree edges, the remaining multigraph has $n - 1$ edges, and remains 3-connected, and consequently has at most $\frac{2n}{3}$ vertices. Moreover, all 3-tree-edge-cuts are preserved, thus we can find them recursively. Since the size of the graph shrinks geometrically, the overall running time is linear.

This is the main novel ingredient of our algorithm.

To derandomize the algorithm, we refine the procedures so that they can deterministically recover third edges for the candidates whenever the corresponding 3-edge-cut exists. The details are mainly technical, meticulous work.

To complete the algorithm, we must reconstruct the partition of V into 4-edge-connected components. Alongside this partition, we also build the tree $H(U, F)$ representing the structure of the 4-edge-connected components. Every 4-edge-connected component is represented by a vertex from U , and there is a bijection between F and the set of all 3-edge-cuts of G . The way a 3-edge-cut separates the graph G into two subgraphs exactly mirrors the way the corresponding edge splits the tree H into two subtrees. Our goal is to construct a function $\psi : V \rightarrow U$ that defines the partition of V into 4-edge-connected.

We proved that, for every edge $(u, v) \in E$, edges from F corresponding to all 3-edge-cuts containing (u, v) form a path in the tree H . Moreover, the endpoints of this path are $\psi(u)$ and $\psi(v)$. This lemma enables us to design an algorithm that traverses G and H simultaneously. When we traverse an edge in G , we traverse the corresponding path in H in parallel. Of course, at the beginning, H is unknown, but we build it interactively during the execution of the algorithm.

We begin with an arbitrary root r and fix its image $\psi(r) \in U$. Then we perform a DFS traversal of G . Whenever we traverse an edge (u, v) in G , we follow the corresponding path in H , starting at $\psi(u)$. Part of this path may have already been created, we add the missing edges, preserving their correct order (which is uniquely determined by the way the associated 3-edge-cuts split G). Finally, the endpoint of this path becomes the image $\psi(v)$. In this way, after visiting each vertex of G in the DFS traversal, we determine all values of the function ψ , which uniquely defines the 4-edge-connected components.

4.1.3 My Contribution

I proposed the main idea of handling the case of cuts consisting of three tree edges in a recursive manner. I also contributed substantially to the algorithm that reconstructs the tree structure and the partition into 4-edge-connected components from the list of 3-edge-cuts. I wrote chapters 3 and 6 and prepared figures for the entire paper.

Chapter 5

Dynamic Strongly Connected Components

Problem 5.1: Fully Dynamic Strongly Connected Components

Input: Directed graph G .

Updates:

- Adding an edge.
- Removing an edge.

Maintained information: The partition of V into strongly connected components.

Problem 5.2: Fully Dynamic Strong Connectivity

Input: Directed graph G .

Updates:

- Adding an edge.
- Removing an edge.

Maintained information: One bit of information whether G is strongly connected.

For dynamic problems, we usually have both updates and queries. However, in our case, since the size of the maintained information is relatively small, we assume that after each update, the entire maintained information is returned.

Both problems are closely related to the fully dynamic reachability problem, where we maintain a directed graph under edge insertions and deletions and support queries that ask whether there is a path between two given vertices.

Sankowski [41] gave a data structure with $\mathcal{O}(n^{1+\rho})$ worst-case update time and $\mathcal{O}(n^\rho)$ query time, where ρ satisfies $1 + 2\rho = \omega(1, \rho, 1)$. Here, $\mathcal{O}(n^{\omega(a,b,c)})$ denotes the time needed to multiply an $n^a \times n^b$ boolean matrix by an $n^b \times n^c$ boolean matrix. Van den Brand et al. [46] gave another trade-off with $\mathcal{O}(n^{\max(\epsilon_1 + \epsilon_2, \omega(1, \epsilon_1, \epsilon_2) - \epsilon_1, \omega(1, 1, \epsilon_2) - \epsilon_2)})$ update and query time for $0 \leq \epsilon_1 \leq \epsilon_2 \leq 1$. With the current state-of-the-art matrix multiplication [2], and using techniques from [46] to bound values of $\omega(a, b, c)$, we obtain $\rho < 0.5275$ and $\max(\epsilon_1 + \epsilon_2, \omega(1, \epsilon_1, \epsilon_2) - \epsilon_1, \omega(1, 1, \epsilon_2) - \epsilon_2) < 1.4057$.

Both of these data structures are Monte Carlo randomized, as they rely on the DeMillo–Lipton–Schwartz–Zippel lemma [11, 47, 42]. Van den Brand et al. [46] also proved conditional lower bounds of $\Omega(n^{1+\rho}) = \Omega(n^{1.527})$ for fully dynamic strongly connected components, and $\Omega(\max(\epsilon_1 + \epsilon_2, \omega(1, \epsilon_1, \epsilon_2) - \epsilon_1, \omega(1, 1, \epsilon_2) - \epsilon_2)) = \Omega(n^{1.405})$ lower bound for fully dynamic strong connectivity. Karczmarz et al. [26] matched the first lower bound in an amortized sense by giving an $\tilde{\mathcal{O}}(n^{1+\rho}) = \mathcal{O}(n^{1.528})$ amortized update time data structure for the fully dynamic strongly connected components problem. However, before our work, no worst-case subquadratic update time data structure was known for this problem.

For fully dynamic strong connectivity, the best previous approach was to use the reachability data structure by Sankowski [41], and ask $\mathcal{O}(n)$ queries per update to verify strong connectivity of the digraph, resulting in $\mathcal{O}(n^{1.528})$ worst-case update time.

In [27] we presented:

- An $\mathcal{O}(n^{1.528})$ worst-case update time data structure for fully dynamic strongly connected components.
- An $\mathcal{O}(n^{1.406})$ worst-case update time data structure for fully dynamic strong connectivity.

Note that these values are slightly better than those stated in the paper due to the improvements in the field of matrix multiplication [2]. Both results match the conditional lower bound implied by an extension of the OMv conjecture [46].

5.1 Our Work

5.1.1 Fully Dynamic Strongly Connected Components

We build upon the reachability data structure by Sankowski [41] with $\mathcal{O}(n^{1.528})$ update time and $\mathcal{O}(n^{0.528})$ query time.

For a given subset $T \subseteq V$, we show how to construct an auxiliary graph G_T , which allows us to maintain, for each vertex v , a bit indicating whether v is strongly connected to some vertex from T . If two vertices have different bits, they are definitely in different strongly connected components.

We appropriately sample $\mathcal{O}(\log^3(n))$ such sets T , so that the probability of encountering two vertices from distinct strongly connected components receiving identical bit vectors is negligible. Crucially, sets should have sizes of every possible order of magnitude to take care of strongly connected components of different sizes.

After each update, we propagate the change to all reachability data structures for G_T . The maintained information allows us to build a sparse digraph H with $\tilde{O}(n)$ edges, which, with high probability, has the same partition into strongly connected components as G . We then run a standard static algorithm [44] to recover the desired partition in $\mathcal{O}(m)$ time.

5.1.2 Fully Dynamic Strong Connectivity

For fully dynamic strong connectivity, we use the reachability data structure by van den Brand et al. [46] with $\mathcal{O}(n^{1.406})$ update and query time. Our reduction performs only $\tilde{O}(1)$ reachability queries per update, ensuring an overall worst-case update time of $\mathcal{O}(n^{1.406})$. Inside our data structure, we store one representative vertex for each source strongly connected component (without ingoing edges in the DAG of strongly connected components), and analogously, one representative vertex for each sink strongly connected component (without outgoing edges).

One edge insertion may contract a whole path of strongly connected components, and one edge deletion may break one connected component into a long path of smaller ones. However, only the first and the last vertex of those paths are of interest to us, so the maintained information about the digraph requires only small-scale adjustments.

We maintain $\mathcal{O}(\log n)$ reachability data structures for special auxiliary digraphs to track which source/sink strongly connected components disappear or appear after each update. Conveniently, when a new sink/source strongly connected component is formed, we can use one of the endpoints of the just-deleted edge as its new representative.

The digraph is strongly connected if and only if the representative of the unique sink strongly connected component can reach the representative of the unique source connected component, which can be checked by one reachability query in G .

It is worth noting that this reduction is deterministic. If the reachability data structure were derandomized, our data structure would also become deterministic.

Our data structure can be extended to maintain the minimum strong connectivity augmentation of a digraph [17].

5.1.3 Planar Strongly Connected Components

We also have a follow-up paper [28] studying fully dynamic strongly connected components in planar digraphs. There, we maintain an implicit representation of strongly connected components in $\tilde{O}(n^{\frac{6}{7}})$ worst-case update time. To our knowledge, this is the first sublinear update data structure for any major subclass of digraphs. Since my contribution to that paper was minor, I do not include its details in this dissertation.

5.1.4 My Contribution

I proposed the $\mathcal{O}(n^{1.406})$ worst-case data structure for the fully dynamic strong connectivity problem.

5.2 Future Research

For the problems considered in our paper, further asymptotic improvements are unlikely, as results match known conditional lower bounds [46]. However, there are related problems that may be promising for future research:

- Maintaining the property that the number of strongly connected components is at most 2. In our framework for strong connectivity, if there is only one sink and only one source connected component, we would need to check whether all the other vertices lie within these two components.
- Maintaining only the number of strongly connected components, without an exact partition of the vertices.

Conditional lower bounds [46] for both these problems are $\mathcal{O}(n^{1.406})$. However, there is no data structure with update time better than $\mathcal{O}(n^{1.528})$, which simply uses our solution for explicitly maintaining all the strongly connected components.

Another promising direction is studying the problems for restricted classes of digraphs, as we did in [28] for planar digraphs.

Chapter 6

Maximum Average Degree

A graph parameter F is called partitionable if for any graph G and any real numbers a and b such that $F(G) < a + b$, the vertex set can be partitioned into $A \uplus B$ in such a way that $F(G[A]) < a$ and $F(G[B]) < b$. It is known that degeneracy [35], maximum degree [34], and treewidth [12] are partitionable parameters. Hendrey, Norin and Wood posed the question whether the maximum average degree (**mad**) is also partitionable, as one of the open problems for the Barbados workshop [1].

This question is also relevant to the theory of reconfiguration graphs of colourings, motivated by Cereceda's conjecture [7], which states that reconfiguration graphs of $(k + 2)$ -colourings of k -degenerate graphs have a quadratic diameter. In particular, Eiben et al. [16] explicitly asked whether for any graph G one can remove an independent set of vertices such that the **mad** of the remaining graph decreases by at least one, which is the special case of partitionability of **mad** in case of $a = 1$.

We make a step towards proving partitionability of **mad** by solving the following problem.

Problem 6.1: Partition of Maximum Average Degree

Input:

- An undirected graph G .
- A positive integer k such that $\text{mad}(G) \geq k$.

Output: Set $S \subseteq V$ such that:

- $G[S]$ is $(k - 1)$ -degenerate.
- $\text{mad}(G[V \setminus S]) < \text{mad}(G) - k$.

In [38], we showed an algorithm solving this problem in polynomial time. The main takeaway from our algorithm is a theorem stating that such a set S always exists regardless of the input. This implies **mad** is partitionable in cases $a = 1$ and $a = 2$, since:

- for $k = 1$, the set S is independent set, so $\text{mad}(G[S]) = 0 < 1$.
- for $k = 2$, the set S induces a forest, so $\text{mad}(G[S]) < 2$.

Our theorem has many combinatorial consequences, most of which were known before, nevertheless, our result can be viewed as a nice way to unify all of them, and in some cases, we improve the state of the art.

6.1 Our Work

6.1.1 The Main Theorem

The main idea is to analyze the flow network used to compute mad . It turns out that finding the maximum flow not only verifies the numerical value of mad , but also reveals which vertices are crucial for reaching this value. We show how to algorithmically extract such vertices so that they induce a $(k - 1)$ -degenerated graph, and then we reconstruct a smaller flow network for the remaining part of the graph, which certifies that the remaining graph $G[V \setminus S]$ has mad reduced by at least k .

6.1.2 Consequences

Our result directly implies many known partition theorems as simple corollaries. For example:

- For a planar graph $G(V, E)$, there exists a partition $V = A \uplus B \uplus C$, such that $G[A]$, $G[B]$, $G[C]$ are forests [8, 39].
- For a planar graph $G(V, E)$ without triangles, there exists a partition $V = A \uplus B$, such that $G[A]$, $G[B]$ are forests [40].
- For a planar graph $G(V, E)$ with the length of the shortest cycle at least 6, there exists a partition $V = A \uplus B$, such that $G[A]$ is a forest and B is an independent set [6].

We also strengthen the following theorem of Dross et al. [15]:

Theorem 6.1 *Let M be a real number such that $M < 3$. Let $d \geq 0$ be an integer and let G be a graph with $\text{mad}(G) < M$. If $d \geq \frac{2}{3-M} - 2$, then $V(G)$ can be partitioned into $A \uplus B$ such that $G[A]$ is an independent set and $G[B]$ is a forest **with maximum degree at most d** .*

We prove the following stronger version:

Theorem 6.2 *Let M be a real number such that $M < 3$. Let $d \geq 0$ be an integer and let G be a graph with $\text{mad}(G) < M$. If $d \geq \frac{2}{3-M} - 2$, then $V(G)$ can be partitioned into $A \uplus B$ such that $G[A]$ is an independent set and $G[B]$ is a forest **whose connected components have size at most $d + 1$** .*

As another novel consequence, we obtain a subexponential bound on the diameter of reconfiguration graphs of generalized colourings of graphs with bounded $\mathbf{mad}$. This is achieved by adapting the framework by Feghali et al. [18], originally developed for graphs of bounded maximum degree.

6.1.3 My Contribution

I proposed the main algorithm and wrote chapter 3 of the paper.

Bibliography

- [1] *Open problems for the Barbados graph theory workshop 2019*. Edited by L. Cook and S. Spirkl, Available at: <https://sites.google.com/site/sophiespirkl/open-problems/2019-open-problems-for-the-barbados-graph-theory-workshop>.
- [2] J. Alman, R. Duan, V. Vassilevska Williams, Y. Xu, Z. Xu, and R. Zhou. More asymmetry yields faster matrix multiplication. In Y. Azar and D. Panigrahi, editors, *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2025, New Orleans, LA, USA, January 12-15, 2025*, pages 2005–2039. SIAM, 2025.
- [3] A. Berger, L. Kozma, M. Mnich, and R. Vincze. Time- and space-optimal algorithms for the many-visits TSP. *CoRR*, abs/1804.06361, 2018.
- [4] A. Berger, L. Kozma, M. Mnich, and R. Vincze. A time- and space-optimal algorithm for the many-visits TSP. In T. M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1770–1782. SIAM, 2019.
- [5] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.
- [6] O. V. Borodin and A. Glebov. On the partition of a planar graph of girth 5 into an empty and an acyclic subgraph. *Diskretnyj Analiz i Issledovanie Operatsij. Seriya 1*, 01 2001.
- [7] L. Cereceda. Mixing graph colourings. PhD thesis, London School of Economics, 2007.
- [8] G. Chartrand, H. V. Kronk, and C. E. Wall. The point-arboricity of a graph. *Israel Journal of Mathematics*, 6(2):169–175, Apr 1968.
- [9] S. S. Cosmadakis and C. H. Papadimitriou. The traveling salesman problem with many visits to few cities. *SIAM J. Comput.*, 13(1):99–108, 1984.
- [10] M. Cygan, J. Nederlof, M. Pilipczuk, M. Pilipczuk, J. M. M. van Rooij, and J. O. Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In *FOCS*, pages 150–159. IEEE Computer Society, 2011.
- [11] R. A. DeMillo and R. J. Lipton. A probabilistic remark on algebraic program testing. *Inf. Process. Lett.*, 7:193–195, 1978.

- [12] G. Ding, B. Oporowski, D. P. Sanders, and D. Vertigan. Partitioning graphs of bounded tree-width. *Combinatorica*, 18(1):1–12, Jan 1998.
- [13] Y. Dinitz, A. V. Karzanov, and M. V. Lomonosov. On the structure of a family of minimum weighted cuts in a graph. *Studies in Discrete Optimization*, page 290–306, 1976.
- [14] Y. Dinitz and J. R. Westbrook. Maintaining the Classes of 4-Edge-Connectivity in a Graph On-Line. *Algorithmica*, 20(3):242–276, Mar 1998.
- [15] F. Dross, M. Montassier, and A. Pinlou. Partitioning sparse graphs into an independent set and a forest of bounded degree. *Electr. J. Comb.*, 25(1):P1.45, 2018.
- [16] E. Eiben and C. Feghali. Toward Cereceda’s conjecture for planar graphs. *Journal of Graph Theory*, 94(2):267–277, 2020.
- [17] K. P. Eswaran and R. E. Tarjan. Augmentation problems. *SIAM J. Comput.*, 5(4):653–665, 1976.
- [18] C. Feghali, M. Johnson, and D. Paulusma. A reconfigurations analogue of Brooks’ theorem and its consequences. *Journal of graph theory.*, 83(4):340–358, December 2016.
- [19] H. N. Gabow and R. E. Tarjan. A Linear-Time Algorithm for a Special Case of Disjoint Set Union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985.
- [20] Z. Galil and G. F. Italiano. Reducing edge connectivity to vertex connectivity. *SIGACT News*, 22(1):57–61, Mar. 1991.
- [21] L. Georgiadis, G. F. Italiano, and E. Kosinas. Computing the 4-edge-connected components of a graph in linear time. In P. Mutzel, R. Pagh, and G. Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPICs*, pages 47:1–47:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
- [22] Y. Gurevich and S. Shelah. Expected computation time for Hamiltonian path problem. *SIAM J. Comput.*, 16(3):486–502, June 1987.
- [23] J. Hopcroft and R. Tarjan. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Commun. ACM*, 16(6):372–378, June 1973.
- [24] J. Hopcroft and R. Tarjan. Dividing a Graph into Triconnected Components. *SIAM Journal on Computing*, 2(3):135–158, 1973.
- [25] A. Kanevsky, R. Tamassia, G. Di Battista, and J. Chen. On-Line Maintenance of the Four-Connected Components of a Graph. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 793–801, 1991.
- [26] A. Karczmarz, A. Mukherjee, and P. Sankowski. Subquadratic dynamic path reporting in directed graphs against an adaptive adversary. *STOC 2022*, pages 1643–1656. ACM, 2022.
- [27] A. Karczmarz and M. Smulewicz. On fully dynamic strongly connected components. In I. L. Gørtz, M. Farach-Colton, S. J. Puglisi, and G. Herman, editors, *31st Annual European Symposium on Algorithms, ESA 2023, September 4-6, 2023, Amsterdam, The Netherlands*, volume 274 of *LIPICs*, pages 68:1–68:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

- [28] A. Karczmarz and M. Smulewicz. Fully dynamic strongly connected components in planar digraphs. In K. Bringmann, M. Grohe, G. Puppis, and O. Svensson, editors, *51st International Colloquium on Automata, Languages, and Programming, ICALP 2024, July 8-12, 2024, Tallinn, Estonia*, volume 297 of *LIPIcs*, pages 95:1–95:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.
- [29] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003.
- [30] T. Korhonen. Linear-time algorithms for k-edge-connected components, k-lean tree decompositions, and more. *CoRR*, abs/2411.02658, 2024.
- [31] T. Korhonen. Linear-time algorithms for k-edge-connected components, k-lean tree decompositions, and more. In M. Koucký and N. Bansal, editors, *Proceedings of the 57th Annual ACM Symposium on Theory of Computing, STOC 2025, Prague, Czechia, June 23-27, 2025*, pages 111–119. ACM, 2025.
- [32] E. Kosinas. Computing the 5-edge-connected components in linear time. In D. P. Woodruff, editor, *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*, pages 1887–2119. SIAM, 2024.
- [33] L. Kowalik, S. Li, W. Nadara, M. Smulewicz, and M. Wahlström. Many visits TSP revisited. In F. Grandoni, G. Herman, and P. Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, pages 66:1–66:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- [34] L. Lovász. On decomposition of graphs. *Studia Sci. Math. Hungar.*, 1:237–238, 1966.
- [35] P. Mihók. Minimal reducible bounds for the class of k-degenerate graphs. *Discrete Mathematics*, 236(1):273 – 279, 2001. Graph Theory.
- [36] W. Nadara, M. Radecki, M. Smulewicz, and M. Sokolowski. Determining 4-edge-connected components in linear time. In P. Mutzel, R. Pagh, and G. Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 71:1–71:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
- [37] W. Nadara, M. Radecki, M. Smulewicz, and M. Sokolowski. Determining 4-edge-connected components in linear time. *CoRR*, abs/2105.01699, 2021.
- [38] W. Nadara and M. Smulewicz. Decreasing the maximum average degree by deleting an independent set or a $\$d\$$ -degenerate subgraph. *Electron. J. Comb.*, 29(1), 2022.
- [39] K. S. Poh. On the linear vertex-arboricity of a planar graph. *Journal of Graph Theory*, 14(1):73–75, 1990.
- [40] A. Raspaud and W. Wang. On the vertex-arboricity of planar graphs. *European Journal of Combinatorics*, 29(4):1064 – 1075, 2008. Homomorphisms: Structure and Highlights.
- [41] P. Sankowski. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *45th Symposium on Foundations of Computer Science FOCS 2004*, pages 509–517. IEEE Computer Society, 2004.

- [42] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.
- [43] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51(3):400–403, 1995.
- [44] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
- [45] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. Syst. Sci.*, 18(2):110–127, 1979.
- [46] J. van den Brand, D. Nanongkai, and T. Saranurak. Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019*, pages 456–480. IEEE Computer Society, 2019.
- [47] R. Zippel. Probabilistic algorithms for sparse polynomials. In *Proc. International Symposium on Symbolic and Algebraic Computation*, volume 72 of *LNCS*, pages 216–226, 1979.

Many Visits TSP Revisited

Łukasz Kowalik 

Institute of Informatics, University of Warsaw, Poland
<https://www.mimuw.edu.pl/~kowalik/>
kowalik@mimuw.edu.pl

Shaohua Li

Institute of Informatics, University of Warsaw, Poland
shaohua.li@mimuw.edu.pl

Wojciech Nadara

Institute of Informatics, University of Warsaw, Poland
w.nadara@mimuw.edu.pl

Marcin Smulewicz

Institute of Informatics, University of Warsaw, Poland
m.smulewicz@mimuw.edu.pl

Magnus Wahlström

Royal Holloway, University of London, UK
Magnus.Wahlstrom@rhul.ac.uk

Abstract

We study the MANY VISITS TSP problem, where given a number $k(v)$ for each of n cities and pairwise (possibly asymmetric) integer distances, one has to find an optimal tour that visits each city v exactly $k(v)$ times. The currently fastest algorithm is due to Berger, Kozma, Mnich and Vincze [SODA 2019, TALG 2020] and runs in time and space $\mathcal{O}^*(5^n)$. They also show a polynomial space algorithm running in time $\mathcal{O}(16^{n+o(n)})$. In this work, we show three main results:

- A randomized polynomial space algorithm in time $\mathcal{O}^*(2^n D)$, where D is the maximum distance between two cities. By using standard methods, this results in a $(1 + \epsilon)$ -approximation in time $\mathcal{O}^*(2^n \epsilon^{-1})$. Improving the constant 2 in these results would be a major breakthrough, as it would result in improving the $\mathcal{O}^*(2^n)$ -time algorithm for DIRECTED HAMILTONIAN CYCLE, which is a 50 years old open problem.
- A tight analysis of Berger et al.'s exponential space algorithm, resulting in an $\mathcal{O}^*(4^n)$ running time bound.
- A new polynomial space algorithm, running in time $\mathcal{O}(7.88^n)$.

2012 ACM Subject Classification Theory of computation $\rightarrow$ Graph algorithms analysis; Theory of computation $\rightarrow$ Parameterized complexity and exact algorithms

Keywords and phrases many visits traveling salesman problem, exponential algorithm

Digital Object Identifier 10.4230/LIPIcs.ESA.2020.66

Funding This workshop was supported by a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 714704 (PI: Marcin Pilipczuk).

Łukasz Kowalik: Supported by ERC Starting Grant TOTAL (Grant Agreement No 677651).

Shaohua Li: Supported by ERC Starting Grant CUTACOMBS (Grant Agreement No 714704).

Wojciech Nadara: Supported by ERC Starting Grant CUTACOMBS (Grant Agreement No 714704).

Marcin Smulewicz: Supported by ERC Starting Grant TOTAL (Grant Agreement No 677651).

Acknowledgements The research leading to the results presented in this paper was partially carried out during the Parameterized Algorithms Retreat of the University of Warsaw, PARUW 2020, held in Krynica-Zdrój in February 2020.



© Łukasz Kowalik, Shaohua Li, Wojciech Nadara, Marcin Smulewicz, and Magnus Wahlström;
licensed under Creative Commons License CC-BY
28th Annual European Symposium on Algorithms (ESA 2020).

Editors: Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders; Article No. 66; pp. 66:1–66:22



Leibniz International Proceedings in Informatics

LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

In the MANY VISITS TSP (MVTSP) we are given a set V of n vertices, with pairwise distances (or costs) $d : V^2 \rightarrow \mathbb{Z}_{\geq 0} \cup \{\infty\}$. We are also given a function $k : V \rightarrow \mathbb{Z}_+$. A valid tour of length ℓ is a sequence of vertices $(x_1, \dots, x_\ell)$, where $\ell = \sum_{v \in V} k(v)$, such that each $v \in V$ appears in the sequence exactly $k(v)$ times. The cost of the tour is $\sum_{i=1}^{\ell-1} d(x_i, x_{i+1}) + d(x_\ell, x_1)$. Our goal is to find a valid tour with minimum cost.

MANY VISITS TSP is a natural generalization of the classical (asymmetric) TRAVELING SALESMAN PROBLEM (TSP), which corresponds to the case when $k(v) = 1$ for every vertex v . Similarly as its special case, MVTSP arises with a variety of applications, including scheduling [24, 17, 6, 27, 12], computational geometry [20] and parameterized complexity [21].

1.1 Related work

The standard dynamic programming for TSP of Bellman [1], Held and Karp [16] running in time $\mathcal{O}^*(2^n)$ can be easily generalized to MVTSP resulting in an algorithm with the running time of $\mathcal{O}^*(\prod_{v \in V} (k(v) + 1))$, as noted by Psaraftis [24]. A breakthrough came in the work of Cosmadakis and Papadimitriou [7] who presented an algorithm running in time $2^{\mathcal{O}(n \log n)} + \mathcal{O}(n^3 \log \ell)$ and space $2^{\mathcal{O}(n \log n)}$, thus essentially removing the dependence on the function k from the bound (the $\log \ell$ factor can be actually skipped if we support the original algorithm with a today's state-of-the-art minimum cost flow algorithm). This may be surprising since the *length* of the output sequence is ℓ . However, beginning from the work of Cosmadakis and Papadimitriou we consider MVTSP with compressed output, namely the output is a multiplicity function which encodes the number of times every edge is visited by the tour. By using a standard Eulerian tour algorithm we can compute an explicit tour from this output.

The crux of the approach of Cosmadakis and Papadimitriou [7] was an observation that every solution can be decomposed to a minimal connected spanning Eulerian subgraph (which enforces connectivity of the solution) and a subgraph satisfying appropriate degree constraints (which completes the tour so that the numbers of visits agree). Moreover, once we guess the degree sequence δ of the Eulerian subgraph, our task splits into two separate tasks: finding a cheapest minimal connected Eulerian subgraph consistent with δ (which is computationally hard) and finding a cheapest subgraph satisfying the degree constraints (which can be solved in polynomial time by a reduction to minimum cost flow).

Yet another breakthrough came only recently, namely Berger, Kozma, Mnich and Vincze [3, 2] improved the running time to $\mathcal{O}^*(5^n)$. Their main contribution is an idea that it is more convenient to use outbranchings (i.e. spanning trees oriented out of the root) to force connectivity of the solution. The result of Berger et al. is the first algorithm for MVTSP which is optimal assuming Exponential Time Hypothesis (ETH) [18], i.e., there is no algorithm in time $2^{o(n)}$, unless ETH fails. Moreover, by applying the divide and conquer approach of Gurevich and Shelah [15] they design a polynomial space algorithm, running in time $\mathcal{O}(16^{n+o(n)})$.

1.2 Our results

In this work, we take the next step in exploration of the MANY VISITS TSP problem: we aim at algorithms which are optimal at a more fine grained level, namely with running times of the form $\mathcal{O}(c^n)$, such that an improvement to $\mathcal{O}((c - \epsilon)^n)$ for any $\epsilon > 0$ meets a kind of natural barrier, for example contradicts the Strong Exponential Time Hypothesis (SETH) [19] or the Set Cover Conjecture (SCC) [8]. Our main result is the following theorem.

► **Theorem 1.1.** *There is a randomized algorithm that solves MANY VISITS TSP in time $\mathcal{O}^*(2^n D)$ and polynomial space, where $D = \max\{d(u, v) : u, v \in V, d(u, v) \neq \infty\}$. The algorithm returns a minimum weight solution with constant probability.*

The natural barrier in this case is connected with DIRECTED HAMILTONICITY, the problem of determining if a directed graph contains a Hamiltonian cycle. Indeed, this is a special case of MANY VISITS TSP with $D = 1$, so an improvement to $\mathcal{O}^*(1.99^n D)$ in Theorem 1.1 would result in an algorithm in time $\mathcal{O}^*(1.99^n)$ for DIRECTED HAMILTONICITY. While it is not known whether such an algorithm contradicts SETH or SCC, the question about its existence is a major open problem which in the last 58 years has seen some progress only for special graph classes, like bipartite graphs [4, 9].

At the technical level, Theorem 1.1 uses the so-called algebraic approach and relies on two key insights. The first one is to enforce connectivity not by guessing a spanning connected subgraph as in the previous works, but by applying the Cut and Count approach of Cygan et al [10]. The second insight is to satisfy the degree constraints using the Tutte matrix [26, 22].

By using standard rounding techniques, we are able to make the algorithm from Theorem 1.1 somewhat useful even if the maximum distance D is large. Namely, we prove the following.

► **Theorem 1.2.** *For any $\epsilon > 0$ there is a randomized $(1 + \epsilon)$ -approximation algorithm that solves MANY VISITS TSP in $\mathcal{O}^*(2^{n\epsilon^{-1}})$ time and polynomial space.*

In Theorems 1.1 and 1.2 the better exponential dependence in the running time was achieved at the cost of sacrificing an $\mathcal{O}(D)$ factor in the running time, or the optimality of the solution. What if we do not want to sacrifice anything? While we are not able to get a $\mathcal{O}^*(2^n)$ algorithm yet, we are able to report a progress compared to the algorithm of Berger et al. in time $\mathcal{O}^*(5^n)$. In fact we do not show a new algorithm but we provide a refined analysis of the previous one. The new analysis is tight (up to a polynomial factor).

► **Theorem 1.3.** *There is an algorithm that solves MANY VISITS TSP in time and space $\mathcal{O}^*(4^n)$.*

In short, Berger et al.'s polyspace $\mathcal{O}^*(16^{n+o(n)})$ time algorithm iterates through all $\mathcal{O}(4^n)$ degree sequences of an outbranching, finds the cheapest outbranching for each sequence in time $\mathcal{O}(4^{n+o(n)})$, and completes it to satisfy the degree constraints using a polynomial time flow computation. Note that it is hard to speed up the cheapest outbranching routine, because for the sequence of $n - 1$ ones and one zero we get essentially the TSP, for which the best known polynomial space algorithm takes time $\mathcal{O}(4^{n+o(n)})$ [15]. However, we are still able to get a significant speed up of their algorithm, roughly, by using a more powerful minimum cost flow network, which allows for computing the cheapest outbranchings in smaller subgraphs.

► **Theorem 1.4.** *There is an algorithm that solves MANY VISITS TSP in time $\mathcal{O}^*(7.88^n)$ and polynomial space.*

Organization of the paper. In Section 3 we show that, essentially, using a polynomial time preprocessing step we can reduce an instance of MANY VISITS TSP to an equivalent one but with demands k bounded by $\mathcal{O}(n^2)$. This reduction is a crucial prerequisite for Section 4 where we prove Theorem 1.1. Next, in Section 5 we prove Theorem 1.3 and in Section 6 we prove Theorem 1.4. We note that in these two sections we do not need the reduction from Section 3, however, in practice, applying it should speed-up the flow computations used in both algorithms described there. Finally, in Section 7 we show Theorem 1.2 and we discuss further research in Section 8.

2 Preliminaries

We use Iverson bracket, i.e., if α is a logical proposition, then the expression $[\alpha]$ evaluates to 1 when α is true and 0 otherwise.

For two integer-valued functions f, g on the same domain D , we write $f \leq g$ when $f(x) \leq g(x)$ for every $x \in D$. Similarly, $f + g$ (resp. $f - g$) denote the pointwise sum (difference) of f and g . This generalizes to functions on different domains D_f, D_g by extending the functions to $D_f \cup D_g$ so that the values outside the original domain are 0.

For a cost function $d : V^2 \rightarrow \mathbb{Z}_{\geq 0} \cup \{\infty\}$, and a multiplicity function $m : V^2 \rightarrow \mathbb{Z}_{\geq 0}$ we denote the cost of m as $d(m) = \sum_{u,v \in V^2} d(u,v)m(u,v)$.

Multisets. Recall that a *multiset* A can be identified by its *multiplicity* function $m_A : U \rightarrow \mathbb{Z}_{\geq 0}$, where U is a set. We write $e \in A$ when $e \in U$ and $m_A(e) > 0$. Consider two multisets A and B . We write $A \subseteq B$ when for every $e \in A$ we have $e \in B$ and $m_A(e) \leq m_B(e)$. Also, $A = B$ when $A \subseteq B$ and $B \subseteq A$. Assume w.l.o.g. that m_A and m_B have the same domain U . Operations on multisets are defined by the corresponding multiplicities as follows: for every $e \in U$, we have $m_{A \cup B}(e) = \max\{m_A(e), m_B(e)\}$, $m_{A \cap B}(e) = \min\{m_A(e), m_B(e)\}$, $m_{A \setminus B}(e) = \max\{m_A(e) - m_B(e), 0\}$, $m_{A \triangle B}(e) = m_{(A \setminus B) \cup (B \setminus A)} = |m_A(e) - m_B(e)|$. This notation extends to the situation when A or B is a set, by using the indicator function $m_A(e) = [e \in A]$.

Directed graphs. Directed graphs (also called digraphs) in this paper can have multiple edges and multiple loops, so sets $E(G)$ will in fact be multisets. We call a directed graph *simple* if it has no multiple edges or loops. We call it *weakly simple* if it has no multiple edges or multiple loops (but single loops are allowed). For a digraph G by $G^\downarrow$ we denote the *support* of G , i.e., the weakly simple digraph on the vertex set $V(G)$ such that $E(G^\downarrow) = \{(u,v) \mid G \text{ has an edge from } u \text{ to } v\}$.

Given a digraph $G = (V, E)$ we define its *multiplicity function* $m_G : V^2 \rightarrow \mathbb{Z}_{\geq 0}$ as the multiplicity function of its edge multiset, i.e., for any pair $u, v \in V$, we put $m_G(u, v) = m_E((u, v))$. Conversely, for a function $m : V^2 \rightarrow \mathbb{Z}_{\geq 0}$ we define the *thick graph* $G_m = (V, E)$ so that $m_G = m$. Abusing notation slightly, we will identify m and G_m , e.g., we can say that m is strongly connected, contains a subgraph, etc.

We call a directed graph *connected* if the underlying undirected graph is connected. Similarly, a *connected component* of a digraph G is a subgraph of G induced by a vertex set of a connected component of the underlying undirected graph.

For a graph G (directed or undirected) and a subset $X \subseteq V(G)$, by $G[X]$ we denote the subgraph induced by X .

Solutions. The following observation follows easily from known properties of Eulerian digraphs.

► **Observation 2.1.** *MANY VISITS TSP has a tour of cost c if and only if there is a multiplicity function $m_G : V^2 \rightarrow \mathbb{Z}_{\geq 0}$ of cost c such that (i) for every $v \in V$, $\sum_{w \in V} m(v, w) = \sum_{w \in V} m(w, v) = k(v)$ and (ii) m contains a spanning connected subgraph.*

Thanks to Observation 2.1, in the remainder of this paper we refer to multiplicity functions as solutions of MVTSP (and some related problems which we are going to define). By standard arguments, the multiplicity function can be transformed to a tour in time $\mathcal{O}(\ell)$. Moreover, Grigoriev and Van de Klundert [14] describe an algorithm which transforms it to a compressed representation of the tour in time $\mathcal{O}(n^4 \log \ell)$.

Out-trees. An *out-tree* is the digraph obtained from a rooted tree by orienting all edges away from the root. If an out-tree T is a subgraph of a directed graph G and additionally T spans the whole vertex set $V(G)$ we call T an *outbranching*. The sequence $\{\text{outdeg}_T(v)\}_{v \in V(T)}$ is called the *outdegree sequence* of T . Consider a set of vertices $X \subseteq V$, $|X| \geq 2$.

► **Lemma 2.2** (Berger et al. [3], Lemma 2.4). *A sequence of nonnegative integers $\{d_v\}_{v \in X}$ is an outdegree sequence of an out-tree spanning X and rooted at $r \in X$ if and only if (i) $d_r \geq 1$ and (ii) $\sum_{v \in X} d_v = |X| - 1$.*

A sequence $\{d_v\}_{v \in X}$ that satisfies (i) and (ii) will be called an *out-tree sequence rooted at r* , or *outbranching sequence rooted at r* when additionally $X = V$. A δ -out-tree means any subtree spanning X with outdegree sequence δ .

3 Reduction to small demands

Consider the following problem, for a family of simple digraphs $\mathcal{F}$.

FIXED DEGREE $\mathcal{F}$ -SUBGRAPH

Input: $d : V^2 \rightarrow \mathbb{Z}_{\geq 0} \cup \{\infty\}$, $\text{in}, \text{out} : V \rightarrow \mathbb{Z}_{\geq 0}$

Question: Find a function $m : V^2 \rightarrow \mathbb{Z}_{\geq 0}$ such that

- (i) G_m contains a member of $\mathcal{F}$ as a spanning subgraph,
- (ii) for every $v \in V$ we have $\text{in}(v) = \text{indeg}_{G_m}(v)$ and $\text{out}(v) = \text{outdeg}_{G_m}(v)$, and so as to minimize the value of $d(m) = \sum_{v,w \in V} d(v,w)m(v,w)$.

In this paper, we will consider two versions of the problem: when $\mathcal{F}$ is the family of all oriented trees, called **FIXED DEGREE CONNECTED SUBGRAPH**, and when $\mathcal{F}$ is the family of all out-trees with a fixed root r , called **FIXED DEGREE SUBGRAPH WITH OUTBRANCHING**. The role of $\mathcal{F}$ is to force connectivity of the instance. Other choices for $\mathcal{F}$ can also be interesting, for example Cosmadakis and Papadimitriou [7] consider the family of minimal Eulerian digraphs.

The goal of this section is to show that, essentially, using a polynomial time preprocessing step we can reduce an instance of **FIXED DEGREE $\mathcal{F}$ -SUBGRAPH** to an equivalent one but with demands in, out bounded by $O(n^2)$.

When considering the instance of **FIXED DEGREE $\mathcal{F}$ -SUBGRAPH** we will use the notation $n = |V|$ and $\ell = \sum_{v \in V} \text{in}(v)$. (Clearly, we can assume that also $\ell = \sum_{v \in V} \text{out}(v)$, for otherwise there is no solution.)

Observe that if the image of d is $\{0, +\infty\}$ we get the natural unweighted version, where we are given a graph with edge set $d^{-1}(0)$ and the goal is to decide if one can choose multiplicities of the edges so that the resulting digraph contains a member of $\mathcal{F}$ and its in- and outdegrees match the demands of in and out .

The following observation follows by standard properties of Eulerian cycles in digraphs and the fact that every strongly connected graph contains an outbranching rooted at arbitrary vertex.

► **Observation 3.1.** *MANY VISITS TSP is a special case of both **FIXED DEGREE CONNECTED SUBGRAPH** and **FIXED DEGREE SUBGRAPH WITH OUTBRANCHING** with $\text{in}(v) = \text{out}(v) = k(v)$ for every vertex $v \in V$.*

In the following lemma, we consider the relaxed problem **FIXED DEGREE SUBGRAPH**, defined exactly as **FIXED DEGREE $\mathcal{F}$ -SUBGRAPH**, but dropping the constraint that solutions must contain a member of $\mathcal{F}$. In what follows, $s_n(\mathcal{F}) = \max_{G \in \mathcal{F}, |V(G)|=n} |E(G)|$. (Note that in applications we consider in this work $\mathcal{F}$ is a family of oriented spanning trees, so $s_n(\mathcal{F}) = n - 1$.)

► **Lemma 3.2.** *Fix an input instance $d : V^2 \rightarrow \mathbb{Z}_{\geq 0} \cup \{\infty\}$, $\text{in}, \text{out} : V^2 \rightarrow \mathbb{Z}_{\geq 0}$. For every optimal solution r of FIXED DEGREE SUBGRAPH there is an optimal solution c' of FIXED DEGREE $\mathcal{F}$ -SUBGRAPH such that for every $u, v \in V$*

$$|r(u, v) - c'(u, v)| \leq s_{|V|}(\mathcal{F}).$$

Before we proceed to a formal proof of Lemma 3.2, let us sketch an intuition behind it. We pick an optimal solution c of FIXED DEGREE $\mathcal{F}$ -SUBGRAPH and let $B \in \mathcal{F}$ be a spanning subgraph in G_c . The symmetric difference between $E(G_r)$ and $E(G_c)$ can be decomposed into “alternating” cycles. It suffices to alternate $|E(B) \setminus E(G_r)| \leq s_n(\mathcal{F})$ of them to enforce containing B . If we alternated *all* the cycles, we would get the cost of exactly $d(c)$, but because of the optimality of r , alternating any of the cycles does not decrease the cost of the solution. Hence alternating a subset of them cannot make the cost bigger than $d(c)$.

Proof. Let c be an arbitrary optimal solution of FIXED DEGREE $\mathcal{F}$ -SUBGRAPH and let B be an arbitrary graph from $\mathcal{F}$ which is a spanning subgraph of G_c . Our plan is to build an optimal solution c' of FIXED DEGREE $\mathcal{F}$ -SUBGRAPH which contains B and does not differ too much from r .

Define multisets $A_c = E(G_c) \setminus E(G_r)$, $A_r = E(G_r) \setminus E(G_c)$ and $A = A_c \cup A_r = E(G_c) \triangle E(G_r)$. In what follows, by an *alternating cycle* we mean an even cardinality set of edges

$$\{(v_0, v_1), (v_2, v_1), (v_2, v_3), (v_4, v_3) \dots, (v_{2\ell-2}, v_{2\ell-1}), (v_0, v_{2\ell-1})\},$$

where edges come alternately from A_c and A_r . Note that an alternating cycle is not really a directed cycle, it is just an orientation of a simple undirected cycle.

Note that for every vertex $v \in V$, among the edges in A that enter (resp. leave) v the number of edges from A_c is the same as the number of edges from A_r (counted with corresponding multiplicities), since both c and r satisfy the degree constraints for the same instance. It follows that A can be decomposed into a multiset $\mathcal{C}$ of alternating simple cycles, i.e.,

$$m_A = \sum_{C \in \mathcal{C}} m_C,$$

where $m_C : V^2 \rightarrow \mathbb{Z}_{\geq 0}$ and for each pair $u, v \in V$ we have $m_C(u, v) = [(u, v) \in C] \cdot m_C(C)$. To clarify, we note that the sum above is over all cycles in $\mathcal{C}$, and not over all copies of cycles.

Denote $B^+ = E(B) \setminus E(G_r)$. Since $B^+ \subseteq A_c$, for each $e \in B^+$, there is at least one cycle in $\mathcal{C}$ that contains e . We choose an arbitrary such cycle and we denote it by C_e . (Note that it may happen that $C_e = C_{e'}$ for two different edges $e, e' \in B^+$.) Let $\mathcal{C}^+ = \{C_e \mid e \in B^+\}$. Then we define c' , by putting for every $u, v \in V$

$$c'(u, v) = r(u, v) + (-1)^{[(u, v) \in A_r]} \sum_{C \in \mathcal{C}^+} [(u, v) \in C]. \quad (1)$$

In other words, c' is obtained from r by iterating over all cycles in $C \in \mathcal{C}^+$, and adding one copy of each edge of $C \cap A_c$ and removing one copy of each edge of $C \cap A_r$.

Let us show that $G_{c'}$ contains B . This is trivial for every $e \in B^+$. When $e \in E(B) \cap E(G_r)$, consider two cases. If $e \notin A_r$, then $c'(e) \geq r(e)$, so $e \in G_{c'}$. If $e \in A_r$, $m_A(e) = r(e) - c(e)$. Then $c'(e) = r(e) - \sum_{C \in \mathcal{C}^+} [(u, v) \in C] \geq r(e) - m_A(e) = c(e) \geq 1$, where the last inequality follows since $B \subseteq G_c$.

To see that c' satisfies the degree constraints, recall that r does so, and note that if in (1) we consider only the summands corresponding to a single cycle $C \in \mathcal{C}^+$, then for every vertex we either add one outgoing edge and remove one outgoing edge, or add one ingoing edge and remove one ingoing edge, or we do not change the set of edges incident to it.

For a cycle $C \in \mathcal{C}$ let $\delta(C) = d(A_c \cap C) - d(A_r \cap C)$. Observe that for every cycle $C \in \mathcal{C}$ we have $\delta(C) \geq 0$, for otherwise $E(G_r) \setminus (C \cap A_r) \cup (C \cap A_c)$ contradicts the optimality of r . It follows that

$$d(c') = d(r) + \sum_{C \in \mathcal{C}^+} \delta(C) \leq d(r) + \sum_{C \in \mathcal{C}} \delta(C) = d(c). \quad (2)$$

Hence, since c is optimal solution of FIXED DEGREE $\mathcal{F}$ -SUBGRAPH, we get that c' is optimal solution of FIXED DEGREE $\mathcal{F}$ -SUBGRAPH as well. Moreover, by (1), for every $u, v \in V$,

$$|c'(u, v) - r(u, v)| \leq |\mathcal{C}^+| \leq |B| \leq s_{|V|}(\mathcal{F}). \quad (3)$$

This ends the proof. $\blacktriangleleft$

As noted in [7, 2], FIXED DEGREE SUBGRAPH can be solved by a reduction to minimum cost flow. By applying Orlin's algorithm [23] we get the following.

► **Observation 3.3** (Folklore, [7, 2]). *FIXED DEGREE SUBGRAPH can be solved in time $O(n^3 \log n)$.*

► **Theorem 3.4** (Kernelization). *There is a polynomial time algorithm which, given an instance $I = (d, \text{in}, \text{out})$ of FIXED DEGREE $\mathcal{F}$ -SUBGRAPH, outputs an instance $I' = (d, \text{in}', \text{out}')$ of the same problem and a function $f : V^2 \rightarrow \mathbb{Z}_{\geq 0}$ such that*

- (i) $\text{in}'(v), \text{out}'(v) = O(n \cdot s_n(\mathcal{F}))$ for every vertex v ,
- (ii) if m^* is an optimal solution for I' , then $f + m^*$ is an optimal solution for I .

The algorithm does not need to know $\mathcal{F}$, just the value of $s_n(\mathcal{F})$.

Proof. Our algorithm begins by finding an optimal solution r of FIXED DEGREE SUBGRAPH using Observation 3.3.

Define $f_0 : V^2 \rightarrow \mathbb{Z}_{\geq 0}$, where for every $v, w \in V$ we put $f_0(v, w) = \max\{r(v, w) - s_n(\mathcal{F}), 0\}$. By Lemma 3.2, there exists an optimal solution c' for instance I such that $c' \geq f_0$. Now define $f : V^2 \rightarrow \mathbb{Z}_{\geq 0}$, where for every $v, w \in V$ we put $f(v, w) = \max\{f_0(v, w) - 1, 0\}$. Finally, we put $\text{in}'(v) = \text{in}(v) - \sum_{w \in V} f(w, v)$ and $\text{out}'(v) = \text{out}(v) - \sum_{w \in V} f(v, w)$. The algorithm outputs $I' = (d, \text{in}', \text{out}')$ and f . In what follows, we show that the output has the desired properties.

For the property (i), consider any vertex $v \in V$ and observe that $\sum_{w \in V} f(v, w) \geq \sum_{w \in V} (f_0(v, w) - 1) \geq \sum_{w \in V} (r(v, w) - s_n(\mathcal{F}) - 1)$. Since r is a feasible solution of I , we have $\text{out}(v) = \sum_{w \in V} r(v, w)$. It follows that $\text{out}'(v) \leq n(1 + s_n(\mathcal{F})) = O(n \cdot s_n(\mathcal{F}))$ as required. The argument for $\text{in}'(v)$ is symmetric.

Now we focus on (ii). Let m^* be an optimal solution for I' . It is easy to check that $f + m^*$ satisfies the degree constraints for the instance I . Also, since m^* contains a subgraph from $\mathcal{F}$, then $f + m^*$ contains the same subgraph. It follows that $f + m^*$ is a feasible solution of I . It suffices to show that $f + m^*$ is an optimal solution for I .

Denote $r = c' - f$. Consider any pair $v, w \in V$ such that $c'(v, w) \geq 1$. We claim that $f(v, w) \leq c'(v, w) - 1$. Indeed, if $f_0(v, w) = 0$ then $f(v, w) = 0 \leq c'(v, w) - 1$, and if $f_0(v, w) \geq 1$ then $f(v, w) = f_0(v, w) - 1 \leq c'(v, w) - 1$. It follows that $r(v, w) \geq 1$. In particular, since c' contains a subgraph from $\mathcal{F}$, then also r contains the same subgraph. It follows that r is a feasible solution for I' (the degree constraints are easy to check). Hence, $d(m^*) \leq d(r)$. It follows that $d(f + m^*) \leq d(r + f) = d(c')$, so $f + m^*$ is indeed an optimal solution for I . $\blacktriangleleft$

4 The small costs case in time $\mathcal{O}^*(2^n D)$

In this section we establish Theorem 1.1. We do it in a bottom-up fashion, starting with a simplified core problem, and next generalizing the solution in a few steps.

4.1 Unweighted decision version with small degree demands

Consider the following problem.

DECISION UNWEIGHTED FIXED DEGREE CONNECTED SUBGRAPH

Input: a digraph $G = (V, E)$, $\text{in}, \text{out} : V \rightarrow \mathbb{Z}_{\geq 0}$

Question: Is there a function $m : V^2 \rightarrow \mathbb{Z}_{\geq 0}$ such that $G_m^\downarrow$ is a connected subgraph of G and for every $v \in V$ we have $\text{in}(v) = \text{indeg}_{G_m}(v)$ and $\text{out}(v) = \text{outdeg}_{G_m}(v)$?

Note that DECISION UNWEIGHTED FIXED DEGREE CONNECTED SUBGRAPH generalizes the directed Hamiltonian cycle problem, which is known to be solvable in $\mathcal{O}^*(2^n)$ time and polynomial space. In this section we show that this running time can be obtained for the more general problem as well, though we need to allow some randomization.

► **Theorem 4.1.** *There is a randomized algorithm which solves an instance $I = (\text{in}, \text{out})$ of DECISION UNWEIGHTED FIXED DEGREE CONNECTED SUBGRAPH in time $\mathcal{O}^*(2^n \text{poly}(M))$ and polynomial space, where $M = \max_v \max\{\text{in}(v), \text{out}(v)\}$. The algorithm is Monte Carlo with one-sided error, i.e., the positive answer is always correct and the negative answer is correct with probability at least p , for any constant $p < 1$.*

Our strategy will be to reduce our problem to detecting a perfect matching in a bipartite graph with an additional connectivity constraint.

We define a bipartite graph $B_G = (O, I, E(B_G))$ as follows. Let $I = \{v_1^I, \dots, v_{\text{in}(v)}^I \mid v \in V(H)\}$, $O = \{v_1^O, \dots, v_{\text{out}(v)}^O \mid v \in V(H)\}$, and $E(B_G) = \{u_i^O v_j^I \mid (u, v) \in E(G)\}$.

► **Observation 4.2.** $|I| = |O| = \mathcal{O}(nM)$ and $|E(B_G)| \leq E(G)M^2 = \mathcal{O}(n^2 M^2)$.

For an undirected graph H by $\mathcal{PM}(H)$ we denote the set of perfect matchings in H . We say that a matching M in B_G is *connected* when for every cut $(X, V \setminus X)$ with $\emptyset \neq X \subsetneq V$ the matching M contains an edge $u_i^O v_j^I$ such that $u \in X$ and $v \in V \setminus X$ or $v \in X$ and $u \in V \setminus X$.

For a matching M in B_G we define a *contraction* of M as function $m : V^2 \rightarrow \mathbb{Z}_{\geq 0}$ such that $m(u, v) = |\{u_i^O v_j^I \in M \mid i \in [\text{out}(u)], j \in [\text{in}(v)]\}|$. In other words G_m is obtained from M by (1) orienting every edge from O to I and (2) identifying all vertices in $\{v_1^I, \dots, v_{\text{in}(v)}^I\} \cup \{v_1^O, \dots, v_{\text{out}(v)}^O\}$ for every $v \in V$, and keeping the multiple edges and loops.

► **Lemma 4.3.** $(G, \text{in}, \text{out})$ is a yes-instance of DECISION UNWEIGHTED FIXED DEGREE CONNECTED SUBGRAPH iff graph B_G contains a connected perfect matching.

Proof. Let M be a connected perfect matching in B_G and let m be its contraction. We claim that m is a solution of $(G, \text{in}, \text{out})$. By the definition of B_G , $G_m^\downarrow$ is a subgraph of G . Since M is connected, G_m is connected as well, and so is $G_m^\downarrow$. Moreover, since M is a perfect matching $\text{in}(v) = \text{indeg}_{G_m}(v)$ and $\text{out}(v) = \text{outdeg}_{G_m}(v)$ for every vertex v .

For the other direction, let m be a solution for $(G, \text{in}, \text{out})$. For every $v \in V$, there are exactly $\text{out}(v)$ edges leaving v in G_m . Let us denote them $e_{v,1}^O, \dots, e_{v,\text{out}(v)}^O$. Similarly, let us denote all the edges entering v by $e_{v,1}^I, \dots, e_{v,\text{in}(v)}^I$. Then we define M as the set of edges of the form $u_i^O v_j^I$ such that G_m contains an edge $e = e_{u,i}^O = e_{v,j}^I$. The fact that M is a perfect matching is clear from the construction. Also, M is connected, for otherwise G_m is not connected. ◀

From now on, let $B = (O, I, E(B))$ be an arbitrary subgraph of B_G . Define the following multivariate polynomial over $\text{GF}(2^t)$, for an integer t to be specified later.

$$R = \sum_{\substack{M \in \mathcal{PM}(B) \\ M \text{ is connected}}} \prod_{e \in M} x_e \quad (4)$$

► **Lemma 4.4.** *R is not the zero polynomial if and only if B contains a connected perfect matching.*

Proof. It is clear that if R is non-zero then B contains a connected perfect matching. For the reverse implication it suffices to notice that every summand in R has a different set of variables, so it does not cancel out with other summands over $\text{GF}(2^t)$. ◀

Our strategy is to test whether R is non-zero by means of DeMillo–Lipton–Schwartz–Zippel Lemma, which we recall below.

► **Lemma 4.5** (DeMillo and Lipton [11], Schwartz [25], Zippel [28]). *Let $P(x_1, x_2, \dots, x_m)$ be a nonzero polynomial of degree at most d over a field $\mathbb{F}$ and let S be a finite subset of $\mathbb{F}$. Then, the probability that P evaluates to zero on a random element $(a_1, a_2, \dots, a_m) \in S^m$ is bounded by $d/|S|$.*

By Lemmas 4.4 and 4.5, the task reduces to *evaluating* R fast. To this end, we will define a different polynomial P which is easier to evaluate and turns out to be equal to R over $\text{GF}(2^t)$.

Consider a subset $X \subseteq V$. Let $I_X = \{v_i^I \in I \mid v \in X, i = 1, \dots, \text{in}(v)\}$ and $O_X = \{v_i^O \in O \mid v \in X, i = 1, \dots, \text{out}(v)\}$. Abusing the notation slightly, we will denote $B[X] = B[I_X \cup O_X]$. Define the following polynomial.

$$P_X = \sum_{M \in \mathcal{PM}(B[X])} \prod_{e \in M} x_e \quad (5)$$

In what follows, v^* is an arbitrary but fixed vertex of V . Define yet another polynomial.

$$P = \sum_{\substack{X \subseteq V \\ v^* \in X}} P_X P_{V \setminus X}. \quad (6)$$

► **Lemma 4.6.** $P = R$.

Proof. For a matching M in B we say that a set $X \subseteq V$ is *consistent* with M when M does not contain an edge $u_i^O v_j^I$ such that $u \in X$ and $v \in V \setminus X$ or $v \in X$ and $u \in V \setminus X$. The family of all subsets of V that are consistent with M will be denoted by $\mathcal{C}(M)$. Then we can rewrite P as follows.

$$\begin{aligned} P &= \sum_{\substack{X \subseteq V \\ v^* \in X}} \sum_{M_1 \in \mathcal{PM}(B[X])} \sum_{M_2 \in \mathcal{PM}(B[V \setminus X])} \prod_{e \in M_1 \cup M_2} x_e && [\text{definition}] \\ &= \sum_{M \in \mathcal{PM}(B)} \sum_{\substack{X \in \mathcal{C}(M) \\ v^* \in X}} \prod_{e \in M} x_e && [\text{group by } M = M_1 \uplus M_2] \\ &= \sum_{M \in \mathcal{PM}(B)} |\{X \in \mathcal{C}(M) \mid v^* \in X\}| \prod_{e \in M} x_e && [\text{trivial}] \end{aligned}$$

Let us consider a perfect matching $M \in \mathcal{PM}(B)$ and the corresponding contraction m . Observe that the number of sets that are consistent with M and contain a vertex v^* is equal to $2^{\text{cc}(M)-1}$, where $\text{cc}(M)$ is the number of connected components of G_m . Indeed, when X is consistent with M , then for every connected component Q of G_m , either $V(Q) \subseteq X$ or $V(Q) \subseteq V \setminus X$. For the component that contains v^* the choice is fixed, while every choice for the remaining components defines a set consistent with M . It follows that when M is not connected $\text{cc}(M) \geq 2$, and the value of $2^{\text{cc}(M)-1}$ is equal to 0 in $\text{GF}(2^t)$, so the corresponding summand vanishes. On the other hand, if M is connected, the corresponding summand equals just $\prod_{e \in M} x_e$ and it does not cancel out with another summand because the monomial has a unique set of variables. It follows that $P = R$. ◀

► **Lemma 4.7** (Tutte, Lovász [26, 22]). *For an arbitrary set $X \subseteq V$, the polynomial P_X can be evaluated using $\text{poly}(n + M)$ field operations.*

Proof. Compute the determinant of the corresponding Tutte matrix of dimension $|O| \times |I|$. ◀

Let us now fix our field, namely $t = \lceil 1 + \log n + \log M \rceil$. Since arithmetic operations in $\text{GF}(2^t)$ can be performed in time $\mathcal{O}(t \log^2 t) = \mathcal{O}(\log(n + M) \log^2 \log(n + m))$, by the definition of P and Lemma 4.7 we get the following corollary.

► **Corollary 4.8.** *P can be evaluated in time $2^n \text{poly}(n + M)$.*

► **Lemma 4.9.** *There is a randomized algorithm which decides if B contains a connected perfect matching in time $\mathcal{O}^*(2^n \text{poly}(M))$ and polynomial space, where $M = \max_v \max\{\text{in}(v), \text{out}(v)\}$. The algorithm is Monte Carlo with one-sided error, i.e., the positive answer is always correct and the negative answer is correct with probability at least p , for any constant $p < 1$.*

Proof. The algorithm evaluates polynomial P using Corollary 4.8 substituting a random element of $\text{GF}(2^t)$ for each variable, and reports “yes” when the evaluation is nonzero and “no” otherwise. If it reported “yes”, then P was a non-zero polynomial and by Lemma 4.4 the answer is correct. Assume it reported ‘no’ for a yes-instance. By Lemma 4.4 P is non-zero. Since $\deg P = |I| \leq nM$, by Lemma 4.5 the probability that P evaluated to 0 is bounded by $\deg P / 2^t \leq 1/2$ and we can make this probability arbitrarily small by repeating the whole algorithm a number of times, and reporting “yes” if at least one evaluation was nonzero. The claim follows. ◀

Theorem 4.1 follows immediately from Lemma 4.3 and Lemma 4.9 applied to B_G .

4.2 Finding the solution

► **Lemma 4.10.** *There is a randomized algorithm which, given a yes-instance of DECISION UNWEIGHTED FIXED DEGREE CONNECTED SUBGRAPH, always returns the corresponding solution m in expected time $\mathcal{O}^*(2^n \text{poly}(M))$. The time can be made deterministic at the cost of introducing arbitrarily small probability of failure.*

In order to prove Lemma 4.10 we cast the problem in the setting of *inclusion oracles* from the work of Björklund et al. [5]. Consider a universe U and an (unknown) family of *witnesses* $\mathcal{F} \subseteq 2^U$. An *inclusion oracle* is a procedure which, given a query set $Y \subseteq U$, answers (either YES or NO) whether there exists at least one witness $W \in \mathcal{F}$ such that $W \subseteq Y$. Björklund et al. prove the following.

► **Theorem 4.11** ([5]). *There exists an algorithm that extracts a witness of size k in $\mathcal{F}$ using in expectation at most $O(k \log |U|)$ queries to a randomized inclusion oracle that has no false positives but may output a false negative with probability at most $p \leq \frac{1}{4}$.*

Proof of Lemma 4.10. Let $U = E(B_G)$ and let $\mathcal{F}$ be the family of all connected perfect matchings in B_G . Note that $|U| = O(n^2 M^2)$ and witnesses in $\mathcal{F}$ have all size $|I| = O(nM)$. Then, Lemma 4.9 provides a randomized inclusion oracle and we can apply Theorem 4.11. (If one insists on deterministic, and not expected, running time, it suffices to chose a sufficiently large constant r and stop the algorithm if it exceeds the expected running time at least r times – by Markov’s inequality, this happens with probability at most $1/r$.) ◀

4.3 Proof of Theorem 1.1

In the lemma below we will adapt the construction from Section 4.1 to the weighted case in a standard way, by introducing a new variable tracking the weight.

► **Lemma 4.12.** *There is a randomized algorithm which solves an instance $I = (d, in, out, w)$ of FIXED DEGREE CONNECTED SUBGRAPH in time $\mathcal{O}^*(2^n D \text{poly}(M))$ and polynomial space, where $M = \max_v \max\{in(v), out(v)\}$ and D is the maximum integer value of d . The algorithm returns a minimum weight solution with probability at least p , for any constant $p < 1$.*

Proof. Define $G = (V, E)$ where $E = \{(u, v) \in V^2 \mid d(u, v) \in \mathbb{Z}_{\geq 0}\}$. Let R' be the polynomial obtained from R by replacing every variable x_e for $e = u_i^O v_j^I \in E(B_G)$ by the product $x_e \cdot y^{d(u, v)}$, where y is a new variable. Proceed similarly with P , obtaining P' . By Lemma 4.4, $P' = R'$. Decompose R' as $R' = \sum_{i=0}^{|I| \cdot D} R'_i y^i$, where R'_i , for every $i = 0, \dots, |I| \cdot D$, is a polynomial in variables $\{x_e\}_{e \in E(B_G)}$. The monomials in R'_i enumerate all matchings M such that the contraction m of M has weight $d(m) = i$. By the construction in the proof of Lemma 4.3 R'_i is non-zero if and only if instance I has a solution of weight i . Using Lagrange interpolation, we can recover the value of each R'_i for random values of the variables $\{x_e\}_{e \in E(B_G)}$ (the values are the same for all the polynomials). The interpolation algorithm requires $|I| \cdot D = \mathcal{O}(nMD)$ evaluations of R' . Since $R' = P'$, by Lemma 4.8 each of them takes $2^n \text{poly}(n + M)$ time. Our algorithm reports the minimum w such that R'_w evaluated to a non-zero element of $\text{GF}(2^t)$, or $+\infty$ if no such w exists. The solution of weight w is then found using Lemma 4.10. The event that the optimum value w^* is not reported means that R'_{w^*} is a non-zero polynomial that evaluated to 0 at the randomly chosen values. By Lemma 4.5 this happens with probability at most $\deg P/2^t \leq 1/2$, and one can make this probability arbitrarily small by standard methods. ◀

Theorem 1.1 follows now immediately by applying Theorem 3.4 which reduces the general problem to the $M = \mathcal{O}(n^2)$ case and solving the resulting instance by Lemma 4.12. Theorem 1.1 says in particular that if finite weights are bounded by a polynomial in n then we can solve MANY VISITS TSP in time $\mathcal{O}^*(2^n)$ and polynomial space by a randomized algorithm with no false positives and with false negatives with arbitrarily small constant probability.

5 The general case

In this section we prove Theorem 1.3, i.e., we show an algorithm solving MANY VISITS TSP in time $\mathcal{O}^*(4^n)$. In fact, we do not introduce a new algorithm, but we consider an algorithm by Berger et al. (Algorithm 5 in [3]) and we provide a refined analysis, resulting in an improved running time bound $\mathcal{O}^*(4^n)$, which is tight up to a polynomial factor.

Let us recall the algorithm of Berger et al., in a slightly changed notation. In fact, they solve a slightly more general problem, namely **FIXED DEGREE SUBGRAPH WITH OUTBRANCHING**. Let $I = (d, \text{in}, \text{out}, r)$ be an instance of this problem, i.e., we want to find a solution $m : V^2 \rightarrow \mathbb{Z}_{\geq 0}$ that satisfies the degree constraints specified by in and out and contains an outbranching rooted at r . In what follows we assume $V = \{1, \dots, n\}$ and $r = 1$.

Consider an outbranching sequence $\{\delta_v\}_{v \in V}$ rooted at $r = 1$. In what follows, all outbranching sequences will be rooted at 1, so we skip specifying the root. Let T_δ be a minimum cost outbranching among all outbranchings with outdegree sequence δ and let r_δ be an optimum solution of **FIXED DEGREE SUBGRAPH** for instance $(d, \text{in}', \text{out}')$ where $\text{out}' = \text{out} - \text{outdeg}_{T_\delta}$ and $\text{in}' = \text{in} - \text{indeg}_{T_\delta}$. Berger et al. note that then $m_\delta = m_{T_\delta} + r_\delta$ is a feasible solution for instance I of **FIXED DEGREE SUBGRAPH WITH OUTBRANCHING**, and moreover it has minimum cost among all solutions that contain an outbranching with outdegree sequence δ . Since r_δ can be found in polynomial time by Observation 3.3, in order to solve instance I it suffices to find outbranchings T_δ for all outbranching sequences δ and return the solution m_δ of minimum cost. Hence, Theorem 1.3 boils down to proving the following lemma.

► **Lemma 5.1.** *There is an algorithm which, for every outbranching sequence δ , finds a minimum cost outbranching among all outbranchings with outdegree sequence δ and runs in time $\mathcal{O}^*(4^n)$.*

We prove Lemma 5.1 by using dynamic programming (DP). However, it will be convenient to present the DP as a recursive function **BESTOUTBRANCHING** with two parameters, $S \subseteq V$ and $\{\delta_v\}_{v \in S}$ (see Algorithm 1). It is assumed that $1 \in S$. We will show that **BESTOUTBRANCHING**(S, δ) returns a minimum cost out-tree among all out-trees with outdegree sequence δ that are rooted at 1 and span S . Our algorithm runs **BESTOUTBRANCHING** for $S = V$ and all outbranching sequences $\delta : V \rightarrow \mathbb{Z}_{\geq 0}$. Whenever **BESTOUTBRANCHING** returns a solution for an input (S, δ) , it is memoized (say, in an efficient dictionary), so that when **BESTOUTBRANCHING** is called with parameters (S, δ) again, the output can be retrieved in polynomial time.

■ **Algorithm 1** A pseudocode of the algorithm from Lemma 5.1.

```

function BESTOUTBRANCHING( $S, \delta$ )
   $v_{\text{first}} \leftarrow \min\{v \in S \mid \delta_v = 0\}$ 
  if  $|S| = 2$  then return  $\{(1, v_{\text{first}})\}$ .
  else
     $\text{minCost} \leftarrow \infty$ 
    for  $w \in S$  do
      if  $(\delta_w \geq 1 \wedge w \neq 1) \vee (\delta_w \geq 2 \wedge w = 1)$  then
         $S' \leftarrow S \setminus \{v_{\text{first}}\}$ 
         $\delta' \leftarrow \delta|_{S'}$ 
         $\delta'_w \leftarrow \delta'_w - 1$ 
         $R_w \leftarrow \text{BESTOUTBRANCHING}(S', \delta') \cup \{(w, v_{\text{first}})\}$ 
        if  $d(R_w) < \text{minCost}$  then
           $\text{minCost} \leftarrow d(R_w)$ 
           $\text{best} \leftarrow R_w$ 
    return best

```

Let us define $\text{lastRmvd}(S) := \max(\{0, 1, 2, \dots, n\} \setminus S)$ and $\text{bad}(S, \delta) := \{v \in S \mid v < \text{lastRmvd}(S) \wedge \delta_v = 0\}$. Let us call (S, δ) a *reachable state* if it meets the following conditions:

- (i) $\delta_1 \geq 1$
- (ii) $\sum_{v \in S} \delta_v = |S| - 1$
- (iii) $|\text{bad}(S, \delta)| \leq 1$

► **Lemma 5.2.** *If function BESTOUTBRANCHING is given a reachable state as input then all recursively called BESTOUTBRANCHING will also be given only reachable states.*

Proof. Let us fix a reachable state (S, δ) for $|S| > 2$ and consider the associated value v_{first} from the algorithm. Denote $S' = S \setminus \{v_{\text{first}}\}$. Clearly, it suffices to show that all pairs (S', δ') created in the **for** loop are reachable states. First, let us argue that $\text{bad}(S', \delta) = \emptyset$. There are two cases:

- Assume $|\text{bad}(S, \delta)| = 0$. In this case $v_{\text{first}} > \text{lastRmvd}(S)$ so $\text{lastRmvd}(S') = v_{\text{first}}$. Then, $\text{bad}(S', \delta) = \{v \in S' \mid v < \text{lastRmvd}(S') \wedge \delta_v = 0\} = \{v \in S \mid v < v_{\text{first}} \wedge \delta_v = 0\} = \emptyset$.
- Assume $|\text{bad}(S, \delta)| = 1$. Then, (1) $\text{lastRmvd}(S') = \text{lastRmvd}(S)$ because $\text{lastRmvd}(S) > v_{\text{first}}$ and (2) $\text{bad}(S, \delta) = \{v_{\text{first}}\}$. It follows that $\text{bad}(S', \delta) \stackrel{(1)}{=} \{v \in S' \mid v < \text{lastRmvd}(S) \wedge \delta_v = 0\} = \text{bad}(S, \delta) \setminus \{v_{\text{first}}\} \stackrel{(2)}{=} \emptyset$.

Let us consider the recursive call of BESTOUTBRANCHING for a particular w . The sequence $\delta'|_{S'}$ differs from δ only at w , so $\text{bad}(S', \delta') \subseteq \{w\} \cup \text{bad}(S', \delta) = \{w\}$. This means that condition (iii) from the definition of a reachable state holds for (S', δ') . Since (S, δ) is reachable, $\delta_1 \geq 1$. Then either $w \neq 1$ and $\delta'_1 = \delta_1 \geq 1$ or $w = 1$ and $\delta'_1 = \delta_1 - 1 \geq 1$, where the last inequality holds thanks to the condition in the **if** statement in Algorithm 1. In both cases, (i) holds for (S', δ') . Finally, (ii) is immediate by the definition of δ' . It follows that (S', δ') is a reachable state, as required. ◀

► **Lemma 5.3.** *If the function BESTOUTBRANCHING is given a reachable state (S, δ) , it returns a cheapest out-tree T rooted at vertex 1, spanning S and with outdegree sequence δ .*

Proof. We will use induction on $|S|$.

In the base case $|S| = 2$, there is only one outbranching spanning S rooted at 1, namely $\{(1, v_{\text{first}})\}$ and it is indeed returned by the algorithm.

In the inductive step assume $|S| > 2$. By conditions (i) and (ii) in the definition of a reachable state and Lemma 2.2, there is at least one out-tree rooted at 1, spanning S , and with outdegree sequence δ . Let T be a cheapest out-tree among all such out-trees. Vertex v_{first} is a leaf of T , since $\delta_{v_{\text{first}}} = 0$. At some point w in the **for** loop in Algorithm 1 is equal to the parent w^* of v_{first} in T . Then, $T \setminus \{(w^*, v_{\text{first}})\}$ is an out-tree rooted at 1, spanning S' , and with outdegree sequence δ' . Since (S', δ') is a reachable state by Lemma 5.2, by the inductive hypothesis we know that a cheapest such out-tree T' will be returned by BESTOUTBRANCHING(S', δ'). In particular, it means that $d(T') \leq d(T \setminus \{(w^*, v_{\text{first}})\})$. Denote $R_{w^*} = T' \cup \{(w^*, v_{\text{first}})\}$. Then, $d(R_{w^*}) = d(T') + d(w^*, v_{\text{first}}) \leq d(T \setminus \{(w^*, v_{\text{first}})\}) + d(w^*, v_{\text{first}}) = d(T)$. It follows that BESTOUTBRANCHING returns a set of edges **best** of cost at most $d(T)$. However **best** = R_w for a vertex w and by applying the induction hypothesis it is easy to see that R_w is an out-tree rooted at 1, spanning S with outdegree sequence δ . The claim follows. ◀

► **Lemma 5.4.** *There are $\mathcal{O}^*(4^n)$ reachable states.*

Proof. Any sequence of n nonnegative integers that sums up to at most $n - 1$ will be called an *extended sequence*. It is well known that there are exactly $\binom{2n-1}{n} < 2^{2n-1} = \mathcal{O}(4^n)$ such sequences. To see this consider sequences of $n - 1$ balls and n barriers and bijectively map them to the sequences of n numbers by counting balls between barriers and discarding the balls after the last barrier.

Let us fix an extended sequence $\bar{\delta} = \{\bar{\delta}_v\}_{v \in V}$, and denote $\bar{s} := n - (1 + \sum_{i=1}^n \bar{\delta}_i)$. We claim that there are only $\mathcal{O}(n)$ reachable states (S, δ) such that $\bar{\delta}|_S = \delta$ and $\bar{\delta}|_{V \setminus S} = 0$. Consider any such pair (S, δ) . Let $(v_1, v_2, \dots, v_k)$ be the vertices of $\{v \in V \mid \bar{\delta}_v = 0\}$ sorted

in increasing order. By the definition of a reachable state we know that $|S| = 1 + \sum_{i=1}^n \bar{\delta}_i$, so $\bar{s} = |\{1, 2, \dots, n\} \setminus S|$. By (ii), for at least one vertex $v \in S$ we have $\bar{\delta}_v = \delta_v = 0$, so $k \geq \bar{s} + 1$. Let us assume that $k \geq \bar{s} + 2$ and $\text{lastRmvd}(S) \geq v_{\bar{s}+2}$. Then, $\{v_1, v_2, \dots, v_{\bar{s}+1}\} \cap S \subseteq \text{bad}(S, \delta)$. Since $v_{\bar{s}+2} \leq \text{lastRmvd}(S) \notin S$, at most $\bar{s} - 1$ elements from $\{v_1, v_2, \dots, v_{\bar{s}+1}\}$ are outside S , so $|\text{bad}(S, \delta)| \geq (\bar{s} + 1) - (\bar{s} - 1) = 2$. This is a contradiction with (S, δ) being a reachable state, which proves that $k \leq \bar{s} + 1$ or $\text{lastRmvd}(S) < v_{\bar{s}+2}$. In any case, $\{1, 2, \dots, n\} \setminus S \subseteq \{v_1, \dots, v_{\bar{s}+1}\}$. There are $\bar{s} + 1 = \mathcal{O}(n)$ ways to choose $\bar{s}$ elements to the set $\{1, 2, \dots, n\} \setminus S$ from $\{v_1, \dots, v_{\bar{s}+1}\}$, so equivalently there are $\mathcal{O}(n)$ sets S such that (S, δ) is a reachable state, $\bar{\delta}|_S = \delta$ and $\bar{\delta}|_{V \setminus S} = 0$.

Every reachable state (S, δ) has the corresponding extended sequence $\{\bar{\delta}\}_{v \in V}$ defined by $\bar{\delta}|_S = \delta$ and $\bar{\delta}|_{V \setminus S} = 0$. Since there are $\mathcal{O}(4^n)$ extended sequences, and each of them has $\mathcal{O}(n)$ corresponding reachable states there are $\mathcal{O}(4^n) \cdot \mathcal{O}(n) = \mathcal{O}^*(4^n)$ reachable states in total. $\blacktriangleleft$

We are ready to prove Lemma 5.1. Recall that our algorithm runs $\text{BESTOUTBRANCHING}(V, \delta)$ for all outbranching sequences δ and uses memoization to avoid repeated computation. We claim that for any outbranching sequence δ , the pair (V, δ) is a reachable state. Indeed, conditions (i) and (ii) hold since δ is an outbranching sequence. By definition, $\text{lastRmvd}(V) = 0$, so $\text{bad}(V, \delta) = \emptyset$ which implies (iii). Hence by Lemma 5.3 the algorithm is correct. By Lemma 5.2 the running time can be bounded by the number of reachable states times a polynomial, which is $\mathcal{O}^*(4^n)$ by Lemma 5.4. This ends the proof of Lemma 5.1 and hence also Theorem 1.3, as discussed in the beginning of this section.

6 Polynomial space

In this section we show Theorem 1.4, that is, we solve MANY VISITS TSP in $\mathcal{O}^*(7.88^n)$ time and polynomial space. Berger et al. [2] solved this problem in $\mathcal{O}(16^{n+o(n)})$ time and polynomial space, with the key ingredient being the following.

► **Lemma 6.1** (Berger et al. [2]). *There is a polynomial space algorithm running in time $\mathcal{O}(4^{n+o(n)})$ which, given an outdegree sequence $\{\delta_v\}_{v \in V}$, a cost function $d : V^2 \rightarrow \mathbb{Z}_{\geq 0}$, and a root $r \in V$ computes the cheapest outbranching rooted at r with the required outdegrees.*

More precisely, the $\mathcal{O}(16^{n+o(n)})$ -time algorithm consists of the following steps:

- (i) Enumerate all $\mathcal{O}(4^n)$ outbranching sequences
- (ii) For each outbranching sequence compute the cheapest outbranching with required degrees using Lemma 6.1 in time $\mathcal{O}(4^{n+o(n)})$
- (iii) For each of these outbranchings complete it to a solution of the original MANY VISITS TSP instance with an optimal solution of FIXED DEGREE SUBGRAPH on the residual degree sequences (in polynomial time, by Observation 3.3).

The intuition behind our approach is as follows. We iterate over all subsets of vertices R . Here, R represents our guess of the set of inner vertices of an outbranching in an optimal solution. Then we perform (i) and (ii) in the smaller subgraph induced by R . Finally, we replace (iii) by a more powerful flow-based algorithm which connects the vertices in $V \setminus R$ to R , and at the same time computes a feasible solution of FIXED DEGREE SUBGRAPH on the residual degree sequences, so that the total cost is minimized. Let $r = |R|$. Clearly, when r is a small fraction of n , we get significant savings in the running time. The closer r/n is to 1 the smaller are the savings, but also the smaller is the number $\binom{n}{r}$ of sets R to examine.

In fact, the real algorithm is slightly more complicated. Namely, we fix an integer parameter K , and then R corresponds to the set of vertices left from an outbranching in an optimal solution after K iterations of removing all leaves. The running time of our algorithm depends on K , because the algorithm actually guesses the layers of leaves in each iteration. The space complexity is polynomial and does not depend on K . In the end of this section, we show that our running time bound is minimized when $K = 4$.

6.1 Our algorithm

Similarly as in Section 5, we solve the more general FIXED DEGREE SUBGRAPH WITH OUTBRANCHING: for a given instance $I = (d, \text{in}, \text{out}, \text{root})$ we want to find a solution $m : V^2 \rightarrow \mathbb{Z}_{\geq 0}$ that satisfies the degree constraints specified by in and out and contains an outbranching rooted at root .

Let T be an arbitrary outbranching. We define a sequence $L_1(T), L_2(T), \dots$ of subsets of $V(T)$ as follows. For $i \geq 1$ let $L_i(T)$ be the set of leaves of $T \setminus (L_1(T) \cup L_2(T) \cup \dots \cup L_{i-1}(T))$ if $|V(T) \setminus (L_1(T) \cup \dots \cup L_{i-1}(T))| > 1$, and otherwise $L_i = \emptyset$. The sets $L_i(T)$ will be called *leaf layers*. Denote $R_i(T) = V \setminus (L_1(T) \cup \dots \cup L_i(T))$ for any $i \geq 1$.

► **Lemma 6.2.** *For every $i \geq 1$ we have $\text{root} \in R_i(T) \setminus L_{i+1}(T)$, $|L_i(T)| \geq |L_{i+1}(T)|$ and $|L_{i+1}| \leq \frac{n - |R_i(T)|}{i}$.*

Proof. In this proof we skip the “ (T) ” in L_i and R_i because there is no ambiguity. Assume $\text{root} \in L_i$ for some $i \geq 1$. It means that root is a leaf in $T \setminus (L_1 \cup L_2 \cup \dots \cup L_{i-1})$. Then $V \setminus (L_1 \cup L_2 \cup \dots \cup L_{i-1}) = \{\text{root}\}$ and $L_i = \emptyset$, a contradiction. Hence $\text{root} \notin L_i$ for all $i \geq 1$, and in consequence $\text{root} \in R_i$ for all $i \geq 1$. However, $\text{root} \in R_{i+1}(T)$ implies that $\text{root} \notin L_{i+1}(T)$, hence $\text{root} \in R_i \setminus L_{i+1}$.

If $|V \setminus (L_1 \cup \dots \cup L_i)| > 1$, then L_{i+1} is the set of leaves of the out-tree $T \setminus (L_1 \cup \dots \cup L_i)$, which is contained in the set of parents of vertices in L_i . Since every vertex in L_i has exactly one parent, $|L_i| \geq |L_{i+1}|$. If $|V \setminus (L_1 \cup \dots \cup L_i)| \leq 1$ then $L_{i+1} = \emptyset$ and clearly $|L_i| \geq |L_{i+1}| = 0$.

Finally, since for every $j < i$ we have $|L_j| \geq |L_i|$ we get $n - |R_i| = |L_1| + \dots + |L_i| \geq i|L_i|$. It follows that $|L_{i+1}| \leq |L_i| \leq \frac{n - |R_i|}{i}$, as required. ◀

Pseudocode of our algorithm is presented as Algorithm 2.

■ **Algorithm 2** A pseudocode of the algorithm from Section 6.1.

```

1: function SOLVE( $G, \text{out}, \text{in}, d, \text{root}$ )
2:    $\text{best} \leftarrow \infty$ 
3:   for  $R, L_{K+1}, \delta$  do
4:      $T_R \leftarrow$  cheapest  $\delta$ -out-tree spanning  $R$  rooted at  $\text{root}$  (Lemma 6.1)
5:      $\text{out}' \leftarrow \text{out} - \text{outdeg}_{T_R}$ 
6:      $\text{in}' \leftarrow \text{in} - \text{indeg}_{T_R}$ 
7:     for  $L_1, \dots, L_K$  do
8:        $F \leftarrow \text{CREATENETWORK}(G, R, \text{out}', \text{in}', d, L_1, \dots, L_K)$ 
9:        $f \leftarrow \text{MINCOSTMAXFLOW}(F)$ 
10:      if  $|f| = \sum_{v \in V(G)} \text{out}'(v)$  and  $\text{cost}(f) + d(T_R) < \text{best}$  then
11:         $\text{best} \leftarrow \text{cost}(f) + d(T_R)$ 
  return best

```

For clarity, in the pseudocode we skipped some constraints that we enforce on the sets L_i and sequence δ . We state them below.

- (C1) $L_{K+1} \subseteq R \subseteq V, \text{root} \in R \setminus L_{K+1}, |L_{K+1}| \leq \frac{n-|R|}{K}$
- (C2) $\{\delta_v\}_{v \in R}$ is a rooted out-tree sequence, i.e., for all $v \in R$ we have $\delta_v \in \mathbb{Z}_{\geq 0}, \sum_{v \in R} \delta(v) = |R| - 1$; also $\delta_{\text{root}} \geq 1$ if $|R| \geq 2$ and $\delta_{\text{root}} = 0$ if $|R| = 1$.
- (C3) for every $v \in L_{K+1}$ we have $\delta_v = 0$ and for every $v \in R \setminus (L_{K+1} \cup \{\text{root}\})$ we have $\delta_v \geq 1$
- (C4) $L_1 \uplus L_2 \uplus \dots \uplus L_K = V \setminus R$
- (C5) $|L_i| \geq |L_{i+1}|$ for $i = 1, \dots, K$.

It is clear that all these possibilities can be enumerated in time proportional to their total number times $O(n)$.

Let us provide some further intuition about Algorithm 2. Consider an optimum solution m of I and any outbranching B in m rooted at root . In Algorithm 2, for any $i = 1, \dots, K+1$, the set L_i is a guess of the leaf layer $L_i(B)$, while R is a guess of $V \setminus (L_1(B) \cup \dots \cup L_K(B))$. Finally, δ is a guess of the outdegree sequence of the out-tree $B[R]$.

In Line 8 we create a flow network, and in line 9 a minimum cost maximum flow is found in polynomial time. In the next section we discuss the flow network and properties of the flow.

6.2 The flow

In this section we consider a run of Algorithm 2, and in particular we assume that the variables $R, \delta, L_1, \dots, L_{K+1}$ have been assigned accordingly. The function `CREATENETWORK` in our algorithm builds a flow network $F = (V(F), E(F), \text{cap}, \text{cost})$, where $E(F)$ is a set of directed edges and cap and cost are functions from edges to integers denoting capacities and costs of corresponding edges. As usual, the function cost extends to flow functions in a natural way, i.e., $\text{cost}(f) = \sum_{e \in E(F)} f(e) \text{cost}(e)$. We let $V(F) = \{s, t\} \cup \{v^I, v^O \mid v \in V(G)\} \cup \{v^C \mid v \in V \setminus R\}$, where s and t denote the source and the sink of F .

We put following edges into $E(F)$:

- (s, v^O) , where $\text{cap}(s, v^O) = \text{out}'(v)$, $\text{cost}(s, v^O) = 0$ for every $v \in V(G)$
- (v^I, t) , where $\text{cap}(v^I, t) = \text{in}'(v)$, $\text{cost}(v^I, t) = 0$ for every $v \in R$
- (v^I, t) , where $\text{cap}(v^I, t) = \text{in}'(v) - 1$, $\text{cost}(v^I, t) = 0$ for every $v \notin R$
- (v^C, t) , where $\text{cap}(v^C, t) = 1$, $\text{cost}(v^C, t) = 0$ for every $v \notin R$
- (u^O, v^I) , where $\text{cap}(u^O, v^I) = \infty$, $\text{cost}(u^O, v^I) = d(u, v)$ for every $(u, v) \in E(G)$
- (u^O, v^C) , where $\text{cap}(u^O, v^C) = \infty$, $\text{cost}(u^O, v^C) = d(u, v)$ for every $v \in L_i, u \in R \cup L_{i+1} \cup \dots \cup L_K, (u, v) \in E(G)$.

We will say that F has a *full flow* if it has a flow f with value $|f| = \sum_{v \in V} \text{out}'(v)$. By the construction of F , then all edges leaving source are saturated, i.e., carry flow equal to their capacity. Since $\sum_{v \in V} \text{out}'(v) = \sum_{v \in V} \text{in}'(v)$, also all edges that enter the sink are saturated.

Essentially, the network above results from extending the standard network used to get Observation 3.3 by vertices v^C . The flow between $\{v^O \mid v \in V\}$ and $\{v^I \mid v \in V\} \cup \{v^C \mid v \in V \setminus R\}$ represents the resulting solution. In a full flow the edges leaving v^C are saturated, so a unit of flow enters every vertex v^C , which results in connecting v in the solution to a higher layer or to R . Thanks to that the solution resulting from adding the out-tree T_R to the solution extracted from f contains an outbranching.

► **Lemma 6.3.** *If f is a full flow of minimum cost in F then there exists a solution of I with cost $\text{cost}(f) + d(T_R)$. Moreover, the solution can be extracted from f in polynomial time.*

Proof. By standard arguments, since all capacities in F are integer, we infer that there is an integral flow of minimum cost (and it can be found in polynomial time), so we assume w.l.o.g. that f is integral.

Let $b : V^2 \rightarrow \{0, 1\}$ denote a function such that $b(u, v) = [(u, v) \in T_R]$. Now we construct a solution $m : V^2 \rightarrow \mathbb{Z}_{\geq 0}$ of I .

$$m(u, v) = \begin{cases} f(u^O, v^I) + b(u, v) & \text{if } v \in R \\ f(u^O, v^I) + f(u^O, v^C) & \text{if } v \notin R. \end{cases}$$

In other words, m describes how many times edge (u, v) was used by the out-tree T_R and flow f in total. Let us verify that m is a feasible solution for I . The degree constraints are easy to verify, so we are left with showing that m contains an outbranching rooted at **root**. To this end it suffices to show that every vertex v is reachable from **root** in G_m . Clearly, this holds for vertices in R , thanks to the out-tree T_R . Pick an arbitrary vertex $v \notin R$. Then $v \in L_i$ for some $i = 1, \dots, K$. We know that $f(v^C, t) = 1$, so there exists u such that $f(u^O, v^C) = 1$. Therefore, v is connected in G_m to a vertex from $R \cup L_{i+1} \cup \dots \cup L_K$. Since v in G_m has an in-neighbor either in R or in a layer with a higher index, we can conclude that there is a path in G_m from R to v . Hence m indeed contains the required outbranching.

Finally, it can be easily checked that $d(m) = \text{cost}(f) + d(T_R)$, what concludes this proof. $\blacktriangleleft$

Let m be a feasible solution for I . Let R, L_i for $i = 1, \dots, K + 1$ be sets of vertices and δ an out-tree sequence on R , as in Algorithm 2. We say that m is *compliant* with $R, L_1, \dots, L_{K+1}$ and δ when m contains an outbranching T rooted at **root** such that $R_K(T) = R$, $L_i(T) = L_i$ for $i = 1, \dots, K + 1$ and δ is equal to the outdegree sequence of $T[R]$.

► **Lemma 6.4.** *Assume that there exists a solution m of I that is compliant with $R, L_1, \dots, L_{K+1}$ and δ . Then F has a full flow f such that $\text{cost}(f) + d(T_R) \leq d(m)$.*

Proof. Let T be an outbranching in m which certifies that m is compliant with $R, L_1, \dots, L_{K+1}$ and δ . Let $p : V^2 \rightarrow \{0, 1\}$ be a function such that for every $u, v \in V$ we have $p(u, v) = [(u, v) \in T]$.

We set $f(s, u) = \text{cap}(s, u)$ for all edges $(s, u) \in E(F)$ and $f(u, t) = \text{cap}(u, t)$ for all edges $(u, t) \in E(F)$. If $v \in V \setminus R$ then we set $f(u^O, v^C) = p(u, v)$. For all $u, v \in V(G)$ we set $f(u^O, v^I) = m(u, v) - p(u, v)$. It can be easily checked that such function f is a full flow and $\text{cost}(f) = d(m) - d(T[R])$. However, since $T[R]$ is a δ -out-tree rooted at **root** and T_R is a cheapest such out-tree, $d(T_R) \leq d(T[R])$. It follows that $\text{cost}(f) \leq d(m) - d(T_R)$, so $\text{cost}(f) + d(T_R) \leq d(m)$ as required. $\blacktriangleleft$

Consider a *minimum cost* full flow f' in F that is found by Algorithm 2 for a choice of $R, L_1, \dots, L_{K+1}, \delta$. The claim above implies that $\text{cost}(f') + d(T_R) \leq d(m)$. However, notice that we do not claim that $\text{cost}(f')$ is the cost of optimal completion of T_R consistent with all guesses, as the intuitions we described earlier might suggest. It could be the case that in the solution resulting from f' , a vertex which was guessed to belong to L_i does not have any out-neighbor that was guessed to belong to L_{i-1} , which would mean that this vertex should be in an earlier layer. However, that is not an issue for the extraction of the global optimum solution of I , because we may get only better solutions than the optimum completion for that particular guess.

6.3 Correctness

► **Lemma 6.5.** *Function SOLVE returns the cost of an optimal solution of I .*

Proof. From Lemma 6.3 we infer that SOLVE returns the cost of a feasible solution of I . It remains to show that it returns a value that is smaller or equal to the cost of an optimal solution of I . To this end, let m be an arbitrary optimal solution of I and let T be an arbitrary outbranching rooted at root in G_m . Let $R = R_K(T)$, $L_i = L_i(T)$ for $i = 1, \dots, K+1$ and let δ be the outdegree sequence of $T[R]$.

Let us verify that $R, L_1, \dots, L_{K+1}$ and δ satisfy constraints (C1)–(C5). We get (C1) and (C5) by Lemma 6.2. (C2) follows from the definition of δ . For (C3), consider two cases. If $|R| > 1$, then L_{K+1} is the set of leaves in R and hence indeed for every $v \in L_{K+1}$ we have $\delta_v = 0$ and for every $v \in R \setminus (L_{K+1} \cup \{\text{root}\})$ we have $\delta_v \geq 1$. When $|R| \leq 1$, we have $L_{K+1} = \emptyset$ and since $\text{root} \in R$ by Lemma 6.2, $R = \{\text{root}\}$. Then both sets L_{K+1} and $R \setminus (L_{K+1} \cup \{\text{root}\})$ are empty, so (C3) trivially holds. Finally, (C4) follows by the definition of leaf layers.

Since $R, L_1, \dots, L_{K+1}$ and δ satisfy constraints (C1)–(C5), then SOLVE reaches this particular evaluation of the variables $R, L_1, \dots, L_{K+1}$ and δ . Then, based on Lemma 6.4, the network F has a full flow f such that $\text{cost}(f) + d(T_R) \leq d(m)$, and it follows that SOLVE returns a value $\text{best} \leq \text{cost}(f) + d(T_R) \leq d(m)$, as required. ◀

Obviously, SOLVE can be easily adapted to return a solution of I with the cost it returns, but we have not taken this into account in Algorithm 2 for the sake of its readability.

6.4 Running time

Having a correct algorithm solving FIXED DEGREE SUBGRAPH WITH OUTBRANCHING in polynomial space, let us analyze its complexity depending on K .

Let us denote $r = |R|$ and $c = |L_{K+1}|$. Recall that $1 \leq r \leq n$ and $0 \leq c \leq \lfloor \frac{n-r}{K} \rfloor$.

If we fix r and c , then there are $\binom{n-1}{r-1}$ guesses for R (it has to contain root) and at most $\binom{r-1}{c}$ guesses for L_{K+1} . Let us bound the number of guesses for δ . By (C2) and (C3), $\sum_{v \in R} \delta_v = r - 1$, and $\delta_v = 0$ iff $v \in L_{K+1}$ so essentially we put $r - 1$ balls into $r - c$ bins that must be nonempty, which is $\binom{r-2}{c-1}$ by standard combinatorics. In the special case $c = 0$ there is one choice for δ , where $\delta_{\text{root}} = 0$.

In total, there are at most $\binom{n}{r} \binom{r}{c}^2$ guesses for all R, L_{K+1}, δ simultaneously. For each of these guesses, using Lemma 6.1 function SOLVE calculates an optimal δ -out-tree spanning R , which takes time $\mathcal{O}(4^{n+o(n)})$. It follows that that part takes time $\mathcal{O}^*(\sum_r \binom{n}{r} 4^{r+o(r)} \sum_c \binom{r}{c}^2)$. Then, SOLVE guesses a partition of $V \setminus R$ into $L_1, \dots, L_K$ in at most K^{n-r} ways. For each such guess, SOLVE spends polynomial time, so that part takes $\mathcal{O}^*(\sum_r \binom{n}{r} K^{n-r} \sum_c \binom{r}{c}^2)$ time. Hence the total running time can be bounded by

$$\mathcal{O}^* \left(2^{o(n)} \sum_{r=1}^n \sum_{c=0}^{\lfloor \frac{n-r}{K} \rfloor} \underbrace{\binom{n}{r} (K^{n-r} + 4^r) \binom{r}{c}^2}_{\xi(r,c)} \right).$$

Since there are polynomially many guesses for r and c , we can actually replace sums with maxima in the expression above and focus on the expression $\xi(r, c) = \binom{n}{r} (K^{n-r} + 4^r) \binom{r}{c}^2$.

We will heavily use the well-known bound $\binom{n}{\alpha n} < 2^{h(\alpha)n}$, where $h(\alpha) = -\alpha \log_2 \alpha - (1 - \alpha) \log_2 (1 - \alpha)$ is the binary entropy function (see e.g. [13]). For readability, let us denote $f(\alpha) = 2^{h(\alpha)}$ and let us point out that f is increasing on the interval $[0, \frac{1}{2}]$ and decreasing on the interval $[\frac{1}{2}, 1]$. Let us denote $\beta := \frac{r}{n}$. We are going to distinguish two cases here.

1. $\frac{n-r}{K} \geq \frac{r}{2}$

This inequality can be rephrased as $r \leq \frac{2}{K+2}n$, which is equivalent to $\beta \leq \frac{2}{K+2}$. We will use here a trivial bound $\binom{r}{c} \leq 2^r$. Then, $\xi(r, c) \leq f(\beta)^n ((K^{1-\beta})^n + 4^{\beta n}) 4^{\beta n} = (f(\beta)K^{1-\beta}4^\beta)^n + (f(\beta)4^{2\beta})^n$

2. $\frac{n-r}{K} < \frac{r}{2}$

In that case we know that $\max_{c=0}^{\lfloor \frac{n-r}{K} \rfloor} \binom{r}{c}^2$ is attained when $c = \lfloor \frac{n-r}{K} \rfloor$ and for that particular value of c we can use the following bound.

$$\begin{aligned} \binom{r}{c}^2 &= \left(\frac{\binom{r}{\lfloor \frac{n-r}{K} \rfloor}}{\binom{r}{r}} \cdot r \right)^2 = \mathcal{O}^* \left(f \left(\frac{\lfloor \frac{n-r}{K} \rfloor}{r} \right)^{2r} \right) = \mathcal{O}^* \left(f \left(\frac{\frac{n-r}{K}}{r} \right)^{2r} \right) = \\ &= \mathcal{O}^* \left(f \left(\frac{1-\beta}{K\beta} \right)^{2\beta n} \right) \end{aligned}$$

In the third equality above we used fact that f is increasing on the interval $[0, \frac{1}{2}]$. To sum up, in this case,

$$\xi(r, c) = \mathcal{O}^* \left(\left(f(\beta)K^{1-\beta} f \left(\frac{1-\beta}{K\beta} \right)^{2\beta} \right)^n + \left(f(\beta)4^\beta f \left(\frac{1-\beta}{K\beta} \right)^{2\beta} \right)^n \right).$$

Our numerical analysis shows that it is optimal to choose $K = 4$. For that particular value of K , the first case applies if and only if $\beta \leq \frac{1}{3}$. Then,

$$\xi(r, c) = (4f(\beta))^n + (4^{2\beta} f(\beta))^n = \mathcal{O}((4f(\beta))^n) = \mathcal{O}((4f(\frac{1}{3}))^n) = \mathcal{O}(7.56^n).$$

Let us now investigate the second case, when $\beta > \frac{1}{3}$. For $\frac{1}{3} < \beta < \frac{1}{2}$ the summand $\left(f(\beta)4^{1-\beta} f \left(\frac{1-\beta}{4\beta} \right)^{2\beta} \right)^n$ dominates, and for $\beta \geq \frac{1}{2}$ the summand $\left(f(\beta)4^\beta f \left(\frac{1-\beta}{4\beta} \right)^{2\beta} \right)^n$ dominates. We have numerically verified that

$$f(\beta)4^{1-\beta} f \left(\frac{1-\beta}{4\beta} \right)^{2\beta} \leq 7.68 \text{ for } \beta \in (\frac{1}{3}, \frac{1}{2})$$

and

$$f(\beta)4^\beta f \left(\frac{1-\beta}{4\beta} \right)^{2\beta} \leq 7.871 \text{ for } \beta \in [\frac{1}{2}, 1].$$

Hence, we can conclude that for $K = 4$ and our algorithm runs in time $\mathcal{O}^*(7.871^{n+o(n)}) = \mathcal{O}(7.88^n)$ and in polynomial space. This concludes the proof of Theorem 1.4.

7 (1 + ε)-approximation

In this section we show theorem 1.2, i.e. we present an algorithm for MANY VISITS TSP which finds a $(1 + \epsilon)$ -approximation in $\mathcal{O}^*(\frac{2^n}{\epsilon})$ time and polynomial space.

To achieve this we consider a more general problem, namely FIXED DEGREE CONNECTED SUBGRAPH. The main idea is to round weights of edges of the given instance, so that we can use the algorithm for polynomially bounded weights from Lemma 4.12 which is an analog of theorem 1.1 for FIXED DEGREE CONNECTED SUBGRAPH.

Let us first consider the case with degrees bounded by a polynomial.

► **Lemma 7.1.** *For given $\epsilon > 0$ and an instance $I = (d, \text{in}, \text{out})$ of FIXED DEGREE CONNECTED SUBGRAPH such that $\text{in}(v), \text{out}(v) \leq \mathcal{O}(n^2)$ for every vertex v there exists an algorithm finding a $(1 + \epsilon)$ -approximate solution in $\mathcal{O}^*\left(\frac{2^n}{\epsilon}\right)$ time and polynomial space.*

Proof. Let us denote the optimal solution for I by OPT. First, our algorithm guesses the most expensive edge used by OPT. Let us denote its cost by E , in particular

$$E \leq d(\text{OPT}). \quad (1)$$

Let us denote by C the universal constant such that $\text{in}(v), \text{out}(v) \leq Cn^2$ for every vertex v and let us round d in the following way

$$d'(u, v) := \begin{cases} \left\lceil \frac{Cn^3}{\epsilon E} d(u, v) \right\rceil & \text{if } d(u, v) \leq E \\ \infty & \text{if } d(u, v) > E \end{cases} \quad (2)$$

If $d'(u, v)$ is finite then it is bounded by $\left\lceil \frac{Cn^3}{\epsilon E} E \right\rceil = \left\lceil \frac{Cn^3}{\epsilon} \right\rceil$. Our algorithm simply returns the optimal solution for instance $I' = (d', \text{in}, \text{out})$ which can be found in $\mathcal{O}^*\left(\frac{2^n}{\epsilon}\right)$ time using the algorithm from Lemma 4.12 with $D = \left\lceil \frac{Cn^3}{\epsilon} \right\rceil$. Let us denote this solution by ALG. Now we only need to prove that ALG is a $(1 + \epsilon)$ -approximation for the original instance I . We know that ALG is an optimal solution for I' , in particular

$$d'(\text{ALG}) \leq d'(\text{OPT}). \quad (3)$$

For every v we have $\text{out}(v) \leq Cn^2$, so

$$\sum_{(u,v) \in V^2} \text{OPT}(u, v) = \sum_{u \in V} \text{out}(u) \leq n \cdot Cn^2. \quad (4)$$

The following chain of inequalities finishes the proof.

$$\begin{aligned} d(\text{ALG}) &\stackrel{(2)}{\leq} \frac{\epsilon E}{Cn^3} d'(\text{ALG}) \stackrel{(3)}{\leq} \frac{\epsilon E}{Cn^3} d'(\text{OPT}) \stackrel{(2)}{\leq} \frac{\epsilon E}{Cn^3} \sum_{(u,v) \in V^2} \text{OPT}(u, v) \left(\frac{d(u, v)Cn^3}{\epsilon E} + 1 \right) = \\ &= d(\text{OPT}) + \frac{\epsilon E}{Cn^3} \sum_{(u,v) \in V^2} \text{OPT}(u, v) \stackrel{(4)}{\leq} d(\text{OPT}) + \frac{\epsilon E}{Cn^3} Cn^3 \stackrel{(1)}{\leq} (1 + \epsilon)d(\text{OPT}) \quad \blacktriangleleft \end{aligned}$$

Now we can generalize the algorithm from Lemma 7.1 by using it as a black box for the general case.

► **Lemma 7.2.** *For a given $\epsilon > 0$ and an instance $I = (d, \text{in}, \text{out})$ of FIXED DEGREE CONNECTED SUBGRAPH there exists an algorithm finding a $(1 + \epsilon)$ -approximate solution in $\mathcal{O}^*\left(\frac{2^n}{\epsilon}\right)$ time and polynomial space.*

Proof. First let us use the algorithm from Theorem 3.4 which outputs an instance $I' = (d, \text{in}', \text{out}')$ of FIXED DEGREE CONNECTED SUBGRAPH and a function $f : V^2 \rightarrow \mathbb{Z}_{\geq 0}$. Let us denote the optimal solution for I' by OPT'. By Theorem 3.4 the optimal solution for I equals OPT' + f . Moreover, we know that $\text{in}', \text{out}'(v) \leq \mathcal{O}(n^2)$ for every vertex v . In particular we can use algorithm from Lemma 7.1 to get a solution ALG' for instance I' such that $d(\text{ALG}') \leq (1 + \epsilon)d(\text{OPT}')$. Our algorithm simply returns solution ALG' + f , which is a solution for I because ALG' is connected and f increases degrees exactly by difference between I and I' . To prove ALG' + f is $(1 + \epsilon)$ -approximation we just need to observe that

$$d(\text{ALG}' + f) \leq (1 + \epsilon)d(\text{OPT}') + d(f) \leq (1 + \epsilon)d(\text{OPT}' + f). \quad \blacktriangleleft$$

FIXED DEGREE CONNECTED SUBGRAPH is a generalization of MANY VISITS TSP so the algorithm from Lemma 7.2 proves Theorem 1.2.

8 Further Research

Since TSP is solvable in time $\mathcal{O}^*(2^n)$ and exponential space [1, 16] and time $\mathcal{O}(4^{n+o(n)})$ and polynomial space [15], the main remaining question is whether these bounds can be achieved for MANY VISITS TSP avoiding in the running time bound the linear dependence on maximum distance D . Another interesting goal is a deterministic version of Theorem 4.1.

References

- 1 Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *J. ACM*, 9(1):61–63, January 1962. doi:10.1145/321105.321111.
- 2 André Berger, László Kozma, Matthias Mnich, and Roland Vincze. Time- and space-optimal algorithms for the many-visits TSP. *CoRR*, abs/1804.06361, 2018. arXiv:1804.06361.
- 3 André Berger, László Kozma, Matthias Mnich, and Roland Vincze. A time- and space-optimal algorithm for the many-visits TSP. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1770–1782. SIAM, 2019. doi:10.1137/1.9781611975482.106.
- 4 Andreas Björklund, Petteri Kaski, and Ioannis Koutis. Directed Hamiltonicity and out-branchings via generalized Laplacians. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, volume 80 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 91:1–91:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ICALP.2017.91.
- 5 Andreas Björklund, Petteri Kaski, and Lukasz Kowalik. Fast witness extraction using a decision oracle. In *ESA*, volume 8737 of *Lecture Notes in Computer Science*, pages 149–160. Springer, 2014.
- 6 Nadia Brauner, Yves Crama, Alexander Grigoriev, and Joris Van de Klundert. A framework for the complexity of high-multiplicity scheduling problems. *J. Comb. Optim.*, 9:313–323, May 2005. doi:10.1007/s10878-005-1414-7.
- 7 Stavros S. Cosmadakis and Christos H. Papadimitriou. The traveling salesman problem with many visits to few cities. *SIAM J. Comput.*, 13(1):99–108, 1984.
- 8 Marek Cygan, Holger Dell, Daniel Lokshtanov, Dániel Marx, Jesper Nederlof, Yoshio Okamoto, Ramamohan Paturi, Saket Saurabh, and Magnus Wahlström. On problems as hard as CNF-SAT. *ACM Trans. Algorithms*, 12(3), May 2016. doi:10.1145/2925416.
- 9 Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Fast Hamiltonicity checking via bases of perfect matchings. *J. ACM*, 65(3), March 2018. doi:10.1145/3148227.
- 10 Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michał Pilipczuk, Johan M. M. van Rooij, and Jakub Onufry Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In *FOCS*, pages 150–159. IEEE Computer Society, 2011.
- 11 R. A. DeMillo and R. J. Lipton. A probabilistic remark on algebraic program testing. *Inf. Process. Lett.*, 7:193–195, 1978.
- 12 Hamilton Emmons and Kamlesh Mathur. Lot sizing in a no-wait flow shop. *Operations Research Letters - ORL*, 17:159–164, May 1995. doi:10.1016/0167-6377(95)00008-8.
- 13 Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2010.
- 14 Alexander Grigoriev and Joris Van de Klundert. On the high multiplicity traveling salesman problem. *Discrete Optimization*, 3:50–62, March 2006. doi:10.1016/j.disopt.2005.11.002.
- 15 Yuri Gurevich and Saharon Shelah. Expected computation time for Hamiltonian path problem. *SIAM J. Comput.*, 16(3):486–502, June 1987. doi:10.1137/0216034.
- 16 Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. In *Proceedings of the 1961 16th ACM National Meeting*, ACM ’61, page 71.201–71.204, New York, NY, USA, 1961. Association for Computing Machinery. doi:10.1145/800029.808532.

- 17 Dorit S. Hochbaum and Ron Shamir. Strongly polynomial algorithms for the high multiplicity scheduling problem. *Oper. Res.*, 39:648–653, 1991.
- 18 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001. doi:10.1006/jcss.2000.1727.
- 19 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001. doi:10.1006/jcss.2001.1774.
- 20 László Kozma and Tobias Mömke. Maximum scatter TSP in doubling metrics. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '17*, page 143–153, USA, 2017. Society for Industrial and Applied Mathematics.
- 21 Michael Lampis. Algorithmic meta-theorems for restrictions of treewidth. In *Algorithms – ESA 2010*, volume 64, pages 549–560, September 2010. doi:10.1007/978-3-642-15775-2_47.
- 22 László Lovász. On determinants, matchings and random algorithms. In *FCT*, volume 79, pages 565–574, January 1979.
- 23 James B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Oper. Res.*, 41(2):338–350, 1993. doi:10.1287/opre.41.2.338.
- 24 Harilaos N. Psaraftis. A dynamic programming approach for sequencing groups of identical jobs. *Operations Research*, 28(6):1347–1359, 1980.
- 25 Jacob T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980. doi:10.1145/322217.322225.
- 26 W. T. Tutte. The factorization of linear graphs. *Journal of the London Mathematical Society*, s1-22(2):107–111, April 1947. doi:10.1112/jlms/s1-22.2.107.
- 27 Jack A. A. van der Veen and Shuzhong Zhang. Low-complexity algorithms for sequencing jobs with a fixed number of job-classes. *Comput. Oper. Res.*, 23(11):1059–1067, November 1996. doi:10.1016/0305-0548(96)00016-0.
- 28 Richard Zippel. Probabilistic algorithms for sparse polynomials. In *Proc. International Symposium on Symbolic and Algebraic Computation*, volume 72 of *LNCS*, pages 216–226, 1979.

Determining 4-Edge-Connected Components in Linear Time

Wojciech Nadara ✉

Institute of Informatics, University of Warsaw, Poland

Mateusz Radecki ✉

University of Warsaw, Poland

Marcin Smulewicz ✉

Institute of Informatics, University of Warsaw, Poland

Marek Sokołowski ✉

Institute of Informatics, University of Warsaw, Poland

Abstract

In this work, we present the first linear time deterministic algorithm computing the 4-edge-connected components of an undirected graph. First, we show an algorithm listing all 3-edge-cuts in a given 3-edge-connected graph, and then we use the output of this algorithm in order to determine the 4-edge-connected components of the graph.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases graphs, connectivity, cuts

Digital Object Identifier 10.4230/LIPIcs.ESA.2021.71

Related Version Full Version: <https://arxiv.org/abs/2105.01699> [33]

Funding Wojciech Nadara: Supported by ERC Starting Grant CUTACOMBS (Grant Agreement No 714704).

Marcin Smulewicz: Supported by ERC Starting Grant TOTAL (Grant Agreement No 677651).

Marek Sokołowski: Supported by ERC Starting Grant BOBR (Grant Agreement No 948057).

1 Introduction

The connectivity of graphs has always been one of the fundamental concepts of graph theory. The foremost connectivity notions in undirected graphs are the k -edge-connectedness and the k -vertex-connectedness. Namely, a graph G is k -edge-connected for $k \geq 1$ if it is connected, and it remains connected after removing any set of at most $k - 1$ edges. Similarly, G is k -vertex-connected if it contains at least $k + 1$ vertices, and it remains connected after the removal of any set of at most $k - 1$ vertices.

These notions can be generalized to the graphs that are not well-connected. Namely, if H is a maximal k -vertex-connected subgraph of G , we say that H is a k -vertex-connected component of G . The edge-connected variant is, however, defined differently: we say that a pair of vertices u, v of G is k -edge-connected if it is not possible to remove at most $k - 1$ edges from G so that u and v end up in different connected components. This relation of k -edge-connectedness happens to be an equivalence relation; this yields a definition of a k -edge-connected component of G as an equivalence class of the relation. We remark that the notions of k -vertex-connected components and k -edge-connected components coincide for $k = 1$ as both simply describe the connected components of G . However, for $k \geq 2$ these definitions diverge; in particular, for $k \geq 3$ the k -edge-connected components of a graph do not even need to be connected.



© Wojciech Nadara, Mateusz Radecki, Marcin Smulewicz, and Marek Sokołowski;
licensed under Creative Commons License CC-BY 4.0
29th Annual European Symposium on Algorithms (ESA 2021).

Editors: Petra Mutzel, Rasmus Pagh, and Grzegorz Herman; Article No. 71; pp. 71:1–71:15

Leibniz International Proceedings in Informatics

LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

71:2 Determining 4-Edge-Connected Components in Linear Time

There has been a plethora of research into the algorithms deciding the k -vertex- and k -edge-connectedness of graphs, and decomposing the graphs into k -vertex- or k -edge-connected components. However, while classical, elementary, and efficient algorithms exist for $k = 1$ and $k = 2$, these problems become increasingly more difficult for the larger values of k . In fact, even for $k = 4$, there were no known linear time algorithms to any of the considered problems. The following description presents the previous work in this area for $k \in \{1, 2, 3, 4\}$, and exhibits the related work for the larger values of k :

k = 1. Here, the notions of k -vertex-connectedness and k -edge-connectedness reduce to that of connectivity and the connected components of a graph. In the static setting, determining the connected components in linear time is trivial. As a consequence, more focus is being laid on dynamic algorithms maintaining the connected components of graphs. In the incremental setting, where the edges can only be added to the dynamic graph, the optimal solution is provided by disjoint-set data structures [38], which solve the problem in the amortized $\mathcal{O}(\alpha(n))$ time per query, where $\alpha(n)$ denotes the inverse of the Ackermann's fast-growing function. The fully dynamic data structures are also considered [44, 7, 9, 18, 19, 20, 40, 27, 22, 25].

k = 2. One step further are the notions of 2-vertex-connectivity (biconnectivity) and 2-edge-connectivity. In the static setting, partitioning of a graph into 2-vertex-connected or 2-edge-connected components are classical problems, both solved in linear time by exploiting the properties of the low function [23]. The incremental versions of both problems are again solved optimally in the amortized $\mathcal{O}(\alpha(n))$ time per query [43]. Significant research has been done in the dynamic setting as well [36, 20, 7, 22, 11, 17, 31].

k = 3. As a next step, we consider 3-vertex-connectivity (triconnectivity) and 3-edge-connectivity. An optimal, linear time algorithm detecting the 3-vertex-connected components was first given by Hopcroft and Tarjan [24]. The first linear time algorithm for 3-edge-connectivity was discovered much later by Galil and Italiano [12], where they present a linear time reduction from the k -edge-connectivity problem to k -vertex-connectivity for $k \geq 3$, showing that in the static setting, the former problem is the easier of the two. This was later followed by a series of works simplifying the solution for 3-edge-connectivity [41, 35, 34, 42]. The incremental setting [32, 21] and the dynamic setting [11, 7] were also considered.

We also mention that in the case of 3-vertex-connectivity, there exists a structure called SPQR-tree which succinctly captures the structure of 2-vertex-cuts in graphs [4, 21]. Its edge-connectivity analogue also exists, but we defer its introduction to the general setting.

k = 4. We move on to the problems of 4-vertex-connectivity and 4-edge-connectivity. A notable result by Kanevsky et al. [26] supports maintaining 4-vertex-connected components in incremental graphs, with an optimal $\mathcal{O}(\alpha(n))$ amortized time per query. Their result also yields the solution for static graphs in $\mathcal{O}(m + n\alpha(n))$ time complexity. By applying the result of Galil and Italiano [12], we derive a static algorithm determining the 4-edge-connected components in the same time complexity. This algorithm is optimal for $m = \Omega(n\alpha(n))$.

Another result by Dinitz and Westbrook [6] supports maintaining the 4-edge-connected components in the incremental setting. Their algorithm processes any sequence of queries in $\mathcal{O}(q + m + n \log n)$ time where q is the number of queries, and m is the total number of inserted edges to the graph.

However, it is striking that the fastest solutions for 4-edge-connectivity and 4-vertex-connectivity for static graphs were derived from the on-line algorithms working in the incremental setting. In particular, no linear time algorithms for $k = 4$ were known before.

$k \geq 5$. As a side note, we also present the current knowledge on the general problems of k -vertex-connectivity and k -edge-connectivity. A series of results [29, 30, 16, 2] show that it is possible to compute the minimum edge cut of a graph (i.e., determine the edge-connectivity of a graph) in near-linear time. The previously mentioned work by Dinitz and Westbrook [6] maintains the k -edge-connected components of an incremental graph which is assumed to already have been initialized with a $(k-1)$ -edge-connected graph. The data structure answers any sequence of on-line queries in $\mathcal{O}(q + m + k^2 n \log(n/k))$ time, where q is the number of queries, and m is the number of edges in the initial graph.

Gomory and Hu [13] proved that for any weighted, undirected graph G there exists a weighted, undirected tree T on the same vertex set such that for any two vertices $s, t \in V(G)$, the value of the minimum s - t edge cut in T is equal to the value of the minimum s - t edge cut in G . Moreover, such a tree can be constructed using $n-1$ invocations of the maximum flow algorithm. In an interesting result by Hariharan et al. [15], the decomposition of any graph into k -edge-connected components is constructed in $\mathcal{O}((m + nk^3) \cdot \text{polylog}(n))$ time, producing a partial Gomory-Hu tree as its result.

Dinitz et al. [5] showed that the set of all minimum edge cuts can be succinctly represented with a *cactus graph*. When the minimum edge cut is odd, this cactus simplifies to a tree (see [8, Corollary 8]). These results imply that if the size of the minimum cut is odd, then the number of minimum cuts is $\mathcal{O}(n)$ and if it is even, then the number of minimum cuts is $\mathcal{O}(n^2)$. The structure of k -vertex-cuts was also investigated [37].

Our results. In this work, we present a linear time, deterministic algorithm partitioning static, undirected graphs into 4-edge-connected components. Even though the area of the dynamic versions of the algorithms for k -edge-connectivity is still thriving, the progress in static variants appears to have plateaued. In particular, both subquadratic algorithms determining the 4-edge-connected components [26, 6] originate from their dynamic incremental equivalents and are almost thirty years old, yet they did not achieve the optimal linear running time. Hence, our work constitutes the first progress in the static setting of 4-edge-connectivity in a long time. As a side result, our algorithm also produces the tree representation of 3-edge-cuts as explained in [8].

Organization of the work. In Section 3 [33], omitted in this abridged version of the work, we show how to reduce the problem of determining 4-edge-connected components to the problem of determining 4-edge-connected components in 3-edge-connected graphs. In Section 4, we show a linear time, randomized Monte Carlo algorithm for listing all 3-edge-cuts in 3-edge-connected graphs. In Section 5, we show how to remove the dependency on the randomness in the algorithm from the previous section, producing a linear time, deterministic algorithm listing all 3-edge-cuts in 3-edge-connected graphs. Then, in Section 6, we construct a tree of 3-edge-cuts in a 3-connected graph, given the list of all its 3-edge-cuts. This tree is then used to determine the 4-edge-connected components of the graph. Finally, in Section 7, we present open problems related to this work.

Section 3 and the proofs of the results marked ★ are deferred to the full version of the work [33] due to the space constraints.

2 Preliminaries

Graphs. In this work, we consider undirected, connected graphs which may contain self-loops and multiple edges connecting pairs of vertices (i.e., multigraphs). The number of vertices of a graph and the number of its edges are usually denoted n and m , respectively.

We use the notions of k -edge-connectedness and k -edge-connected components defined in Section 1. Moreover, we say that a set of k edges of a graph forms a k -edge-cut (or a k -cut for simplicity) if the removal of these edges from the graph disconnects it.

DFS trees. Consider a run of the *depth-first search* algorithm [39] on a connected graph G . A *depth-first search tree* (or a *DFS tree*) is a spanning tree $\mathcal{T}$ of G , rooted at the source of the search r , containing all the edges traversed by the algorithm. After the search is performed, each vertex v is assigned two values: its *preorder* $\text{pre}(v)$ (also called *discovery time* or *arrival time*) and *postorder* $\text{post}(v)$ (also *finishing time* or *departure time*). Their definitions are standard [3]; it can be assumed that the values range from 1 to $2n$ and are pairwise different.

The edges of $\mathcal{T}$ are called *tree edges*, and the remaining edges are called *back edges* or *non-tree edges*. In this setup, every back edge e connects two vertices remaining in ancestor-descendant relationship in $\mathcal{T}$; moreover, the graph $\mathcal{T} + e$ contains exactly one cycle, named the *fundamental cycle* of e . For a vertex v of G , we define $\mathcal{T}_v$ to be the subtree of $\mathcal{T}$ rooted at v ; similarly, for a tree edge e whose deeper endpoint is v , we set $\mathcal{T}_e = \mathcal{T}_v$.

When a DFS tree $\mathcal{T}$ of G is fixed, it is common to introduce directions to the edges of the graph: all tree edges of $\mathcal{T}$ are directed away from the root of $\mathcal{T}$, and all back edges are pointed towards the root of $\mathcal{T}$. Then, uv is a directed edge (either a tree or a back edge) whose *origin* (or *tail*) is u , and whose *destination* (or *head*) is v .

For our convenience, we introduce the following definition: a back edge $e = pq$ *leaps over* a vertex v if $p \in \mathcal{T}_v$, but $q \notin \mathcal{T}_v$; we analogously define *leaping over* a tree edge f .

Moreover, we define a partial order $\leq_{\mathcal{T}}$ on the vertices of G and the tree edges of $\mathcal{T}$ as follows: $x \leq_{\mathcal{T}} y$ if the simple path in $\mathcal{T}$ connecting the root of $\mathcal{T}$ with y also contains x . Then, $\leq_{\mathcal{T}}$ has one minimal element – the root of $\mathcal{T}$ – and each maximal element is a leaf of $\mathcal{T}$. When the tree $\mathcal{T}$ is clear from the context, we may write $\leq$ instead of $\leq_{\mathcal{T}}$. We may also use $x < y$ for $x \leq y \wedge x \neq y$. Using the precomputed preorder and postorder values in $\mathcal{T}$, we can verify if $x \leq_{\mathcal{T}} y$ holds for given $x, y \in V(G) \cup E(\mathcal{T})$ in constant time.

We use the classical low function defined by Hopcroft and Tarjan [23]. However, for our purposes it is more convenient to define it as a function $\text{low} : E(\mathcal{T}) \rightarrow (E \setminus E(\mathcal{T})) \cup \{\perp\}$ such that for a tree edge e , $\text{low}(e)$ is the back edge uv leaping over e minimizing the preorder of its head v , breaking ties arbitrarily; or $\perp$, if no such edge exists. This function can be computed for all tree edges in time linear with respect to the size of the graph.

Xors. For sets A and B , by $A \oplus B$ we denote their symmetric difference, which is $(A \cup B) \setminus (A \cap B)$, and we call it a *xor* of A and B . Moreover, for a pair of non-negative integers a and b , by $a \oplus b$ we denote their xor, that is, an integer whose binary representation is a bitwise symmetric difference of the binary representations of a and b . The definitions can be easily generalized to the symmetric differences of multiple sets or integers.

4 Simple randomized algorithm

In this section, we will describe a randomized linear time algorithm listing 3-edge-cuts in 3-edge-connected graphs. In particular, the existence of this algorithm will imply that the number of 3-edge-cuts in any 3-edge-connected graph is at most linear. Since the algorithm is significantly simpler than its deterministic variant, and it already contains most of the core ideas of this work, we believe it serves as a good intermediate step in the explanation.

The overview of the algorithm is as follows. First, we will construct a randomized oracle (Lemma 5) that given two edges e, f of the 3-edge-connected graph G , verifies in constant time whether there exists another edge g such that $\{e, f, g\}$ is a 3-edge-cut of G ; and if such

an edge exists, it is returned by the oracle. It will be apparent from the description of the oracle that such an edge – if it exists – is unique. Thanks to Lemma 5, the description of our randomized algorithm will be simplified significantly: now, we only need to identify two out of three edges of each 3-edge-cut of G . Then, the remaining edge of each cut can be easily recovered using a single oracle call.

Next, our algorithm will consider an arbitrary DFS tree $\mathcal{T}$ of G , categorize all 3-edge-cuts by the number $t \in \{0, 1, 2, 3\}$ of tree edges of $\mathcal{T}$ in the cut, and find all 3-edge-cuts separately for each value of t . The case $t = 0$ is trivial: $\mathcal{T}$ is a spanning tree of G , hence each edge cut must contain at least one edge of $\mathcal{T}$. The cases $t = 1$ and $t = 2$ will proceed by considering all tree edges of $\mathcal{T}$ separately. For each such tree edge e , we will find all 3-edge cuts containing both e and some non-tree edge leaping over e . By performing a case study, we will show that for each e , there is only a constant number of 3-edge-cuts of this form for $t \in \{1, 2\}$; and moreover, each of them can be located by us in constant time after a linear time preprocessing of $\mathcal{T}$. Finally, the case $t = 3$ will be solved by a recursive call to the algorithm on a properly contracted graph.

We begin the detailed explanation with the description of some auxiliary data structures.

► **Theorem 1** ([10]). *There exists a data structure for the disjoint set union problem which, when initialized with an undirected tree $\mathcal{T}$ (the “union tree”) on n vertices, creates n singleton sets. After the initialization, the data structure accepts the following queries in any order:*

- *find(x): returns the index of the set containing x ,*
- *union(x, y): if x and y are in different sets, then an arbitrary one of them is replaced with their union and the other one with the empty set. This query can only be issued if xy is an edge of $\mathcal{T}$.*

The data structure executes any sequence of q queries in total $\mathcal{O}(n + q)$ time.

► **Lemma 2** (★). *It is possible to enrich the data structure from Theorem 1, so that after rooting it at an arbitrary vertex, we are able to answer the following query in constant time:*

- *lowest(x): returns the smallest vertex of the set containing x with respect to $\leq_{\mathcal{T}}$.*

Note that each set induces a connected subgraph of $\mathcal{T}$, so its smallest vertex is well-defined.

► **Theorem 3** (★). *There exists a deterministic algorithm that takes as input:*

- *an undirected, unrooted tree $\mathcal{T}$ with n vertices,*
- *p weighted paths $P_1, P_2, \dots, P_p$ in the tree, where the path P_i has weight $w_i \in \{0, 1, \dots, C\}$,*
- *and a positive integer k ,*

and for each edge e of the tree returns the indices of k paths with the lowest weight containing e , breaking ties arbitrarily; if e is a part of fewer than k paths, all such paths are returned. The time complexity of the algorithm is $\mathcal{O}(nk + p + C)$.

We proceed to the description of our randomized algorithm. Let us choose an arbitrary vertex r of the graph, and perform a depth-first search from r . Let $\mathcal{T}$ be the resulting DFS tree. We are now going to define a *hashing function* $H : E \rightarrow \mathcal{P}(E)$, where $\mathcal{P}(E)$ denotes the powerset of E ; the value $H(e)$ will be called a *hash* of e . If $e \notin \mathcal{T}$, then we define $H(e) = \{e\}$. Otherwise, we take $H(e)$ as the set of non-tree edges leaping over e . Let us note that the set C_e of edges f such that $e \in H(f)$ forms a cycle – the fundamental cycle of e .

► **Lemma 4** (★). *For a connected graph $G = (V, E)$ and a subset A of its edges, the graph $G' := G - A$ is disconnected if and only if there is a nonempty subset $B \subseteq A$ such that xor of the hashes of the edges in B is an empty set.*

71:6 Determining 4-Edge-Connected Components in Linear Time

Since our graph has no 1-edge-cuts or 2-edge-cuts, no created hashes are empty, and no two edges have equal hashes. Hence, removing a set of three edges disconnects a 3-edge-connected graph if and only if xor of the hashes of all of them is the empty set. Moreover, after removing some 3-edge-cut, the graph disconnects into exactly two components, and no removed edge connects vertices within one component.

As storing hashes as sets of edges would be inefficient, we define *compressed hashes*, resembling the notion of sketches introduced by Ahn et al. [1] and Kapron et al. [28]. We express compressed hashes as b -bit numbers where $b = \lceil 3 \log_2(m) \rceil$, i.e., as a function $CH : E \rightarrow \{0, \dots, 2^b - 1\}$. For each non-tree edge e , we draw $CH(e)$ randomly and uniformly from the set of b -bit numbers. For a tree edge e , we define its compressed hash $CH(e)$ as the xor of the compressed hashes of the edges in its hash, i.e., $CH(e) = \bigoplus_{f \in H(e)} CH(f)$. Note that since $b = \mathcal{O}(\log m)$, we can perform arithmetic operations on compressed hashes in constant time. However, this comes at a cost of allowing the collisions of compressed hashes.

For the ease of exposition, in the description of this algorithm we will use hash tables; because of that, the algorithm described in this section will have expected linear instead of worst-case linear running time. Formally, we create a hash table $M : \{0, \dots, 2^b - 1\} \rightarrow E \cup \{\perp\}$, which for any b -bit number x returns an edge whose compressed hash is equal to x ; or $\perp$ if no such edge exists. In the unlikely event that there are multiple edges whose compressed hashes are equal to x , M returns any of them.

We will now categorize 3-edge-cuts based on the number of tree edges they contain, and show how to handle each case. We remark that each 3-edge-cut must intersect $\mathcal{T}$ as it is a spanning tree of G . Therefore, a 3-edge-cut may intersect $\mathcal{T}$ in one edge (Subsection 4.1), two edges (Subsection 4.2), or three edges (Subsection 4.3).

Throughout the case analysis, we will heavily rely on the following fact:

► **Lemma 5.** *Given two edges e and f of some 3-edge-cut in a 3-edge-connected graph, the remaining edge is uniquely identified by its hash: $H(e) \oplus H(f)$.*

Proof. If $\{e, f, g\}$ is a 3-edge-cut, then $H(e) \oplus H(f) \oplus H(g) = \emptyset$, so $H(g) = H(e) \oplus H(f)$. ◀

4.1 One tree edge

Let e be an edge of $\mathcal{T}$ that is the only tree edge of some 3-edge-cut. The two resulting connected components after the removal of such a cut are $\mathcal{T}_e$ and $\mathcal{T} \setminus \mathcal{T}_e$. As e is not a bridge, $\text{low}(e)$ is well-defined and connects $\mathcal{T}_e$ with $\mathcal{T} \setminus \mathcal{T}_e$, so it belongs to the cut as well. By Lemma 5, this uniquely determines the third edge of this cut. Thus, in order to detect all such cuts, we iterate over all tree edges e and for each of them, we look up in M if there exists an edge with compressed hash equal to $CH(e) \oplus CH(\text{low}(e))$. If it exists and if it turns out to be a non-tree edge g , we output the triple $\{e, \text{low}(e), g\}$ as a 3-edge-cut.

4.2 Two tree edges

Let e and f be the edges of $\mathcal{T}$ that form a 3-edge-cut c together with some back edge g . Without loss of generality, we can assume that $e <_{\mathcal{T}} f$, thanks to the following fact:

► **Lemma 6 (★).** *The edges e and f are comparable with respect to $\leq_{\mathcal{T}}$.*

The two connected components of $G \setminus c$ are $L := \mathcal{T}_e \setminus \mathcal{T}_f$ and $R := \mathcal{T} \setminus L$, and the remaining back edge g connects L and R . We distinguish two cases, differing in the location of g (see Figure 2 from the full version of this work): *two tree edges, lower case*, where g connects L with $\mathcal{T}_f$, and *two tree edges, upper case*, where g connects L with $\mathcal{T} \setminus \mathcal{T}_e$.

Two tree edges, lower case. Let A be the set of back edges between $\mathcal{T}_f$ and $\mathcal{T} \setminus \mathcal{T}_f$. It consists of the edge g , connecting L with $\mathcal{T}_f$, and of several edges connecting $\mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_f$. Let B be the set of heads of edges from A (we remind that back edges are directed towards the root r). All elements of B lie on the path from f to the root r , and the head of g is the deepest element of B .

Using this observation, we will use the algorithm from Theorem 3. We initialize an instance of it with the tree $\mathcal{T}$, $k = 1$, $C = 2n$. Then, for each back edge $e = xy$, we create one input path P_e from x to y of weight $w_e := 2n - \text{pre}(y)$. The algorithm determines, for each tree edge f , the back edge leaping over f with the largest preorder of its head; call this edge $\text{MaxUp}(f)$. This back edge is the only candidate for g , given f .

Hence, we can find all such cuts by firstly initializing the data structure, and then iterating over all tree edges f . For each f , we take $g = \text{MaxUp}(f)$. Knowing f and g , we can look up in M whether there exists an edge whose compressed hash is $CH(f) \oplus CH(g)$ (Lemma 5). If it exists and if it is a tree edge, then we call it e and output the triple $\{e, f, g\}$ as a 3-edge-cut.

Two tree edges, upper case. Let A be the set of non-tree edges between $\mathcal{T}_e$ and $\mathcal{T} \setminus \mathcal{T}_e$. It consists of the back edge g , connecting L with $\mathcal{T} \setminus \mathcal{T}_e$, and of several edges connecting $\mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_e$. Let B be the set of the tails of the edges from A . Let $v \in L$ be the tail of g . As $v \notin \mathcal{T}_f$, then we either have $\text{pre}(v) < \min_{u \in \mathcal{T}_f} \text{pre}(u)$, or $\text{pre}(v) > \max_{u \in \mathcal{T}_f} \text{pre}(u)$. In the first case, v is the vertex with the smallest preorder which is a tail of some edge leaping over e ; while in the second case v is the analogous vertex with the largest preorder.

Based on this observation, we will again use the algorithm from Theorem 3 again. We initialize one instance of it with the tree $\mathcal{T}$, $k = 1$, and $C = 2n$. Then for each back edge $e = xy$, we create one input path P_e from x to y of weight $w_e := \text{pre}(x)$. We also initialize another instance of this algorithm in the same way, only that we set $w_e := 2n - \text{pre}(x)$ instead. The first instance determines, for each tree edge e , the back edge $\text{MinDn}(e)$ leaping over e with the smallest preorder of its tail; while the second instance determines the analogous edge $\text{MaxDn}(e)$, with the largest preorder of its tail. By our considerations above, if any desired $\{e, f, g\}$ cut exists, then $g \in \{\text{MinDn}(e), \text{MaxDn}(e)\}$.

Hence, we can find all such cuts by firstly initializing both instances of the data structure. Then, we iterate over all tree edges e , and for each e we check both candidates for g . If our hash table contains an edge whose compressed hash is $CH(e) \oplus CH(g)$ and it is a tree edge, then we call it f and output the triple $\{e, f, g\}$ as a 3-edge-cut.

4.3 Three tree edges

Solving this case in a way similar to the previous cases seems intractable. We consider this subsection, together with the time analysis following it, as one of the key ideas of this work.

In this case, we assume that no non-tree edges belong to the cut. Therefore, we can contract all of them simultaneously and recursively list all 3-edge-cuts in the resulting graph. Since 3-edge-cuts in the contracted graph exactly correspond to the 3-edge-cuts consisting solely of tree edges in the original graph, this reduction is sound. The contraction is performed in $\mathcal{O}(n + m)$ time by identifying the vertices within the connected components of $G - E(\mathcal{T})$. Let G' be the graph after these contractions. Since G is 3-edge-connected, G' is 3-edge connected as well, so the assumption about the input graph being 3-edge-connected is preserved. We do not modify the value of b in the subsequent recursive calls, even though the value of m decreases.

4.4 Time and correctness analysis

We will now compute the expected time complexity of our algorithm. The subroutines from Subsections 4.1 and 4.2 clearly take expected linear time. The only nontrivial part of the analysis is the recursion in Subsection 4.3. Let $T(m)$ denote the maximum expected time our algorithm needs to solve any graph with at most m edges. Since our graph is 3-edge-connected, the degree of each vertex in G is at least 3, so $|E(G)| \geq \frac{3}{2}|V(G)| \geq \frac{3}{2}|E(G')| \Rightarrow |E(G')| \leq \frac{2}{3}|E(G)|$. Therefore, we have that $T(m) \leq \mathcal{O}(m) + T(\frac{2}{3}m)$. The solution to this recurrence is $T(m) = \mathcal{O}(m)$, hence the whole algorithm runs in expected linear time.

We proceed to proving that our algorithm works correctly with sufficiently high probability. The only reason it can output wrong result comes from the compression of hashes – if we used their uncompressed version instead, the algorithm would clearly be correct.

The maximum number of queries to our hash table is linear in terms of n , which follows from a similar argument to the one presented in Subsection 4.4. Hence, there exists some absolute constant c such that the number of queries is bounded by cn . For each query with value q , and for each edge whose hash is not equal to the hash whose compressed version we ask about, there is 2^{-b} probability that compressed version of this hash is equal to q . Hence, the probability that we ever get a false positive is bounded from above by $cnm2^{-b}$. Since $b = \lceil 3 \log_2(m) \rceil$ and $n \leq m$, we infer that $cnm2^{-b} \leq \frac{c}{m}$. Therefore, our algorithm works correctly with probability at least $1 - \frac{c}{m}$.

5 Deterministic algorithm

Having established a linear time randomized algorithm producing 3-edge-cuts in 3-edge-connected graphs, we will now determinize it by designing deterministic implementations of the subroutines for each of the cases considered in the randomized algorithm. Three cases need to be derandomized: “one tree edge” (Subsection 4.1), “two tree edges, lower case”, and “two tree edges, upper case” (Subsection 4.2). In the following description, we cannot use compressed hashes anymore: the compression is a random process which inevitably results in false positives. Instead, we will exploit additional properties of 3-edge-cuts in order to produce an efficient deterministic implementation of the algorithm.

Recall that in the description of the randomized implementation of the algorithm, we defined the values $\text{low}(e)$, $\text{MaxUp}(e)$, $\text{MinDn}(e)$, and $\text{MaxDn}(e)$ for any tree edge e . For the deterministic variant of the algorithm, we generalize these notions: we define $\text{low1}(e)$, $\text{low2}(e)$, and $\text{low3}(e)$ as the three back edges leaping over e with the minimal preorders of their targets; in particular, we set $\text{low1}(e) := \text{low}(e)$. We analogously define $\text{MaxUp1}(e)$, $\text{MaxUp2}(e)$, $\text{MinDn1}(e)$, $\text{MinDn2}(e)$, $\text{MaxDn1}(e)$, and $\text{MaxDn2}(e)$. We remark that $\text{low3}(e)$ might not exist if there are fewer than three edges leaping over e ; in this case, we put $\text{low3}(e) := \perp$. However, all the other values must exist – otherwise, at most one edge would leap over e , which would mean that this edge, together with e , would form a 2-edge-cut of G .

It is straightforward to compute all the values defined above in linear time; for instance, the generalizations of MaxUp , MinDn , and MaxDn can be determined by passing $k = 2$ to the algorithm in Theorem 3 instead of $k = 1$. Hence, in the following description, we will assume that all the values above have already been computed.

5.1 One tree edge

Recall that in this case, we are to find all 3-edge-cuts intersecting a fixed depth-first search tree $\mathcal{T}$ in a single edge. This is fairly straightforward: if some 3-edge-cut C contains exactly one tree edge e of $\mathcal{T}$, then the border of the cut is exactly $\mathcal{T}_e$; hence, this cut must include

all back edges leaping over e . Therefore, e belongs to a 3-edge-cut of this kind if and only if $\text{low3}(e) = \perp$, i.e. if there only exist two back edges leaping over e ; in this case, e and these two edges form a 3-edge-cut. Since this check can be easily performed in $\mathcal{O}(n)$ time for all tree edges in $\mathcal{T}$, the whole subroutine runs in $\mathcal{O}(n + m)$ time complexity.

5.2 Two tree edges, lower case

Recall that this case requires us to find all 3-edge-cuts intersecting a fixed depth-first search tree $\mathcal{T}$ in two edges, say e and f , such that $e < f$ and the remaining back edge g connects $\mathcal{T}_f$ with $\mathcal{T}_e \setminus \mathcal{T}_f$. Hence, g is the only back edge connecting $\mathcal{T}_f$ with $\mathcal{T}_e \setminus \mathcal{T}_f$, no back edges connect $\mathcal{T}_e \setminus \mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_e$, but there may be multiple back edges connecting $\mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_e$.

We shall exploit the fact that in a valid 3-edge-cut of this kind, all back edges leaping over e must originate from $\mathcal{T}_f$. This observation severely limits the set of possible tree edges f . This is formalized by the notion of the *deepest down cut* of e :

► **Definition 7** (deepest down cut). *For a tree edge e in $\mathcal{T}$, we define the deepest down cut of e , denoted $\text{DeepestDnCut}(e)$, as the deepest tree edge $f \geq e$ for which all back edges leaping over e originate from $\mathcal{T}_f$.*

► **Lemma 8** (★). *For every tree edge e , $\text{DeepestDnCut}(e)$ is defined correctly and uniquely. Moreover, every 3-edge-cut $\{e, f, g\}$ of the considered kind satisfies $e < f \leq \text{DeepestDnCut}(e)$.*

► **Lemma 9** (★). *For every tree edge e , $\text{DeepestDnCut}(e) = uv$ is the tree edge whose head v is the lowest common ancestor of two vertices: the tails of $\text{MinDn1}(e)$ and $\text{MaxDn1}(e)$.*

Thanks to Lemma 9, we can compute $\text{DeepestDnCut}(e)$ in constant time for each tree edge e as the lowest common ancestor of two vertices can be computed in constant time after linear preprocessing [14].

We now shift our focus to the lower tree edge f . As in Section 4, the only possible candidate g for a back edge of the cut leaping over f is given by $\text{MaxUp1}(f)$. The only problematic part is locating the remaining tree edge e . Previously, we utilized randomness in order to calculate the compressed hash of e given the compressed hashes of f and g . Here, we instead use the following fact:

► **Lemma 10** (★). *If $\{e, f, g\}$ is a 3-edge-cut of the considered kind for some tree edges e and $f > e$, then e is the deepest tree edge satisfying $\text{DeepestDnCut}(e) \geq f$.*

Lemmas 8 and 10 naturally lead to the following idea: given a tree edge e , the set of edges f satisfying $\text{DeepestDnCut}(e) \geq f$ is a path P_e connecting the head of e with the head of $\text{DeepestDnCut}(e)$; we assign this path a weight w_e equal to the depth of e in $\mathcal{T}$. We then invoke Theorem 3 with the tree $\mathcal{T}$, the weighted paths $\{P_e \mid e \in E(\mathcal{T})\}$, and $k = 1$. This lets us find, in $\mathcal{O}(n)$ time, for each tree edge $f \in E(\mathcal{T})$, the path P_e of the maximum weight containing f as an edge. This path naturally corresponds to the tree edge e from Lemma 10. This way, for every tree edge f , we have uniquely identified a back edge g and a tree edge e such that the only possible 3-edge-cut containing f as a deeper tree edge is $\{e, f, g\}$.

It only remains to verify that $\{e, f, g\}$ is a 3-edge-cut. Since $\text{DeepestDnCut}(e) \geq f$ guarantees that all back edges leaping over e originate from $\mathcal{T}_f$, we only need to check that exactly one edge (that is, g) connects $\mathcal{T}_f$ with $\mathcal{T}_e \setminus \mathcal{T}_f$. As $g = \text{MaxUp1}(f)$, it suffices to verify that the edge $\text{MaxUp2}(f)$, which is a back edge leaping over f whose head is the deepest apart from g , also leaps over e (i.e., it does not terminate in $\mathcal{T}_e$).

It can be easily verified that the implementation of the subroutine is deterministic and runs in linear time with respect to the size of G .

5.3 Two tree edges, upper case

Recall that in this case, we are required to find all 3-edge-cuts intersecting $\mathcal{T}$ in two edges e and f , such that $e < f$ and the remaining back edge g connects $\mathcal{T}_e \setminus \mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_e$. Similarly to the randomized case, we use the fact that given a tree edge e , the tail of g has either the smallest preorder (i.e., $g = \text{MinDn1}(e)$) or the largest preorder (i.e., $g = \text{MaxDn1}(e)$) among all the back edges leaping over e . Without loss of generality, assume that $g = \text{MinDn1}(e)$; the latter case is analogous.

In a similar vein to previous case, observe that if $\{e, f, g\}$ is a 3-edge-cut for some $f > e$, then all back edges leaping over e other than g must originate from $\mathcal{T}_f$.

This leads to the following slight generalization of DeepestDnCut (Definition 7):

► **Definition 11.** *For a tree edge e in $\mathcal{T}$, we define the value $\text{DeepestDnCutNoMin}(e)$ as the deepest tree edge $f \geq e$ for which all back edges leaping over e other than $\text{MinDn1}(e)$ originate from $\mathcal{T}_f$.*

The process of computation of $\text{DeepestDnCut}(e)$ asserted by Lemma 9 can be easily modified to match our needs: we take $\text{DeepestDnCutNoMin}(e)$ as the tree edge whose head is the lowest common ancestor of the tails of $\text{MinDn2}(e)$ and $\text{MaxDn1}(e)$. We refer the reader to Figure 4 from the full version of the work for a helpful picture.

Now, fix the shallower tree edge e of the cut. Let $f_0 := \text{DeepestDnCutNoMin}(e)$. Then, if some tree edge $f > e$ belongs to the 3-edge-cut $\{e, f, g\}$, then $f \leq f_0$ (otherwise, there would be multiple edges connecting $\mathcal{T}_e \setminus \mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_e$). The natural question is then: is $\{e, f_0, g\}$ a 3-edge-cut? The only possible problem is that there may exist back edges connecting $\mathcal{T}_{f_0}$ with $\mathcal{T}_e \setminus \mathcal{T}_{f_0}$. Fortunately, if this is the case, then the back edge $\text{MaxUp1}(f_0)$, dependent only on f_0 , is one of these back edges.

Hence, let $h_0 := \text{MaxUp1}(f_0)$. If h_0 leaps over e , we are done, and $\{e, f_0, g\}$ is an edge cut. Otherwise, the subtree $\mathcal{T}_f$ for the sought 3-edge-cut $\{e, f, g\}$ must contain the head of h_0 (or else h_0 would connect $\mathcal{T}_f$ with $\mathcal{T}_e \setminus \mathcal{T}_f$). Let then $f_1, e \leq f_1 < f_0$ be the deepest tree edge containing the head of h_0 , that is, the edge whose head coincides with the head of h_0 . Then, the edge f of the 3-edge-cut $\{e, f, g\}$ must satisfy $e < f \leq f_1$. We can then repeat this procedure: given f_1 , we compute $h_1 := \text{MaxUp1}(f_1)$, and either h_1 leaps over e and we are done, or we calculate another edge $f_2 < f_1$ supplying a better bound on the depth of f .

Since the graph is finite, this process terminates in $k = \mathcal{O}(n)$ steps, producing the lowest tree edge $f_k \geq e$ such that no back edge connects $\mathcal{T}_{f_k}$ with $\mathcal{T}_e \setminus \mathcal{T}_{f_k}$, and no back edge other than g connects $\mathcal{T}_e \setminus \mathcal{T}_{f_k}$ with $\mathcal{T} \setminus \mathcal{T}_e$. If $f_k = e$, then no 3-edge-cut $\{e, f, g\}$ exists for $f > e$. Otherwise, $\{e, f_k, g\}$ is naturally a correct 3-edge-cut. Since two edges of a 3-edge-cut uniquely identify the third edge (Lemma 5), we conclude that this is the only 3-edge-cut containing e and g . We refer the reader to Figure 5 from the full version of the paper for a helpful picture depicting the iterative process above.

In order to optimize the algorithm, we seek to optimize the iterative process described above. Indeed, for each $e \in E(\mathcal{T})$, the process is very similar: start with some edge $f > e$, and then repeatedly replace f with a higher edge, until a replacement would result in an edge closer to the root than e . We can model this process with a rooted tree $\mathcal{U}$ defined as follows:

- the vertices of $\mathcal{U}$ are the tree edges of $\mathcal{T}$, and $\perp$,
- $\perp$ is the root of $\mathcal{U}$,
- for a non-root vertex e of $\mathcal{U}$, its parent is the tree edge with the same head as $\text{MaxUp1}(e)$.

If the head of $\text{MaxUp1}(e)$ coincides with the root of $\mathcal{T}$, then the parent of e in $\mathcal{U}$ is $\perp$. Naturally, $\mathcal{U}$ can be constructed in linear time with respect to the size of $\mathcal{T}$; moreover, if an edge p is a parent of another edge q in $\mathcal{U}$, then $p <_{\mathcal{T}} q$.

Now, the iteration is equivalent to the repeated replacement of f_0 with its parent in $\mathcal{U}$ as long as the parent is greater or equal than e with respect to $<_{\mathcal{T}}$. In other words, the final edge $f' := \text{final}(f_0, e)$ is taken as the shallowest ancestor of f_0 in $\mathcal{U}$ for which $f' \geq_{\mathcal{T}} e$.

This leads to the final idea: we simulate a forest of rooted subtrees of $\mathcal{U}$ using a disjoint set union data structure $F_{\mathcal{U}}$ (Theorem 1). Each subtree additionally keeps its root, which can be retrieved from $F_{\mathcal{U}}$ (Lemma 2). Initially, each subtree of $\mathcal{U}$ contains a single vertex.

After the initialization of $F_{\mathcal{U}}$, we iterate e over the tree edges of $\mathcal{T}$ in the decreasing order of depth in $\mathcal{T}$. Throughout the process, we maintain the following invariant on $F_{\mathcal{U}}$: an edge $ef \in E(\mathcal{U})$ for $e < f$ has been added to the forest if and only if e has been considered at any previous iteration as the shallower tree edge of the cut. Hence, at the beginning of the iteration for a given edge e , we add to $F_{\mathcal{U}}$ all tree edges of $\mathcal{U}$ originating from e . At this point of time, for every tree edge f such that $f >_{\mathcal{T}} e$, the edge $\text{final}(f, e)$ is given by $F_{\mathcal{U}}.\text{lowest}(f)$. This reduces the entire iterative process described above to a single **lowest** query on $F_{\mathcal{U}}$.

We initialized $F_{\mathcal{U}}$ on a tree with n vertices, and we issued $\mathcal{O}(n)$ queries to it in total. Therefore, the whole subroutine runs in time linear with respect to the size of G .

Summing up, we replaced each randomized subroutine with its deterministic counterpart, preserving the linear guarantee on the runtime of the algorithm. We conclude that there exists a deterministic linear time algorithm listing 3-edge-cuts in 3-edge-connected graphs.

6 Reconstructing the structure of 4-edge-connected components

In this section, we show how to build a structure of 4-edge-connected components of a 3-edge-connected graph G , given the set $\mathcal{C}$ of all 3-edge-cuts in G .

First of all, recall what such a structure looks like.

► **Theorem 12.** [8, Corollary 8] *For a 3-edge-connected graph $G = (V, E)$, there exists a tree $H = (U, F)$ and functions $\phi : \mathcal{C} \rightarrow F$ and $\psi : V \rightarrow U$, such that ϕ is a bijection from 3-edge-cuts of G to the edges of H , and ψ maps (not necessarily surjectively) vertices of G to the vertices of H so that whole 4-edge-connected components are mapped to the same vertex.*

Moreover, if a 3-edge-cut c partitions the vertices of G into two parts V_1 and V_2 , then $\phi(c)$ partitions the vertices of H into U_1 and U_2 such that $\psi^{-1}(U_1) = V_1$ and $\psi^{-1}(U_2) = V_2$.

We refer the reader to Figure 6 in the full version for an illustrative example of a decomposition postulated by Theorem 12.

The tree H is usually unrooted in the literature. However, we are going to root it. Namely, we take a depth-first search tree $\mathcal{T}$ of G , rooted at some vertex r , and we root H at $\psi(r)$. For a vertex $u \in U$, we let H_u denote the subtree of H rooted at u .

► **Definition 13.** *For a 3-edge-cut c , we define $P(c)$ as the set of vertices from the connected component of $G \setminus c$ not containing r .*

We remark that since c is a minimal cut, $G \setminus c$ consists of two connected components, so $P(c)$ is determined uniquely.

► **Lemma 14 (★).** *Let $v, u \in U$, and let $e_1, e_2, \dots, e_k \in F$ be the sequence of edges of H on the path from v and u in H . If v is an ancestor of u , then for each pair of integers i, j such that $1 \leq i < j \leq k$, we have $|P(\phi^{-1}(e_i))| > |P(\phi^{-1}(e_j))|$.*

► **Lemma 15 (★).** *Given a 3-edge-connected graph G and the set of all its 3-edge-cuts $\mathcal{C}$, the sizes of $P(c)$ for all $c \in \mathcal{C}$ can be computed in linear time with respect to the size of G .*

71:12 Determining 4-Edge-Connected Components in Linear Time

To reconstruct H , we will also use the following structural lemma about cuts sharing the same edge of graph G .

► **Lemma 16 (★).** *For an edge $(u, v) = e \in E$, let $l(e)$ be the set of all 3-edge-cuts containing e . The image $\phi(l(e))$, i.e., the set of all edges of H corresponding to the edge cuts containing e , forms a path in H between $\psi(u)$ and $\psi(v)$.*

We remark an edge case in Lemma 16: if $\psi(u) = \psi(v)$ for some edge $e = (u, v)$, then the image $\phi(l(e))$ is empty. Moreover, since each 3-edge-cut $c \in \mathcal{C}$ contains at least one tree edge of $\mathcal{T}$, each edge $\phi(c)$ is covered by at least one path $\phi(l(e))$ for $e \in E(\mathcal{T})$. Equivalently, H is a tree, rooted at $\psi(r)$, equal to the union of all paths $\phi(l(e))$ for $e \in E(\mathcal{T})$. This representation of H is the cornerstone of our algorithm reconstructing H from G and $\mathcal{C}$.

► **Lemma 17.** *There exists a linear time algorithm which, given a graph G and the list $\mathcal{C}$ of all 3-edge-cuts of G , constructs the tree H , along with the mappings ϕ and ψ .*

Sketch of the proof. For each edge $e \in \mathcal{T}$, create a list $l(e)$ of all 3-edge-cuts c containing e , sorted decreasingly by the size of $P(c)$ using radix sort. Let $e_1, \dots, e_{n-1} \in E(\mathcal{T})$ be the sequence of edges visited by a depth-first search of $\mathcal{T}$ and let $e_i = u_i v_i$, where u_i is the vertex of $\mathcal{T}$ closer to the root r . We create H iteratively; initially, H is a single vertex $\psi(r)$. We maintain the following invariant after k iterations of the algorithm: H is a connected tree, rooted at $\psi(r)$, equal to the union of all paths $\phi(l(e_i))$ for $i \in \{1, 2, \dots, k\}$.

It is clear that after $n - 1$ iterations, H will be the required rooted tree. Consider the k -th iteration of the algorithm, $k \in [1, n - 1]$, in which we need to add to H the path $\phi(l(e_k))$, originating from $\psi(u_k)$ and terminating at $\psi(v_k)$. It can be proved that $\psi(u_k)$, together with the vertical path connecting it with the root of H , is already in H . Now, adding the path $\phi(l(e_k))$ to H is rather straightforward: first, starting from $\psi(u_k)$, we go up the tree H along the edges of H corresponding to the edge cuts containing e_k . Then, we proceed down the tree: we iterate the list $l(e_k)$ of cuts, excluding the cuts corresponding to the edges visited in the first part of the traversal. For each such cut, we go down the tree along the edge corresponding to the cut (creating it, if necessary). Each 3-edge-cut c is considered in only a constant number of iterations of the algorithm: an edge of H corresponding to c is traversed by the path $\phi(l(e_k))$ if and only if $e_k \in c$. Therefore, the time complexity of all iterations in total is $\mathcal{O}(m + |\mathcal{C}|) = \mathcal{O}(m + n)$. ◀

This concludes the construction of a tree representing all 3-edge-cuts in G . As a result, each vertex of H , as long as it is not empty, contains a single 4-edge-connected component of G . Hence, this algorithm also computes the decomposition of a 3-edge-connected graph G into 4-edge-connected components in total linear time.

7 Open problems

As a natural open problem whose resolving would complement this result nicely, we suggest investigating if it is possible to extend our result to vertex connectivity or to the higher-order edge connectivity.

Problem 1. Given an undirected graph $G = (V, E)$, is it possible to find all 4-vertex-connected components of G in linear time?

Problem 2. Given an undirected graph $G = (V, E)$, is it possible to find all 5-edge-connected components of G in linear time?

We also remark that our algorithm assumes the word RAM model in which we can perform any arithmetic and bitwise operations on pointers and $\mathcal{O}(\log n)$ -bit words in constant time; this is required by the linear time data structure of Gabow and Tarjan (Theorem 1, [10]). The natural question is whether this assumption can be avoided.

Problem 3. Given an undirected graph $G = (V, E)$, is it possible to find all 4-edge-connected components of G in linear time in the pointer machine model?

References

- 1 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing Graph Structure via Linear Measurements. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, page 459–467, USA, 2012. Society for Industrial and Applied Mathematics. doi:10.1137/1.9781611973099.40.
- 2 Nalin Bhardwaj, Antonio Molina Lovett, and Bryce Sandlund. A Simple Algorithm for Minimum Cuts in Near-Linear Time. In Susanne Albers, editor, *17th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2020, June 22–24, 2020, Tórshavn, Faroe Islands*, volume 162 of *LIPIcs*, pages 12:1–12:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.SWAT.2020.12.
- 3 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: <http://mitpress.mit.edu/books/introduction-algorithms>.
- 4 Giuseppe Di Battista and Roberto Tamassia. On-Line Graph Algorithms with SPQR-Trees. In Michael S. Paterson, editor, *Automata, Languages and Programming*, pages 598–611, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.
- 5 Yefim Dinitz, Alexander V. Karzanov, and Michael V. Lomonosov. On the structure of a family of minimum weighted cuts in a graph. *Studies in Discrete Optimization*, page 290–306, 1976.
- 6 Yefim Dinitz and Jeffery R. Westbrook. Maintaining the Classes of 4-Edge-Connectivity in a Graph On-Line. *Algorithmica*, 20(3):242–276, March 1998. doi:10.1007/PL00009195.
- 7 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997. doi:10.1145/265910.265914.
- 8 Tamás Fleiner and András Frank. A quick proof for the cactus representation of mincuts. Technical Report QP-2009-03, Egerváry Research Group, Budapest, 2009.
- 9 Greg N. Frederickson. Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications. *SIAM Journal on Computing*, 14(4):781–798, 1985. doi:10.1137/0214055.
- 10 Harold N. Gabow and Robert Endre Tarjan. A Linear-Time Algorithm for a Special Case of Disjoint Set Union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985. doi:10.1016/0022-0000(85)90014-5.
- 11 Zvi Galil and Giuseppe F. Italiano. Fully Dynamic Algorithms for Edge Connectivity Problems. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, STOC '91, page 317–327, New York, NY, USA, 1991. Association for Computing Machinery. doi:10.1145/103418.103454.
- 12 Zvi Galil and Giuseppe F. Italiano. Reducing edge connectivity to vertex connectivity. *SIGACT News*, 22(1):57–61, 1991. doi:10.1145/122413.122416.
- 13 Ralph E. Gomory and T. C. Hu. Multi-Terminal Network Flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961. URL: <http://www.jstor.org/stable/2098881>.

- 14 Dov Harel and Robert Endre Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.
- 15 Ramesh Hariharan, Telikepalli Kavitha, and Debmalaya Panigrahi. Efficient Algorithms for Computing All Low s-t Edge Connectivities and Related Problems. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, page 127–136, USA, 2007. Society for Industrial and Applied Mathematics.
- 16 Monika Henzinger, Satish Rao, and Di Wang. *Local Flow Partitioning for Faster Edge Connectivity*, pages 1919–1938. Society for Industrial and Applied Mathematics, 2017. doi:10.1137/1.9781611974782.125.
- 17 Monika Rauch Henzinger. Fully dynamic biconnectivity in graphs. *Algorithmica*, 13(6):503–538, June 1995. doi:10.1007/BF01189067.
- 18 Monika Rauch Henzinger and Valerie King. Randomized Dynamic Graph Algorithms with Polylogarithmic Time per Operation. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '95, page 519–527, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/225058.225269.
- 19 Monika Rauch Henzinger and Mikkel Thorup. Sampling to Provide or to Bound: With Applications to Fully Dynamic Graph Algorithms. *Random Struct. Algorithms*, 11(4):369–379, 1997.
- 20 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. doi:10.1145/502090.502095.
- 21 Jacob Holm and Eva Rotenberg. Worst-Case Polylog Incremental SPQR-Trees: Embeddings, Planarity, and Triconnectivity. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '20, page 2378–2397, USA, 2020. Society for Industrial and Applied Mathematics.
- 22 Jacob Holm, Eva Rotenberg, and Mikkel Thorup. Dynamic Bridge-Finding in $\tilde{O}(\log^2 n)$ Amortized Time. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '18, page 35–52, USA, 2018. Society for Industrial and Applied Mathematics.
- 23 John Hopcroft and Robert Tarjan. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Commun. ACM*, 16(6):372–378, 1973. doi:10.1145/362248.362272.
- 24 John Hopcroft and Robert Tarjan. Dividing a Graph into Triconnected Components. *SIAM Journal on Computing*, 2(3):135–158, 1973. doi:10.1137/0202012.
- 25 Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully Dynamic Connectivity in $O(\log n(\log \log n)^2)$ Amortized Expected Time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, page 510–520, USA, 2017. Society for Industrial and Applied Mathematics.
- 26 Arkady Kanevsky, Roberto Tamassia, Giuseppe Di Battista, and Jianer Chen. On-Line Maintenance of the Four-Connected Components of a Graph. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 793–801, 1991. doi:10.1109/SFCS.1991.185451.
- 27 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, page 1131–1142, USA, 2013. Society for Industrial and Applied Mathematics.
- 28 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, page 1131–1142, USA, 2013. Society for Industrial and Applied Mathematics. doi:10.1137/1.9781611973105.81.
- 29 David R. Karger. Minimum Cuts in Near-Linear Time. *J. ACM*, 47(1):46–76, 2000. doi:10.1145/331605.331608.

- 30 Kenichi Kawarabayashi and Mikkel Thorup. Deterministic Global Minimum Cut of a Simple Graph in Near-Linear Time. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '15, page 665–674, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2746539.2746588.
- 31 Johannes A. La Poutré and Jeffery Westbrook. Dynamic 2-Connectivity with Backtracking. *SIAM J. Comput.*, 28(1):10–26, 1999. doi:10.1137/S0097539794272582.
- 32 Johannes A. La Poutré, Jan van Leeuwen, and Mark H. Overmars. Maintenance of 2- and 3-edge-connected components of graphs I. *Discrete Mathematics*, 114(1):329–359, 1993. doi:10.1016/0012-365X(93)90376-5.
- 33 Wojciech Nadara, Mateusz Radecki, Marcin Smulewicz, and Marek Sokołowski. Determining 4-edge-connected components in linear time. *CoRR*, abs/2105.01699, 2021. arXiv:2105.01699.
- 34 Hiroshi Nagamochi and Toshihide Ibaraki. A linear time algorithm for computing 3-edge-connected components in a multigraph. *Japan Journal of Industrial and Applied Mathematics*, 9(2):163, June 1992. doi:10.1007/BF03167564.
- 35 Nima Norouzi and Yung H. Tsin. A simple 3-edge connected component algorithm revisited. *Information Processing Letters*, 114(1):50–55, 2014. doi:10.1016/j.ipl.2013.09.010.
- 36 Richard Peng, Bryce Sandlund, and Daniel Dominic Sleator. Optimal Offline Dynamic 2, 3-Edge/Vertex Connectivity. In Zachary Friggstad, Jörg-Rüdiger Sack, and Mohammad R. Salavatipour, editors, *Algorithms and Data Structures - 16th International Symposium, WADS 2019, Edmonton, AB, Canada, August 5-7, 2019, Proceedings*, volume 11646 of *Lecture Notes in Computer Science*, pages 553–565. Springer, 2019. doi:10.1007/978-3-030-24766-9_40.
- 37 Seth Pettie and Longhui Yin. The Structure of Minimum Vertex Cuts, 2021. arXiv:2102.06805.
- 38 Robert E. Tarjan and Jan van Leeuwen. Worst-Case Analysis of Set Union Algorithms. *J. ACM*, 31(2):245–281, March 1984. doi:10.1145/62.2160.
- 39 Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. doi:10.1137/0201010.
- 40 Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, STOC '00, page 343–350, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/335305.335345.
- 41 Yung H. Tsin. A Simple 3-Edge-Connected Component Algorithm. *Theory of Computing Systems*, 40(2):125–142, February 2007. doi:10.1007/s00224-005-1269-4.
- 42 Yung H. Tsin. Yet another optimal algorithm for 3-edge-connectivity. *Journal of Discrete Algorithms*, 7(1):130–146, 2009. Selected papers from the 1st International Workshop on Similarity Search and Applications (SISAP). doi:10.1016/j.jda.2008.04.003.
- 43 Jeffery Westbrook and Robert E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7(1):433–464, June 1992. doi:10.1007/BF01758773.
- 44 Christian Wulff-Nilsen. Faster Deterministic Fully-Dynamic Graph Connectivity. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, page 1757–1769, USA, 2013. Society for Industrial and Applied Mathematics.

Determining 4-edge-connected components in linear time*

Wojciech Nadara[†]

Mateusz Radecki[‡]

Marcin Smulewicz[§]

Marek Sokołowski[¶]

September 21, 2025

Abstract

In this work, we present the first linear time deterministic algorithm computing the 4-edge-connected components of an undirected graph. First, we show an algorithm listing all 3-edge-cuts in a given 3-edge-connected graph, and then we use the output of this algorithm in order to determine the 4-edge-connected components of the graph.



European Research Council
Established by the European Commission



*This research is a part of projects that have received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant Agreement 714704 (W. Nadara), 677651 (M. Smulewicz), and 948057 (M. Sokołowski)).

[†]Institute of Informatics, University of Warsaw, Poland (w.nadara@mimuw.edu.pl)

[‡]University of Warsaw, Poland (mr386052@students.mimuw.edu.pl)

[§]Institute of Informatics, University of Warsaw, Poland (m.smulewicz@mimuw.edu.pl)

[¶]Institute of Informatics, University of Warsaw, Poland (marek.sokolowski@mimuw.edu.pl)

1 Introduction

The connectivity of graphs has always been one of the fundamental concepts of graph theory. The foremost connectivity notions in the world of undirected graphs are the *k-edge-connectedness* and the *k-vertex-connectedness*. Namely, a graph G is *k-edge-connected* for $k \geq 1$ if it is connected, and it remains connected after removing any set of at most $k - 1$ edges. Similarly, G is *k-vertex-connected* if it contains at least $k + 1$ vertices, and it remains connected after the removal of any set of at most $k - 1$ vertices.

These notions can be generalized to the graphs that are not well-connected. Namely, if H is a maximal *k-vertex-connected* subgraph of G , we say that H is a *k-vertex-connected component* of G . The edge-connected variant is, however, defined differently: we say that a pair of vertices u, v of G is *k-edge-connected* if it is not possible to remove at most $k - 1$ edges from G so that u and v end up in different connected components. This relation of *k-edge-connectedness* happens to be an equivalence relation; this yields a definition of a *k-edge-connected component* of G as an equivalence class of the relation. We remark that the notions of *k-vertex-connected components* and *k-edge-connected components* coincide for $k = 1$ as both simply describe the connected components of G . However, for $k \geq 2$ these definitions diverge; in particular, for $k \geq 3$ the *k-edge-connected components* of a graph do not even need to be connected.

There has been a plethora of research into the algorithms deciding the *k-vertex-* and *k-edge-connectedness* of graphs, and decomposing the graphs into *k-vertex-* or *k-edge-connected components*. However, while classical, elementary, and efficient algorithms exist for $k = 1$ and $k = 2$, these problems become increasingly more difficult for the larger values of k . In fact, even for $k = 4$, there were no known linear time algorithms to any of the considered problems. The following description presents the previous work in this area for $k \in \{1, 2, 3, 4\}$, and exhibits the related work for the larger values of k :

k = 1. Here, the notions of *k-vertex-connectedness* and *k-edge-connectedness* reduce to that of connectivity and the connected components of a graph. In the static setting, determining the connected components in linear time is trivial. As a consequence, more focus is being laid on dynamic algorithms maintaining the connected components of graphs. In the incremental setting, where the edges can only be added to the dynamic graph, the optimal solution is provided by disjoint-set data structures [37], which solve the problem in the amortized $\mathcal{O}(\alpha(n))$ time per query, where $\alpha(n)$ denotes the inverse of the Ackermann’s fast-growing function. The fully dynamic data structures are also considered [7, 9, 18, 19, 20, 22, 25, 27, 38, 42].

k = 2. One step further are the notions of 2-vertex-connectivity (biconnectivity) and 2-edge-connectivity. In the static setting, partitioning of a graph into 2-vertex-connected or 2-edge-connected components are classical problems, both solved in linear time by exploiting the properties of the low function [23]. The incremental versions of both problems are again solved optimally in the amortized $\mathcal{O}(\alpha(n))$ time per query [41]. Significant research has been done in the dynamic setting as well [7, 11, 17, 20, 22, 30, 34].

k = 3. As a next step, we consider 3-vertex-connectivity (triconnectivity) and 3-edge-connectivity. An optimal, linear time algorithm detecting the 3-vertex-connected components was first given by Hopcroft and Tarjan [24]. The first linear time algorithm for 3-edge-connectivity was discovered much later by Galil and Italiano [12], where they present a linear time reduction from

the k -edge-connectivity problem to k -vertex-connectivity for $k \geq 3$, showing that in the static setting, the former problem is the easier of the two. This was later followed by a series of works simplifying the solution for 3-edge-connectivity [32, 33, 39, 40]. The incremental setting [21, 31] and the dynamic setting [7, 11] were also considered.

We also mention that in the case of 3-vertex-connectivity, there exists a structure called SPQR-tree which succinctly captures the structure of 2-vertex-cuts in graphs [3, 21]. Its edge-connectivity analogue also exists, but we defer its introduction to the general setting.

k = 4. We move on to the problems of 4-vertex-connectivity and 4-edge-connectivity. A notable result by Kanevsky et al. [26] supports maintaining 4-vertex-connected components in incremental graphs, with an optimal $\mathcal{O}(\alpha(n))$ amortized time per query. Their result also yields the solution for static graphs in $\mathcal{O}(m + n\alpha(n))$ time complexity. By applying the result of Galil and Italiano [12], we derive a static algorithm determining the 4-edge-connected components in the same time complexity. This algorithm is optimal for $m = \Omega(n\alpha(n))$.

Another result by Dinitz and Westbrook [6] supports maintaining the 4-edge-connected components in the incremental setting. Their algorithm processes any sequence of queries in $\mathcal{O}(q + m + n \log n)$ time where q is the number of queries, and m is the total number of inserted edges to the graph.

However, it is striking that the fastest solutions for 4-edge-connectivity and 4-vertex-connectivity for static graphs were derived from the on-line algorithms working in the incremental setting. In particular, no linear time algorithms for $k = 4$ were known before.

k ≥ 5. As a side note, we also present the current knowledge on the general problems of k -vertex-connectivity and k -edge-connectivity. A series of results [1, 16, 28, 29] show that it is possible to compute the minimum edge cut of a graph (i.e., determine the edge-connectivity of a graph) in near-linear time. The previously mentioned work by Dinitz and Westbrook [6] maintains the k -edge-connected components of an incremental graph which is assumed to already have been initialized with a $(k - 1)$ -edge-connected graph. The data structure answers any sequence of on-line queries in $\mathcal{O}(q + m + k^2 n \log(n/k))$ time, where q is the number of queries, and m is the number of edges in the initial graph.

Gomory and Hu [13] proved that for any weighted, undirected graph G there exists a weighted, undirected tree T on the same vertex set such that for any two vertices $s, t \in V(G)$, the value of the minimum s - t edge cut in T is equal to the value of the minimum s - t edge cut in G . Moreover, such a tree can be constructed using $n - 1$ invocations of the maximum flow algorithm. In an interesting result by Hariharan et al. [15], the decomposition of any graph into k -edge-connected components is constructed in $\mathcal{O}((m + nk^3) \cdot \text{polylog}(n))$ time, producing a partial Gomory-Hu tree as its result.

Dinitz et al. [5] showed that the set of all minimum edge cuts can be succinctly represented with a *cactus graph*. When the minimum edge cut is odd, this cactus simplifies to a tree (see [8, Corollary 8]). These results imply that if the size of the minimum cut is odd, then the number of minimum cuts is $\mathcal{O}(n)$ and if it is even, then the number of minimum cuts is $\mathcal{O}(n^2)$. The structure of k -vertex-cuts was also investigated [35].

1.1 Our results

In this work, we present a linear time, deterministic algorithm partitioning static, undirected graphs into 4-edge-connected components. Even though the area of the dynamic versions of

the algorithms for k -edge-connectivity is still thriving, the progress in static variants appears to have plateaued. In particular, both subquadratic algorithms determining the 4-edge-connected components [6, 26] originate from their dynamic incremental equivalents and are almost thirty years old, yet they did not achieve the optimal linear running time. Hence, our work constitutes the first progress in the static setting of 4-edge-connectivity in a long time. As a side result, our algorithm also produces the tree representation of 3-edge-cuts as explained in [8].

1.2 Organization of the work

The paper is organized as follows. In Section 3, we show how to reduce the problem of determining 4-edge-connected components to the problem of determining 4-edge-connected components in 3-edge-connected graphs. In Section 4, we show a linear time, randomized Monte Carlo algorithm for listing all 3-edge-cuts in 3-edge-connected graphs. In Section 5, we show how to remove the dependency on the randomness in the algorithm from the previous section, producing a linear time, deterministic algorithm listing all 3-edge-cuts in 3-edge-connected graphs. Then, in Section 6, we construct a tree of 3-edge-cuts in a 3-connected graph, given the list of all its 3-edge-cuts. This tree is then used to determine the 4-edge-connected components of the graph. Finally, in Section 7, we present open problems related to this work.

2 Preliminaries

Graphs. In this work, we consider undirected, connected graphs which may contain self-loops and multiple edges connecting pairs of vertices (i.e., multigraphs). The number of vertices of a graph and the number of its edges are usually denoted n and m , respectively.

We use the notions of k -edge-connectedness and k -edge-connected components defined in Section 1. Moreover, we say that a set of k edges of a graph forms a k -edge-cut (or a k -cut for simplicity) if the removal of these edges from the graph disconnects it.

DFS trees. Consider a run of the *depth-first search* algorithm [36] on a connected graph G . A *depth-first search tree* (or a *DFS tree*) is a spanning tree $\mathcal{T}$ of G , rooted at the source of the search r , containing all the edges traversed by the algorithm. After the search is performed, each vertex v is assigned two values: its *preorder* $\text{pre}(v)$ (also called *discovery time* or *arrival time*) and *postorder* $\text{post}(v)$ (also *finishing time* or *departure time*). Their definitions are standard [2]; it can be assumed that the values range from 1 to $2n$ and are pairwise different.

The edges of $\mathcal{T}$ are called *tree edges*, and the remaining edges are called *back edges* or *non-tree edges*. In this setup, every back edge e connects two vertices remaining in ancestor-descendant relationship in $\mathcal{T}$; moreover, the graph $\mathcal{T} + e$ contains exactly one cycle, named the *fundamental cycle* of e . For a vertex v of G , we define $\mathcal{T}_v$ to be the subtree of $\mathcal{T}$ rooted at v ; similarly, for a tree edge e whose deeper endpoint is v , we set $\mathcal{T}_e = \mathcal{T}_v$.

When a DFS tree $\mathcal{T}$ of G is fixed, it is common to introduce directions to the edges of the graph: all tree edges of $\mathcal{T}$ are directed away from the root of $\mathcal{T}$, and all back edges are pointed towards the root of $\mathcal{T}$. Then, uv is a directed edge (either a tree or a back edge) whose *origin* (or *tail*) is u , and whose *destination* (or *head*) is v .

For our convenience, we introduce the following definition: a back edge $e = pq$ *leaps over* a vertex v of the graph if $p \in \mathcal{T}_v$, but $q \notin \mathcal{T}_v$; we analogously define *leaping over* a tree edge f .

Moreover, we define a partial order $\leq_{\mathcal{T}}$ on the vertices of G and the tree edges of $\mathcal{T}$ as follows: $x \leq_{\mathcal{T}} y$ if the simple path in $\mathcal{T}$ connecting the root of $\mathcal{T}$ with y also contains x . Then, $\leq_{\mathcal{T}}$ has one minimal element—the root of $\mathcal{T}$ —and each maximal element is a leaf of $\mathcal{T}$. When the tree $\mathcal{T}$ is clear from the context, we may write $\leq$ instead of $\leq_{\mathcal{T}}$. We may also use $x < y$ for $x \leq y \wedge x \neq y$. Using the precomputed preorder and postorder values in $\mathcal{T}$, we can verify if $x \leq_{\mathcal{T}} y$ holds for given $x, y \in V(G) \cup E(\mathcal{T})$ in constant time.

We use the classical low function defined by Hopcroft and Tarjan [23]. However, for our purposes it is more convenient to define it as a function $\text{low} : E(\mathcal{T}) \rightarrow (E \setminus E(\mathcal{T})) \cup \{\perp\}$ such that for a tree edge e , $\text{low}(e)$ is the back edge uv leaping over e minimizing the preorder of its head v , breaking ties arbitrarily; or $\perp$, if no such edge exists. This function can be computed for all tree edges in time linear with respect to the size of the graph.

Xors. For sets A and B , by $A \oplus B$ we denote their symmetric difference, which is $(A \cup B) \setminus (A \cap B)$, and we call it a *xor* of A and B . Moreover, for non-negative integers a and b , by $a \oplus b$ we denote their xor, that is, an integer whose binary representation is a bitwise symmetric difference of the binary representations of a and b . The definitions can be easily generalized to the symmetric differences of multiple sets or integers.

3 Reduction to the 3-edge-connected case

In this section, we present a way to reduce the problem of building the structure of 4-connected components to a set of independent, simpler instances of the problem. Each produced instance will be a 3-edge-connected graph corresponding to a single 3-edge-connected component of the original graph. This transformation of the input will be vital to the correctness of our work since the algorithm described in the following sections assumes the 3-edge-connectedness of the input graph. We remark that this is not a new contribution [6]; we present it here for completeness only.

Firstly, each connected component of G can be considered independently. Similarly, bridges (i.e., 1-edge-cuts) split the given graph into independent 2-edge-connected components. Moreover, it can be shown that for $k \geq 2$, the family of k -edge-connected components of G will not be altered by the removal of the bridges. Thus, without loss of generality, we assume that G is 2-edge-connected.

For a 2-edge-connected graph G , we first build a structure of its 3-edge-connected components. The shape of this structure is a *cactus graph*, asserted by the following theorem:

Theorem 3.1 ([4]). *For a given 2-edge-connected graph G , there exists an auxiliary graph $H = (U, F)$ such that*

- *each edge $e \in F$ lies on exactly one simple cycle of H ,*
- *U is the family of all 3-edge-connected components of G ,*
- *there exists a bijection $\phi : E' \rightarrow F$ where $E' \subseteq E$ is the set of all edges belonging to some 2-edge-cut of G , such that for every edge $e = uv \in E'$, its image $\phi(e)$ is an edge of H connecting the 3-edge-connected component containing u with the 3-edge-connected component containing v ,*

- a pair of edges $e, f \in E'$ forms a 2-edge-cut if and only if $\phi(e)$ and $\phi(f)$ belong to the same cycle of H .

In order to build the structure from Theorem 3.1, we use the result of Galil and Italiano [12] to find all 3-edge-connected components of G . Then, H is defined as a quotient graph created by identifying the vertices within each 3-edge-connected component. This can easily be performed in linear time with respect to the size of G .

By definition, the partition of the vertices of G into 4-edge-connected components is a refinement of the partition into 3-edge-connected components. However, we can no longer restrict our attention to the independent subgraphs induced on 3-edge-connected components (which is what we did in the case of connected components and 2-edge-connected components). In fact, the 3-edge-connected components of G may even be disconnected. In order to handle this problem, we will need to transform G in order to turn each of its 3-edge-connected components into a separate connected component which itself is 3-edge-connected.

Let us now fix $X \subseteq V$ —a 3-edge-connected component of G . We will now construct a 3-edge-connected graph $S = (X, Y)$ whose partition into 4-edge-connected components will be the same as the partition of X into 4-edge-connected components in the original graph. Initially, we assign to Y the set of all edges of $G[X]$. Then, for each cycle C of H incident to X ($X \in V(H)$), we add an additional special edge to Y . Formally, let $\phi(e_1), \phi(e_2) \in F$ be the two edges of the cycle C in H that are incident to X . Then, both edges $e_1, e_2 \in E$ have exactly one endpoint belonging to X ; denote them a and b , respectively. Then, we add ab to Y as the additional special edge for the cycle C . Intuitively, the new edge simulates a path in G connecting a and b which is internally disjoint with X and which goes through the 3-edge-connected components around the cycle C .

It now turns out that S is 3-edge-connected and captures the connectivity properties of X in G :

Lemma 3.2 ([4, 6]). *The graph S defined above has the following properties:*

- it is 3-edge-connected,
- for each 3-edge-cut c of G dividing X into nonempty parts, there is a 3-edge-cut c' of S dividing X in the same way,
- for each 3-edge-cut c' of S , there is a nonempty set of 3-edge-cuts of G , each dividing X in the same way as c' .

By Lemma 3.2, each 4-edge-connected component $Y \subseteq X$ of S is also a 4-edge-connected component of G . Therefore, the reduction is sound.

Given the structure of 3-edge-connected components of G , it is easy to construct graphs S for each particular X in total $\mathcal{O}(n + m)$ time, hence the only remaining part is to construct the structure of 4-edge-connected components for each S independently.

Observe that the total number of special edges added to all 3-edge-connected components is equal to the total length of the cycles in H . Thus, the reduction can easily be performed in time linear with respect to the size of G . As a result, without loss of generality, we can assume that the given graph G is 3-edge-connected.

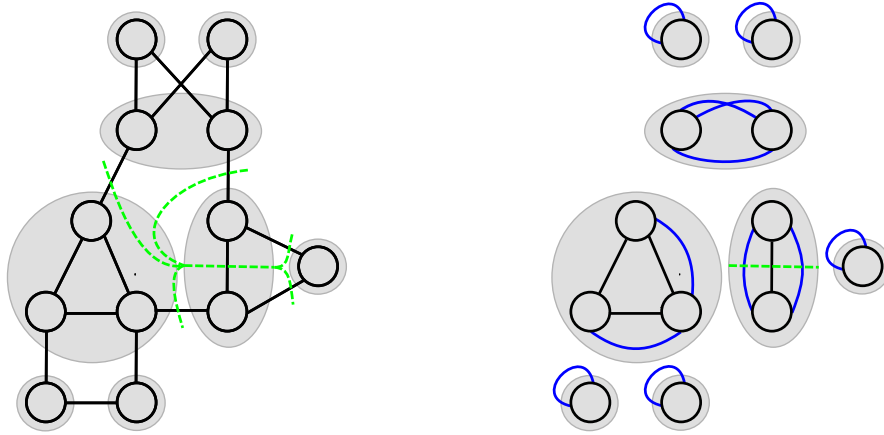


Figure 1: A 2-edge-connected graph with marked 3-edge-connected components (left) and corresponding graphs S for each X with blue special edges (right). Green dashed edges show an example of all 3-edge-cuts (left) corresponding to one 3-edge-cut (right).

4 Simple randomized algorithm

In this section, we are going to describe a randomized linear time algorithm listing 3-edge-cuts in 3-edge-connected graphs. In particular, the existence of this algorithm will imply that the number of 3-edge-cuts in any 3-edge-connected graph is at most linear. It has no major advantages over the algorithm presented in the succeeding section, but it is significantly simpler and it already contains most of the core ideas. Thus, it serves as a good intermediate step in the explanation.

The overview of the algorithm is as follows. First, we will construct a randomized oracle (Lemma 4.5) that given two edges e, f of the 3-edge-connected graph G , verifies in constant time whether there exists another edge g such that $\{e, f, g\}$ is a 3-edge-cut of G ; and if such an edge exists, it is returned by the oracle. It will be apparent from the description of the oracle that such an edge—if it exists—is unique. Thanks to Lemma 4.5, the description of our randomized algorithm will be simplified significantly: now, we only need to identify two out of three edges of each 3-edge-cut of G . Then, the remaining edge of each cut can be easily recovered using a single oracle call.

Next, our algorithm will consider an arbitrary DFS tree $\mathcal{T}$ of G , categorize all 3-edge-cuts by the number $t \in \{0, 1, 2, 3\}$ of tree edges of $\mathcal{T}$ in the cut, and find all 3-edge-cuts separately for each value of t . The case $t = 0$ is trivial: $\mathcal{T}$ is a spanning tree of G , hence each edge cut must contain at least one edge of $\mathcal{T}$. The cases $t = 1$ and $t = 2$ will proceed by considering all tree edges of $\mathcal{T}$ separately. For each such tree edge e , we will find all 3-edge cuts containing both e and some non-tree edge leaping over e . By performing a case study, we will show that for each e , there is only a constant number of 3-edge-cuts of this form for $t \in \{1, 2\}$; and moreover, each of them can be located by us in constant time after a linear time preprocessing of $\mathcal{T}$. Finally, the case $t = 3$ will be solved by a recursive call to the algorithm on a properly contracted graph.

We begin with the description of some auxiliary data structures.

Theorem 4.1 ([10]). *There exists a data structure for the disjoint set union problem which, when initialized with an undirected tree $\mathcal{T}$ (the “union tree”) on n vertices, creates n singleton*

sets. After the initialization, the data structure accepts the following queries in any order:

- **find**(x): returns the index of the set containing x ,
- **union**(x, y): if x and y are in different sets, then an arbitrary one of them is replaced with their union and the other one with the empty set. This query can only be issued if xy is an edge of $\mathcal{T}$.

The data structure executes any sequence of q queries in total $\mathcal{O}(n + q)$ time.

Lemma 4.2. *It is possible to enrich the data structure from Theorem 4.1, so that after rooting it at an arbitrary vertex, we are able to answer the following query in constant time:*

- **lowest**(x): returns the smallest vertex of S with respect to $\leq_{\mathcal{T}}$, where S is the set containing x .

Note that each set induces a connected subgraph of $\mathcal{T}$, hence the smallest vertex of S is well-defined.

Proof. We enrich our structure with information about the lowest common ancestor of all elements of such a set. Call this $\text{LCA}(S)$ for a set S . Whenever we merge two sets S and R , it holds that one of $\text{LCA}(S)$ and $\text{LCA}(R)$ is the ancestor of the other. Moreover, this ancestor can be determined in constant time. Hence, $\text{LCA}(S \cup R)$ can be determined in constant time. $\square$

Theorem 4.3. *There exists a deterministic algorithm that takes as input:*

- an undirected, unrooted tree $\mathcal{T}$ with n vertices,
- p weighted paths $P_1, P_2, \dots, P_p$ in the tree, where the path P_i ($1 \leq i \leq p$) is the unique path between vertices u_i, v_i and has weight $w_i \in \{0, 1, \dots, C\}$,
- and a positive integer k ,

and for each edge e of the tree returns the indices of k paths with the lowest weight containing e , breaking ties arbitrarily; if e is a part of fewer than k paths, all such paths are returned. The time complexity of the algorithm is $\mathcal{O}(nk + p + C)$.

Proof. Since weights w_i belong to the set $\{0, 1, \dots, C\}$ we can sort paths $P_1, \dots, P_p$ with respect to their weights in time $\mathcal{O}(p + C)$. Hence, from this point on, we assume that it has already been done and that $w_1 \leq w_2 \leq \dots \leq w_p$.

We now process these paths in that order. We create the data structure from Lemma 4.2 and initialize it with $\mathcal{T}$. For each edge $e \in E(\mathcal{T})$, we are going to maintain a set R_e , which at the end will be the set of the indices of the desired paths for edge e .

We root $\mathcal{T}$ in an arbitrary vertex, which allows us to define the function $\text{parent}(v)$ mapping v to its parent in $\mathcal{T}$.

We define an auxiliary function $\text{Go}(u, v, i)$ with the following pseudocode:

Algorithm 1. Updating information with the path P_i on its part from u_i to $\text{LCA}(u_i, v_i)$

```

function GO( $u, v, i$ )
   $u \leftarrow \text{lowest}(u)$ 
  while  $u$  is not an ancestor of  $v$  do
     $e \leftarrow (u, \text{parent}(u))$ 
     $R_e \leftarrow R_e \cup \{i\}$ 
    if  $|R_e| = k$  then  $\text{union}(u, \text{parent}(u))$ 
     $u \leftarrow \text{lowest}(\text{parent}(u))$ 

```

Whenever we process path P_i , we execute $\text{GO}(u_i, v_i, i)$ and $\text{GO}(v_i, u_i, i)$.

We maintain an invariant that after processing the paths $P_1, \dots, P_i$, sets R_e are the sets of indices of k first paths containing e ; or all of them, if there are less than k paths among $P_1, \dots, P_i$ containing e as an edge. Moreover, $\text{union}(u, v)$ is executed as soon as the size of R_{uv} becomes equal to k . The reader is encouraged to think of the **union** as the contraction of the edge uv in $\mathcal{T}$. A contracted subgraph is identified with the lowest common ancestor of all its vertices (i.e., the result of $\text{lowest}(x)$ for any x from such a subgraph). The function $\text{GO}(u, v, i)$ traverses all non-contracted edges on the path from u to l , where l is the lowest common ancestor of u and v , and adds the index i to the sets R_e for all traversed edges e (and contracts them if necessary). The total size of all sets R_e will never exceed nk , hence the total number of iterations of the WHILE loop throughout all executions of GO will not exceed nk as well. We infer that the time complexity of this algorithm is $\mathcal{O}(nk + p + C)$. $\square$

We proceed to the description of our randomized algorithm. Let us choose an arbitrary vertex r of the graph, and perform a depth-first search from r . Let $\mathcal{T}$ be the resulting DFS tree. We are now going to define a *hashing function* $H : E \rightarrow \mathcal{P}(E)$, where $\mathcal{P}(E)$ denotes the powerset of E ; the value $H(e)$ will be called a *hash* of e . If $e \notin \mathcal{T}$, then we define $H(e) = \{e\}$. Otherwise, we take $H(e)$ as the set of non-tree edges leaping over e . Let us note that the set C_e of edges f such that $e \in H(f)$ forms a cycle—the fundamental cycle of e .

Lemma 4.4. *For a connected graph $G = (V, E)$ and a subset A of its edges, the graph $G' := G - A$ is disconnected if and only if there is a nonempty subset $B \subseteq A$ such that xor of the hashes of the edges in B is an empty set.*

Proof. Let us start with proving that if $G' = (V, E \setminus A)$ is disconnected, then there is a nonempty subset $B \subseteq A$ such that xor of hashes of edges from B is the empty set.

If G' is disconnected, then we can partition V into two nonempty sets L and R such that all edges between L and R belong to A . Let B be the set of the edges between L and R . We claim that xor of the hashes of the edges from B is the empty set. Denote that xor by X . Let us take any edge e . Since C_e is a cycle, it contains an even number of edges from B . Hence $e \notin X$, which proves that $X = \emptyset$.

Now, let us prove that if there exists a subset $B \subseteq A$ such that xor of the hashes of its edges is the empty set, then G' is disconnected. Let us color vertices of G red and blue such that two vertices connected by a tree edge of $\mathcal{T}$ have different colors if and only if this tree edge belongs to B . Since $\mathcal{T}$ is a tree, this coloring always exists, and is unique up to swapping the colors. B has to contain at least one tree edge; otherwise xor of the hashes of its edges would clearly be nonempty. Hence, in such a coloring, there are vertices of both colors. Let L and R denote the nonempty sets of vertices colored blue and red, respectively. We claim that

there is no edge in $E \setminus B$ connecting L and R , which in turn will conclude the proof. For the sake of contradiction, assume that such an edge $e \in E \setminus B$ exists. It clearly cannot be a tree edge based on how we defined our coloring. If e is a non-tree edge connecting the vertices of different colors, it has to leap over an odd number of tree edges in B . These are exactly the edges of $B \cap E(\mathcal{T})$ that contain e in their hashes. Since xor of the hashes of the edges from B is assumed to be the empty set, e has to be contained in B as well, because e is the only non-tree edge that contains e in its hash—a contradiction. $\square$

Since our graph has no 1-edge-cuts, there are no edges whose hashes are empty sets; and since our graph has no 2-edge-cuts, no two edges have equal hashes. Hence, removing a set of three edges disconnects a 3-edge-connected graph if and only if xor of the hashes of all of them is the empty set. Moreover, after removing some 3-edge-cut, the graph disconnects into exactly two components, and no removed edge connects vertices within one component.

As storing hashes as sets of edges would lead to inefficient computations, we will define *compressed hashes*. We express them as b -bit numbers, where $b = \lceil 3 \log_2(m) \rceil$, i.e., as a function $CH : E \rightarrow \{0, \dots, 2^b - 1\}$. For each non-tree edge e , we draw $CH(e)$ randomly and uniformly from the set of b -bit numbers. For a tree edge e , we define its compressed hash $CH(e)$ as the xor of the compressed hashes of the edges in its hash, i.e. $CH(e) = \bigoplus_{f \in H(e)} CH(f)$. Note that since $b = \mathcal{O}(\log m)$, we can perform arithmetic operations on compressed hashes in constant time. However, this comes at a cost of allowing the collisions of compressed hashes: it might happen (although with a very low probability) that two sets of edges have equal xors of their compressed hashes, but unequal xors of their original hashes.

For the ease of exposition, in the description of this algorithm we will use hash tables, which only guarantee the expected average constant time complexity per lookup. Because of that, the described algorithm will have expected linear instead of worst-case linear running time. However, there is a way to remove the dependence on hash tables and keep the running time worst-case linear. This will be explained at the end of this section (Subsection 4.7). For the time being, we create a hash table $M : \{0, \dots, 2^b - 1\} \rightarrow E \cup \{\perp\}$, which for any b -bit number x returns an edge whose compressed hash is equal to x ; or $\perp$ if no such edge exists. In the unlikely event that there are multiple edges whose compressed hashes are equal to x , M returns any of them.

We will now categorize 3-edge-cuts based on the number of tree edges they contain, and explain how to handle each case. Throughout the case analysis, we will use the following fact multiple times:

Lemma 4.5. *Given two edges e and f of some 3-edge-cut in a 3-edge-connected graph, the remaining edge is uniquely identified by its hash: $H(e) \oplus H(f)$.*

Proof. If $\{e, f, g\}$ is a 3-edge-cut, then $H(e) \oplus H(f) \oplus H(g) = \emptyset$, so $H(g) = H(e) \oplus H(f)$. $\square$

4.1 Zero tree edges

As $\mathcal{T}$ is a spanning connected subgraph, 3-edge-cuts not containing any tree edges simply do not exist.

4.2 One tree edge

Let e be an edge of $\mathcal{T}$ that is the only tree edge of some 3-edge-cut. The two resulting connected components after the removal of such a cut are $\mathcal{T}_e$ and $\mathcal{T} \setminus \mathcal{T}_e$ (we are slightly abusing the

notation here; we actually mean that the connected components are the subgraphs induced by the vertex sets of $\mathcal{T}_e$ and $\mathcal{T} \setminus \mathcal{T}_e$, respectively). Since e is not a bridge, $\text{low}(e)$ is well-defined and it connects $\mathcal{T}_e$ with $\mathcal{T} \setminus \mathcal{T}_e$, so it belongs to this cut as well. This uniquely determines the third edge of this cut as it has to be the unique edge whose hash is $H(e) \oplus H(\text{low}(e))$.

In order to detect all such cuts, we iterate over all tree edges e and for each of them we look up in M whether there exists an edge whose compressed hash is $CH(e) \oplus CH(\text{low}(e))$. If it exists and if it turns out to be a non-tree edge, then we call it g and output the triple $\{e, \text{low}(e), g\}$ as a 3-edge-cut.

4.3 Two tree edges

Let e and f be the edges of $\mathcal{T}$ that form a 3-edge-cut together with some back edge g .

Lemma 4.6. *The edges e and f are comparable with respect to $\leq_{\mathcal{T}}$.*

Proof. Assume otherwise. Then $\mathcal{T}_e$ and $\mathcal{T}_f$ are disjoint, and $L := \mathcal{T}_e \cup \mathcal{T}_f$ and $R := \mathcal{T} \setminus L$ are two connected components resulting from the removal of the 3-edge-cut containing e and f . However, $\text{low}(e)$ and $\text{low}(f)$ are well-defined different edges connecting L and R , so any cut between L and R would need to contain $e, f, \text{low}(e)$ and $\text{low}(f)$ at the same time—a contradiction. This proves that e and f are comparable. $\square$

Without loss of generality, assume that $e <_{\mathcal{T}} f$. Then, the two resulting connected components are $L := \mathcal{T}_e \setminus \mathcal{T}_f$ and $R := \mathcal{T} \setminus L$. The remaining back edge g has to connect L and R . We will distinguish two cases, differing in the location of the remaining back edge: *two tree edges, lower case*, where g connects L with $\mathcal{T}_f$, and *two tree edges, upper case*, where g connects L with $\mathcal{T} \setminus \mathcal{T}_e$.

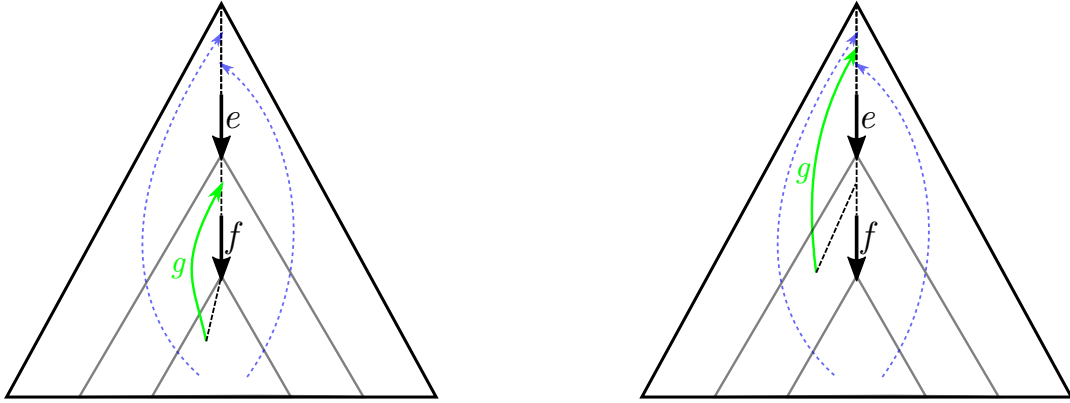


Figure 2: The settings in 4.3.1 and 4.3.2. Cuts that we are looking for are formed by edges $\{e, f, g\}$ in both cases.

4.3.1 Two tree edges, lower case

Let A be the set of back edges between $\mathcal{T}_f$ and $\mathcal{T} \setminus \mathcal{T}_f$. It consists of the edge g , which connects L with $\mathcal{T}_f$, and of several edges connecting $\mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_e$. Let B be the set of heads of edges from A (we remind that back edges are directed towards the root r). All elements of B lie on the path from f to the root r , and the head of g is the deepest element of B .

Based on this observation, we are going to use the data structure from Theorem 4.3. We initialize an instance of it with the tree $\mathcal{T}$, $k = 1$, $C = 2n$ and paths $\{P_e \mid e \text{ is a back edge}\}$, where for each non-tree edge $e = xy$, where y is its head, we create one input path P_e from x to y , and we set $w_e := 2n - \text{pre}(y)$. This data structure lets us for each tree edge f determine the back edge leaping over f with the biggest value of preorder of its head. We call this edge $\text{MaxUp}(f)$. This back edge is the only candidate for the edge g for a fixed edge f .

Hence, we can find all such cuts by firstly initializing the data structure, and then iterating over all tree edges f . For each f , we take $g = \text{MaxUp}(f)$. Knowing f and g , we can look up in M whether there exists an edge whose compressed hash is $CH(f) \oplus CH(g)$ (Lemma 4.5). If it exists and if it is a tree edge, then we call it e and output the triple $\{e, f, g\}$ as a 3-edge-cut.

4.3.2 Two tree edges, upper case

Let A be the set of non-tree edges between $\mathcal{T}_e$ and $\mathcal{T} \setminus \mathcal{T}_e$. It consists of the back edge g , which connects L with $\mathcal{T} \setminus \mathcal{T}_e$, and of several edges connecting $\mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_e$. Let B be the set of the tails of the edges from A . Let $v \in L$ be the tail of g . As $v \notin \mathcal{T}_f$, either for all vertices $u \in \mathcal{T}_f$ it holds that $\text{pre}(v) < \text{pre}(u)$, or for all vertices $u \in \mathcal{T}_f$ it holds that $\text{pre}(u) < \text{pre}(v)$. In the first case, v is the vertex with the smallest preorder which is a tail of some edge leaping over e ; while in the second case v is the analogous vertex with largest preorder.

Based on this observation, we are going to use the data structure from Theorem 4.3 again. We initialize one instance of it with the tree $\mathcal{T}$, $k = 1$, $C = 2n$ and the paths $\{P_e \mid e \text{ is a back edge}\}$, where for each back edge $e = xy$, we create one input path P_e from x to y , and set $w_e := \text{pre}(x)$. We also initialize another instance of this data structure in the same way, with the only difference that we set $w_e := 2n - \text{pre}(x)$ instead. The first instance lets us for each tree edge e determine the edge $\text{MinDn}(e)$ leaping over e with the smallest preorder of its tail; while the second instance determines the analogous edge $\text{MaxDn}(e)$, only with the largest preorder. By our considerations above, if any desired $\{e, f, g\}$ cut exists, then $g \in \{\text{MinDn}(e), \text{MaxDn}(e)\}$.

Hence, we can find all such cuts by firstly initializing both instances of the data structure. Then, we iterate over all tree edges e , and for each e we check both candidates for g . If our hash table contains an edge whose compressed hash is $CH(e) \oplus CH(g)$ and it is a tree edge, then we call it f and output the triple $\{e, f, g\}$ as a 3-edge-cut.

4.4 Three tree edges

Solving this case in a way similar to the previous cases seems intractable. We consider this subsection, together with the time analysis following it, as one of the key ideas of this work.

In this case, we assume that no non-tree edges belong to the cut. Therefore, we can contract all of them simultaneously and recursively list all 3-edge-cuts in the resulting graph. Since 3-edge-cuts in the contracted graph exactly correspond to the 3-edge-cuts consisting solely of tree edges in the original graph, this reduction is sound. The contraction is performed in $\mathcal{O}(n + m)$ time by identifying the vertices within the connected components of $G - E(\mathcal{T})$. Let G' be the graph after these contractions. Since G is 3-edge-connected, G' has to be 3-edge connected as well, so the assumption about the input graph being 3-edge-connected is preserved. We do not modify the value of b in the subsequent recursive calls, even though the value of m will decrease.

4.5 Time analysis

We will now compute the expected time complexity of our algorithm. The subroutines detecting the cuts with at most two tree edges clearly take expected linear time. The only nontrivial part of the time analysis is the recursion in Subsection 4.4. Let $T(m)$ denote the maximum expected time our algorithm needs to solve any graph with at most m edges. Since our graph is 3-edge-connected, the degree of each vertex in G is at least three, so $|E(G)| \geq \frac{3}{2}|V(G)| \geq \frac{3}{2}|E(G')| \Rightarrow |E(G')| \leq \frac{2}{3}|E(G)|$. The bound on $T(m)$ follows:

$$T(m) \leq \mathcal{O}(m) + T\left(\frac{2}{3}m\right).$$

The solution to this recurrence is $T(m) = \mathcal{O}(m)$, hence the whole algorithm runs in expected linear time.

4.6 Correctness analysis

In this subsection, we are going to prove that our algorithm works with sufficiently high probability. The only reason it can output wrong result comes from the compression of hashes—if we used their uncompressed version instead, the algorithm would clearly be correct.

The maximum number of queries to our hash table is linear in terms of n , which follows from a similar argument to the one presented in Subsection 4.5. Hence, there exists some absolute constant c such that the number of queries is bounded by cn . For each query with value q , and for each edge whose hash is not equal to the hash whose compressed version we ask about, there is 2^{-b} probability that compressed version of this hash is equal to q . Hence, the probability that we ever get a false positive is bounded from above by $cnm2^{-b}$. If we never get a false positive, then our algorithm returns the correct output. Since $b = \lceil 3 \log_2(m) \rceil$ and $n \leq m$, we infer that $cnm2^{-b} \leq \frac{c}{m}$. Therefore, our algorithm works correctly with probability at least $1 - \frac{c}{m}$.

4.7 Removing hash tables

As mentioned earlier, there is a way to avoid using hash tables in our algorithm. Since the values of compressed hashes of edges are polynomial in m (at most $2m^3$), we are able to sort them in $\mathcal{O}(m)$ time using the radix sort. Now, instead of requiring a data structure that allows us to query the existence of an edge with a particular value of its compressed hash, we create an object representing such query, which we are going to answer in the future in an offline manner. After gathering all such objects, we may sort them together with edges, where as keys for comparisons we use values of compressed hashes for edges and values of required compressed hashes for queries. All the keys for comparisons are polynomial, so we are able to sort them together in $\mathcal{O}(m)$ time using the radix sort. After the sorting, we are able to easily answer all queries offline.

This concludes the description of the randomized variant of the algorithm.

5 Deterministic algorithm

Having established a linear time randomized algorithm producing 3-edge-cuts in 3-edge-connected graphs, we will now determinize it by designing deterministic implementations of

the subroutines for each of the cases considered in the randomized variant of the algorithm. Three cases need to be derandomized: “one tree edge” (Subsection 4.2), “two tree edges, lower case” (Subsection 4.3.1), and “two tree edges, upper case” (Subsection 4.3.2). In the following description, we cannot use compressed hashes anymore: the compression is a random process which inevitably results in false positives. Instead, we will exploit additional properties of 3-edge-cuts in order to produce an efficient deterministic implementation of the algorithm.

Recall that in the description of the randomized implementation of the algorithm, we defined the values $\text{low}(e)$, $\text{MaxUp}(e)$, $\text{MinDn}(e)$, and $\text{MaxDn}(e)$ for any tree edge e . The values $\text{low}(e)$ and $\text{MaxUp}(e)$ were defined as the back edges $x = uv$ leaping over e whose head v has the smallest possible preorder in $\mathcal{T}$, or the largest possible preorder in $\mathcal{T}$, respectively, breaking ties arbitrarily. The values $\text{MinDn}(e)$ and $\text{MaxDn}(e)$ were defined analogously, only that we chose edges with the smallest possible preorder of the source u , and the largest possible preorder of u , respectively.

For the deterministic variant of the algorithm, we generalize these notions: we define $\text{low1}(e)$, $\text{low2}(e)$, and $\text{low3}(e)$ as the three back edges leaping over e with the minimal preorders of their targets; in particular, we set $\text{low1}(e) := \text{low}(e)$. We analogously define $\text{MaxUp1}(e)$, $\text{MaxUp2}(e)$, $\text{MinDn1}(e)$, $\text{MinDn2}(e)$, $\text{MaxDn1}(e)$, and $\text{MaxDn2}(e)$. We remark that $\text{low3}(e)$ might not exist if there are fewer than three edges leaping over e ; in this case, we put $\text{low3}(e) := \perp$. However, all the other values must exist—otherwise, at most one edge would leap over e , which would mean that this edge, together with e , would form a 2-edge-cut of G of cardinality 2.

All the values defined above can be computed for every tree edge e in linear time with respect to the size of G : the values $\text{low1}(e)$, $\text{low2}(e)$, and $\text{low3}(e)$ are computed in a single depth-first search pass along $\mathcal{T}$ in the same way as the original low function is computed, only that three lowest back edges are computed instead of just one. The generalizations of MaxUp , MinDn , and MaxDn are determined in the same way as in Section 4, but $k = 2$ is passed to the algorithm from Theorem 4.3 instead of $k = 1$ so that two best back edges of each kind are computed instead of just one. Hence, in the following description, we will assume that all the values above have already been computed.

5.1 One tree edge

Recall that in this case, we are to find all 3-edge-cuts intersecting a fixed depth-first search tree $\mathcal{T}$ in a single edge. This is fairly straightforward: if some 3-edge-cut C contains exactly one tree edge e of $\mathcal{T}$, then the border of the cut is exactly $\mathcal{T}_e$; hence, this cut must include all back edges leaping over e . Therefore, e belongs to a 3-edge-cut of this kind if and only if $\text{low3}(e) = \perp$, i.e. if there only exist two back edges leaping over e ; in this case, e and these two edges form a 3-edge-cut. Since this check can be easily performed in $\mathcal{O}(n)$ time for all tree edges in $\mathcal{T}$, the whole subroutine runs in $\mathcal{O}(n + m)$ time complexity.

5.2 Two tree edges, lower case

Recall that this case requires us to find all 3-edge-cuts intersecting a fixed depth-first search tree $\mathcal{T}$ in two edges, say e and f , such that $e < f$ and the remaining back edge g connects $\mathcal{T}_f$ with $\mathcal{T}_e \setminus \mathcal{T}_f$ (Figure 2). Hence, g is the only back edge connecting $\mathcal{T}_f$ with $\mathcal{T}_e \setminus \mathcal{T}_f$, no back edges connect $\mathcal{T}_e \setminus \mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_e$, but there may be multiple back edges connecting $\mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_e$.

We shall exploit the fact that in a valid 3-edge-cut of this kind, there are no back edges originating from $\mathcal{T}_e \setminus \mathcal{T}_f$ and leaping over e . For a fixed edge e , this severely limits the set of possible tree edges f below e in $\mathcal{T}$, since all back edges leaping over e must originate from $\mathcal{T}_f$. This is formalized by the notion of the *deepest down cut* of e :

Definition 1 (deepest down cut). *For a tree edge e in $\mathcal{T}$, we define the deepest down cut of e , denoted $\text{DeepestDnCut}(e)$, as the deepest tree edge $f \geq e$ for which all back edges leaping over e originate from $\mathcal{T}_f$.*

Lemma 5.1. *For every tree edge e , $\text{DeepestDnCut}(e)$ is defined correctly and uniquely. Moreover, every 3-edge-cut $\{e, f, g\}$ of the considered kind satisfies $e < f \leq \text{DeepestDnCut}(e)$.*

Proof. Let $\mathcal{S}_e$ be the set of the tails of all the edges leaping over e . Then, $\text{DeepestDnCut}(e)$ is the deepest tree edge $f = uv$ such that $\mathcal{S}_e \subseteq V(\mathcal{T}_v)$. Equivalently, v must be an ancestor of all the vertices in $\mathcal{S}_e$; hence, the deepest such vertex, the *lowest common ancestor* of $\mathcal{S}_e$, is defined uniquely. The correctness of the definition of $\text{DeepestDnCut}(e)$ follows. For the latter part of the lemma, observe that $f \leq \text{DeepestDnCut}(e)$ if and only if every edge leaping over e originates from $\mathcal{T}_f$. $\square$

Lemma 5.2. *For every tree edge e , $\text{DeepestDnCut}(e) = uv$ is the tree edge whose head v is the lowest common ancestor of two vertices: the tails of $\text{MinDn1}(e)$ and $\text{MaxDn1}(e)$.*

Proof. Let $\mathcal{S}_e$ be the set defined in the proof of Lemma 5.1. Then, $\text{DeepestDnCut}(e) = uv$ is the tree edge whose head v is the lowest common ancestor of $\mathcal{S}_e$. Equivalently, v is the lowest common ancestor of two vertices: one of minimum preorder in $\mathcal{S}_e$, and the other of maximum preorder in $\mathcal{S}_e$. By definition, those two vertices are exactly the tails of $\text{MinDn1}(e)$ and $\text{MaxDn1}(e)$, respectively. $\square$

Thanks to Lemma 5.2, we can compute $\text{DeepestDnCut}(e)$ in constant time for each tree edge e as the lowest common ancestor of two vertices can be computed in constant time after linear preprocessing [14].

We now shift our focus to the lower tree edge f . As in Section 4, the only possible candidate g for a back edge of the cut leaping over f is given by $\text{MaxUp1}(f)$. The only problematic part is locating the remaining tree edge e . Previously, we utilized randomness in order to calculate the compressed hash of e given the compressed hashes of f and g . Here, we instead use the following fact:

Lemma 5.3. *If $\{e, f, g\}$ is a 3-edge-cut of the considered kind for some tree edges e and $f > e$, then e is the deepest tree edge satisfying $\text{DeepestDnCut}(e) \geq f$.*

Proof. Naturally, the condition $\text{DeepestDnCut}(e) \geq f$ must be satisfied if $\{e, f, g\}$ is a 3-edge-cut (Lemma 5.1). It only remains to prove that e is the deepest edge with this property. Pick two tree edges e_1, e_2 such that $e_1 < e_2 < f$, both satisfying the condition regarding their deepest down cuts. We then partition the tree into four parts: $\mathcal{T} \setminus \mathcal{T}_{e_1}$, $\mathcal{T}_{e_1} \setminus \mathcal{T}_{e_2}$, $\mathcal{T}_{e_2} \setminus \mathcal{T}_f$, and $\mathcal{T}_f$. By the assumptions, all back edges leaping over e_1 or over e_2 must originate from $\mathcal{T}_f$ (Figure 3).

Since G is 3-edge-connected, $\{e_2, f\}$ cannot be its edge cut; there must exist a back edge x_1 connecting $\mathcal{T}_{e_2} \setminus \mathcal{T}_f$ with the rest of the graph. It cannot leap over e_2 as it would also need to originate from $\mathcal{T}_f$; hence, x_1 connects $\mathcal{T}_f$ with $\mathcal{T}_{e_2} \setminus \mathcal{T}_f$. Analogously, $\{e_1, e_2\}$ is not

an edge cut of G , from which we infer that there exists another back edge x_2 connecting $\mathcal{T}_f$ with $\mathcal{T}_{e_1} \setminus \mathcal{T}_{e_2}$. Since we produced two separate back edges x_1, x_2 connecting $\mathcal{T}_f$ with $\mathcal{T}_{e_1} \setminus \mathcal{T}_f$, there cannot exist any 3-edge-cut containing both e_1 and f . The statement of the lemma follows. $\square$

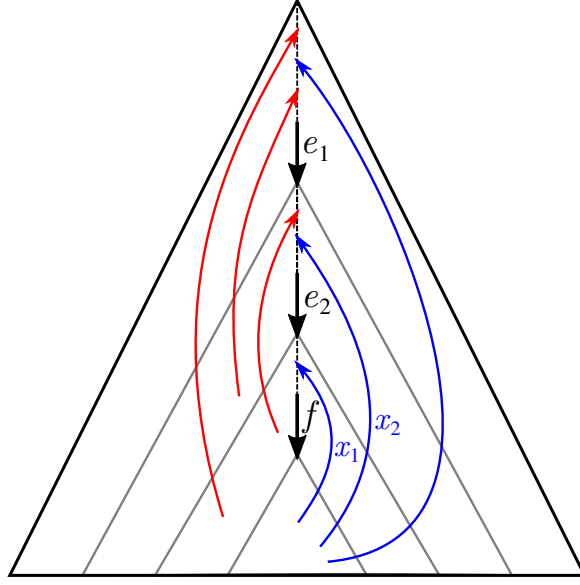


Figure 3: The setting in Lemma 5.3. Given that $\text{DeepestDnCut}(e_1) \geq f$ and $\text{DeepestDnCut}(e_2) \geq f$, the red back edges are forbidden from occurring in G , while the blue back edges can appear in G . We prove that both x_1 and x_2 appear in the graph.

Lemmas 5.1 and 5.3 naturally lead to the following idea: given a tree edge e , the set of edges f satisfying $\text{DeepestDnCut}(e) \geq f$ is a path P_e connecting the head of e with the head of $\text{DeepestDnCut}(e)$; we assign this path a weight w_e equal to the depth of e in $\mathcal{T}$. We then invoke Theorem 4.3 with the tree $\mathcal{T}$, the weighted paths $\{P_e \mid e \in E(\mathcal{T})\}$, and $k = 1$. This allows us to find, in $\mathcal{O}(n)$ time, for each tree edge $f \in E(\mathcal{T})$, the path P_e of the maximum weight containing f as an edge. This path naturally corresponds to the tree edge e from Lemma 5.3. This way, for every tree edge f , we have uniquely identified a back edge g and a tree edge e such that the only possible 3-edge-cut containing f as a deeper tree edge, if it exists, is $\{e, f, g\}$.

It only remains to verify that $\{e, f, g\}$ is a 3-edge-cut. We need to make sure that:

- All back edges leaping over e originate from $\mathcal{T}_f$. The equivalent condition $\text{DeepestDnCut}(e) \geq f$ is guaranteed by the algorithm, so we do not need to check it again.
- Exactly one edge (g) connects $\mathcal{T}_f$ with $\mathcal{T}_e \setminus \mathcal{T}_f$. Since $g = \text{MaxUp1}(f)$, it suffices to verify that the edge $\text{MaxUp2}(f)$, which is a back edge leaping over f whose head is the deepest apart from g , also leaps over e (i.e., it does not terminate in $\mathcal{T}_e$).

It can be easily verified that the implementation of the subroutine is deterministic and runs in linear time with respect to the size of G .

5.3 Two tree edges, upper case

Recall that in this case, we are required to find all 3-edge-cuts intersecting $\mathcal{T}$ in two edges e and f , such that $e < f$ and the remaining back edge g connects $\mathcal{T}_e \setminus \mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_e$ (Figure 2). Similarly to the randomized case, we use the fact that given a tree edge e , the tail of g has either the smallest preorder (i.e., $g = \text{MinDn1}(e)$) or the largest preorder (i.e., $g = \text{MaxDn1}(e)$) among all the back edges leaping over e . Without loss of generality, assume that $g = \text{MinDn1}(e)$; the latter case is analogous.

In a similar vein to previous case, observe that if $\{e, f, g\}$ is a 3-edge-cut for some $f > e$, then all back edges leaping over e other than g must originate from $\mathcal{T}_f$.

This leads to the following slight generalization of `DeepestDnCut` (Definition 1):

Definition 2. For a tree edge e in $\mathcal{T}$, we define the value $\text{DeepestDnCutNoMin}(e)$ as the deepest tree edge $f \geq e$ for which all back edges leaping over e other than $\text{MinDn1}(e)$ originate from $\mathcal{T}_f$.

The process of computation of $\text{DeepestDnCut}(e)$ asserted by Lemma 5.2 can be easily modified to match our needs: recall that $\text{DeepestDnCut}(e)$ is the tree edge whose head was the lowest common ancestor of the origins of $\text{MinDn1}(e)$ and $\text{MaxDn1}(e)$. Since we exclude $\text{MinDn1}(e)$ from the condition in Definition 2, we use the edge $\text{MinDn2}(e)$ instead: that is, the back edge leaping over e , with the next lowest preorder of the tail. Therefore, we compute $\text{DeepestDnCutNoMin}(e)$ as the tree edge whose head is the lowest common ancestor of the tails of $\text{MinDn2}(e)$ and $\text{MaxDn1}(e)$ (Figure 4).

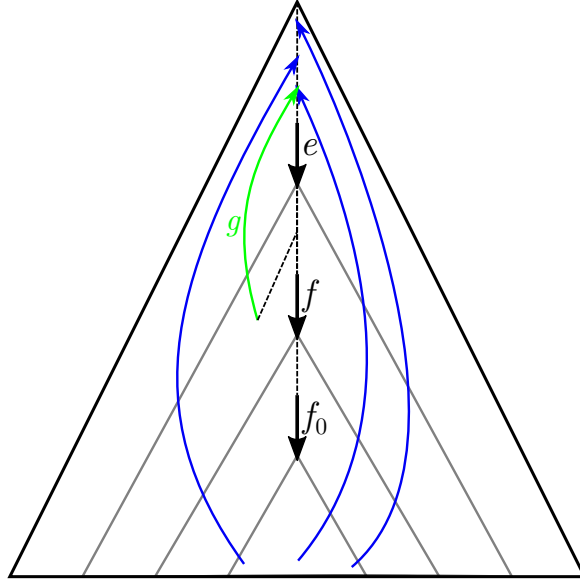


Figure 4: The computation of $f_0 = \text{DeepestDnCutNoMin}(e)$ excludes $g = \text{MinDn1}(e)$, and computes the lowest common ancestor of the remaining back edges leaping over e .

Now, fix the shallower tree edge e of the cut. Let $f_0 := \text{DeepestDnCutNoMin}(e)$. Then, if some tree edge $f > e$ belongs to the 3-edge-cut $\{e, f, g\}$, then $f \leq f_0$ (otherwise, there would be multiple edges connecting $\mathcal{T}_e \setminus \mathcal{T}_f$ with $\mathcal{T} \setminus \mathcal{T}_e$). The natural question is then: is $\{e, f_0, g\}$

a 3-edge-cut? The only possible problem is that there may exist back edges connecting $\mathcal{T}_{f_0}$ with $\mathcal{T}_e \setminus \mathcal{T}_{f_0}$. Fortunately, if this is the case, then the back edge $\text{MaxUp1}(f_0)$, dependent only on f_0 , is one of these back edges.

Hence, let $h_0 := \text{MaxUp1}(f_0)$. If h_0 leaps over e , we are done, and $\{e, f_0, g\}$ is an edge cut. Otherwise, the subtree $\mathcal{T}_f$ for the sought 3-edge-cut $\{e, f, g\}$ must contain the head of h_0 (or else h_0 would connect $\mathcal{T}_f$ with $\mathcal{T}_e \setminus \mathcal{T}_f$). Let then f_1 , $e \leq f_1 < f_0$ be the deepest tree edge containing the head of h_0 , that is, the edge whose head coincides with the head of h_0 . Then, the edge f of the 3-edge-cut $\{e, f, g\}$ must satisfy $e < f \leq f_1$. We can then repeat this procedure: given f_1 , we compute $h_1 := \text{MaxUp1}(f_1)$, and either h_1 leaps over e and we are done, or we calculate another edge $f_2 < f_1$ supplying a better bound on the depth of f (Figure 5). Since the graph is finite, this process terminates in $k = \mathcal{O}(n)$ steps, producing the lowest tree edge $f_k \geq e$ such that no back edge connects $\mathcal{T}_{f_k}$ with $\mathcal{T}_e \setminus \mathcal{T}_{f_k}$, and no back edge other than g connects $\mathcal{T}_e \setminus \mathcal{T}_{f_k}$ with $\mathcal{T} \setminus \mathcal{T}_e$. If $f_k = e$, then no 3-edge-cut $\{e, f, g\}$ exists for $f > e$. Otherwise, $\{e, f_k, g\}$ is naturally a correct 3-edge-cut. Since two edges of a 3-edge-cut uniquely identify the third edge (Lemma 4.5), we conclude that this is the only 3-edge-cut containing e and g .

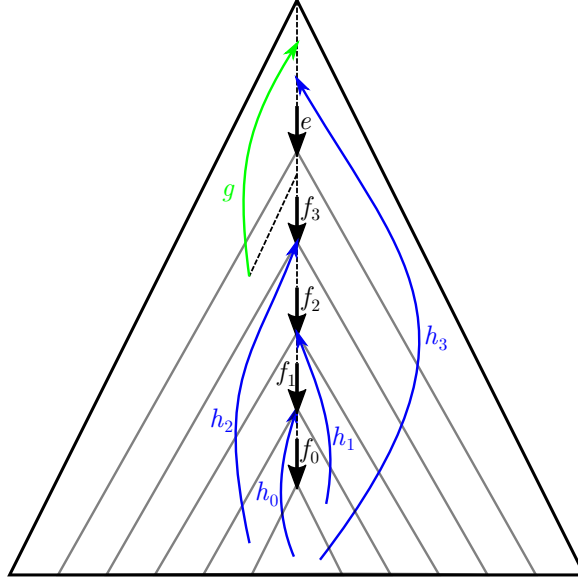


Figure 5: The main iteration of the algorithm. We set $f_0 = \text{DeepestDnCutNoMin}(e)$, $h_i = \text{MaxUp1}(f_i)$, and f_{i+1} is the tree edge with the same head as h_i .

We can now sum up the inefficient deterministic implementation of the algorithm for the current case (Algorithm 2).

Algorithm 2. Inefficient deterministic solution for “two tree edges, upper case”

```

function SOLVETWOTREEEDGESUPPER
  for  $e \in E(\mathcal{T})$  do
     $f_0 \leftarrow \text{DeepestDnCutNoMin}(e)$ 
     $g \leftarrow \text{MinDn1}(e)$ 
     $i \leftarrow 0$ 
    while  $f_i \neq e$  and  $\text{MaxUp1}(e).\text{head} \in \mathcal{T}_e$  do
       $h_i \leftarrow \text{MaxUp1}(e)$ 
       $f_{i+1} \leftarrow$  the tree edge with the same head as  $h_i$ 
       $i \leftarrow i + 1$ 
    if  $f_i \neq e$  then
      add  $\{e, f_i, g\}$  to the list of 3-edge-cuts
     $\triangleright$  Run the analogous algorithm for  $\text{DeepestDnCutNoMax}$  instead of  $\text{DeepestDnCutNoMin}$ .

```

In order to optimize the algorithm, we seek to eliminate the **while** loop. Indeed, for each $e \in E(\mathcal{T})$, the process is very similar: start with some edge $f > e$, and then repeatedly replace f with a higher edge, until a replacement would result in an edge closer to the root than e . We can model this process with a rooted tree $\mathcal{U}$ defined as follows:

- the vertices of $\mathcal{U}$ are the tree edges of $\mathcal{T}$, and $\perp$,
- $\perp$ is the root of $\mathcal{U}$,
- for a non-root vertex e of $\mathcal{U}$, its parent is the tree edge with the same head as $\text{MaxUp1}(e)$.
If the head of $\text{MaxUp1}(e)$ coincides with the root of $\mathcal{T}$, then the parent of e in $\mathcal{U}$ is $\perp$.

Naturally, $\mathcal{U}$ can be constructed in linear time with respect to the size of $\mathcal{T}$; moreover, if an edge p is a parent of another edge q in $\mathcal{U}$, then $p <_{\mathcal{T}} q$.

Now, the **while** loop is equivalent to repeated replacement of f_0 with its parent in $\mathcal{U}$ as long as the parent is greater or equal than e with respect to $<_{\mathcal{T}}$. In other words, the final edge $f' := \text{final}(f_0, e)$ is taken as the shallowest ancestor of f_0 in $\mathcal{U}$ for which $f' \geq_{\mathcal{T}} e$.

This leads to the final idea: we simulate a forest of rooted subtrees of $\mathcal{U}$ using a disjoint set union data structure $F_{\mathcal{U}}$ (Theorem 4.1). Each subtree additionally keeps its root, which can be retrieved from $F_{\mathcal{U}}$ (Lemma 4.2). Initially, each subtree of $\mathcal{U}$ contains a single vertex.

After the initialization of $F_{\mathcal{U}}$, we iterate e over the tree edges of $\mathcal{T}$ in the decreasing order of depth in $\mathcal{T}$. Throughout the process, we maintain the following invariant on $F_{\mathcal{U}}$: an edge $ef \in E(\mathcal{U})$ for $e < f$ has been added to the forest if and only if e has been considered at any previous iteration of the main loop. Hence, at the beginning of the iteration for a given edge e , we add to $F_{\mathcal{U}}$ all tree edges of $\mathcal{U}$ originating from e . At this point of time, for every tree edge f such that $f >_{\mathcal{T}} e$, the edge $\text{final}(f, e)$ is given by $F_{\mathcal{U}}.\text{lowest}(f)$. This reduces the entire **while** loop to a single **lowest** query on $F_{\mathcal{U}}$ (Algorithm 3).

Algorithm 3. Efficient deterministic solution for “two tree edges, upper case”

```

function SOLVETWOTREEEDGESUPPER
   $\mathcal{U} \leftarrow$  the rooted tree on  $E(\mathcal{T}) \cup \{\perp\}$  defined above
   $F_{\mathcal{U}} \leftarrow$  the disjoint set union data structure built on  $\mathcal{U}$  (Lemma 4.2)
  for  $e \in E(\mathcal{T})$ , in order from the deepest to the shallowest do
    for  $c$  – child of  $e$  in  $\mathcal{U}$  do
       $F_{\mathcal{U}}.\text{union}(e, c)$ 
       $g \leftarrow \text{MinDn1}(e)$ 
       $f_0 \leftarrow \text{DeepestDnCutNoMin}(e)$ 
       $f' \leftarrow F_{\mathcal{U}}.\text{lowest}(f_0)$ 
      if  $f' \neq e$  then
        add  $\{e, f', g\}$  to the list of 3-edge-cuts
   $\triangleright$  Run the analogous algorithm for DeepestDnCutNoMax instead of DeepestDnCutNoMin.

```

It can be easily seen that we initialize $F_{\mathcal{U}}$ on a tree with n vertices, and we issue $\mathcal{O}(n)$ queries to it in total. Therefore, the whole subroutine runs in time linear with respect to the size of G .

Summing up, we replaced each randomized subroutine with its deterministic counterpart, preserving the linear guarantee on the runtime of the algorithm. We conclude that there exists a deterministic linear-time algorithm listing 3-edge-cuts in 3-edge-connected graphs.

6 Reconstructing the structure of 4-edge-connected components

In this section, we show how to build a structure of 4-edge-connected components of a 3-edge-connected graph G , given the set $\mathcal{C}$ of all 3-edge-cuts in G .

First of all, recall what such a structure looks like.

Theorem 6.1. [8, Corollary 8] *For a 3-edge-connected graph $G = (V, E)$, there exists a tree $H = (U, F)$ along with functions $\phi : \mathcal{C} \rightarrow F$ and $\psi : V \rightarrow U$, such that ϕ is a bijection from the 3-edge-cuts of G to the edges of H , and ψ maps (not necessarily surjectively) the vertices of G to the vertices of H in such a way that the whole 4-edge-connected components are mapped to the same vertex.*

Moreover, if a 3-edge-cut c partitions the vertices of G into two parts V_1 and V_2 , then $\phi(c)$ partitions the vertices of H into U_1 and U_2 such that $\psi^{-1}(U_1) = V_1$ and $\psi^{-1}(U_2) = V_2$.

Figure 6 shows an example of decomposition postulated by Theorem 6.1.

The tree H is usually unrooted in the literature. However, we are going to root it. Namely, we take a depth-first search tree $\mathcal{T}$ of G , rooted at some vertex r , and we root H at $\psi(r)$. For a vertex $u \in U$, we let H_u denote the subtree of H rooted at u .

Definition 3. *For a 3-edge-cut c , we define $P(c)$ as the set of vertices from the connected component of $G \setminus c$ not containing r .*

We remark that since c is a minimal cut, $G \setminus c$ consists of two connected components, so $P(c)$ is determined uniquely.

Lemma 6.2. *Let $v, u \in U$, and let $e_1, e_2, \dots, e_k \in F$ be the sequence of edges of H on the path from v and u in H . If v is an ancestor of u , then for each pair of integers i, j such that $1 \leq i < j \leq k$, we have $|P(\phi^{-1}(e_i))| > |P(\phi^{-1}(e_j))|$.*

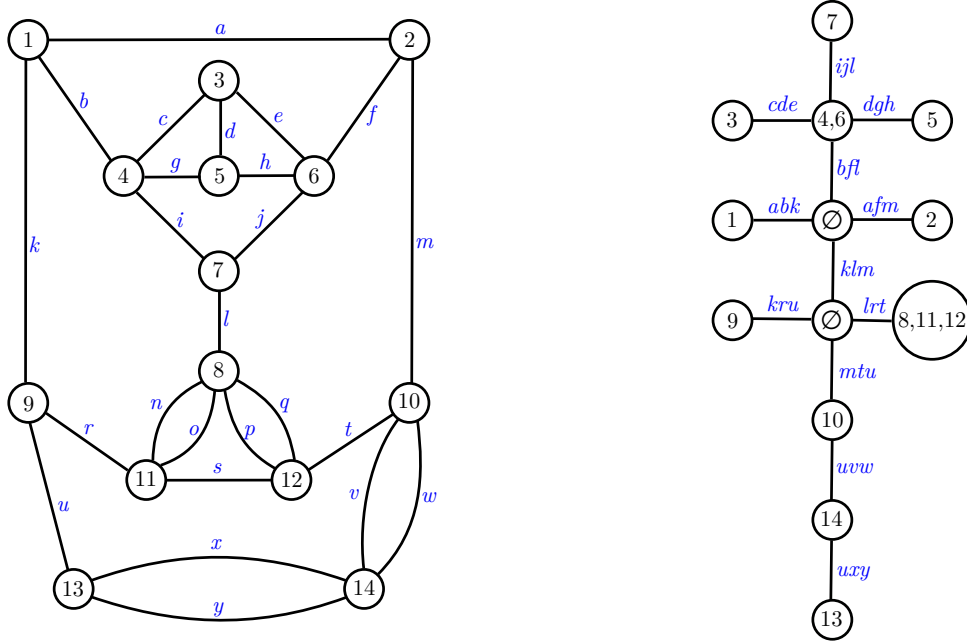


Figure 6: A 3-edge-connected graph with 14 vertices and 25 edges (left) and its tree H (right).

Proof. It suffices to only consider the inequalities for the consecutive edges from the considered path. To this end, let us fix some $i \in [1, k - 1]$ and consider the edges $(x, y) := e_i$ and $(y, z) := e_{i+1}$. By the definition of H , we have $P(\phi^{-1}(e_i)) = \psi^{-1}(H_y)$ since both sides describe the connected component of $G \setminus \phi^{-1}(e_i)$ not containing r . Similarly, $P(\phi^{-1}(e_{i+1})) = \psi^{-1}(H_z)$. As H_z is a subtree of H_y , we infer that

$$P(\phi^{-1}(e_{i+1})) = \psi^{-1}(H_z) \subseteq \psi^{-1}(H_y) = P(\phi^{-1}(e_i)).$$

This implies that $|P(\phi^{-1}(e_i))| \geq |P(\phi^{-1}(e_{i+1}))|$. However, observe that each set $P(c)$ uniquely defines a cut in the graph G as the set of all edges between $P(c)$ and $V \setminus P(c)$. By the definition of H , we have $\phi^{-1}(e_i) \neq \phi^{-1}(e_{i+1})$. Therefore, $P(\phi^{-1}(e_i)) \neq P(\phi^{-1}(e_{i+1}))$, which finishes the proof of the lemma. $\square$

Lemma 6.3. *Given a 3-edge-connected graph G and the set of all its 3-edge-cuts $\mathcal{C}$, the sizes of $P(c)$ for all $c \in \mathcal{C}$ can be computed in linear time with respect to the size of G .*

Proof. Let us consider a 3-edge-cut c . If two vertices are connected by an edge (u, v) of the cut, then exactly one of u, v is contained in $P(c)$. Similarly, if an edge (u, v) is not contained in c , then $u \in P(c) \Leftrightarrow v \in P(c)$. The root r is by definition not in $P(c)$. Therefore, to decide whether a vertex v belongs to $P(c)$, it is enough to check the parity of the number of edges of the cut on the path from r to v in $\mathcal{T}$. Precisely, let us define $\text{cntOnPath}(c, v) := \sum_{e \in c \cap \mathcal{T}} [v \in \mathcal{T}_e]$ as the number of the tree edges of the cut c on the path from the root of $\mathcal{T}$ to a vertex $v \in V(\mathcal{T})$, and $\text{deeper}(e)$ as the deeper endpoint of e in $\mathcal{T}$. Then, a vertex v is disconnected from the root r by the 3-edge-cut c if there is an odd number of the edges of c on the path from r to v in $\mathcal{T}$:

$$|P(c)| = \sum_{v \in V} [2 \nmid \text{cntOnPath}(c, v)] = \sum_{\substack{e \in \mathcal{T} \cap c \\ 85}} (-1)^{1 + \text{cntOnPath}(c, \text{deeper}(e))} \cdot |\mathcal{T}_e|.$$

Assuming the sizes of the subtrees of $\mathcal{T}$ have been precomputed, the value of $P(c)$ can be computed for a particular cut c in constant time. The total time complexity of this algorithm is $\mathcal{O}(m + |\mathcal{C}|)$, which is linear in the size of G as $|\mathcal{C}| \leq \mathcal{O}(n)$. $\square$

To reconstruct H , we will also use the following structural lemma about cuts sharing the same edge of graph G .

Lemma 6.4. *For an edge $(u, v) = e \in E$, let $l(e)$ be the set of all 3-edge-cuts containing e . The image $\phi(l(e))$, i.e., the set of all edges of H corresponding to the edge cuts containing e , forms a path in H between $\psi(u)$ and $\psi(v)$.*

Proof. Let us consider the vertices $t_u := \psi(u)$ and $t_v := \psi(v)$ of H , and pick an edge $f \in F$. If f lies on the path between t_u and t_v in H , then f divides H into two parts U_1 and U_2 such that $t_u \in U_1$ and $t_v \in U_2$. Thus, f corresponds to the cut $\phi^{-1}(f)$ separating u from v , which must necessarily contain $e = (u, v)$ as an edge.

Similarly, if f does not lie on the path between t_u and t_v in H , then f divides H into two parts U_1 and U_2 such that $t_u, t_v \in U_1$ and $t_u, t_v \notin U_2$. Therefore, the cut $\phi^{-1}(f)$ does not separate u from v . Since this cut is minimal, we get that $e \notin \phi^{-1}(f)$. This completes the proof. $\square$

We remark an edge case in Lemma 6.4: if $\psi(u) = \psi(v)$ for some edge $e = (u, v)$, then the image $\phi(l(e))$ is empty. Moreover, since each 3-edge-cut $c \in \mathcal{C}$ contains at least one tree edge of $\mathcal{T}$, each edge $\phi(c)$ is covered by at least one path $\phi(l(e))$ for $e \in E(\mathcal{T})$. Equivalently, H is a tree, rooted at $\psi(r)$, equal to the union of all paths $\phi(l(e))$ for $e \in E(\mathcal{T})$. This representation of H is the cornerstone of our algorithm reconstructing H from G and $\mathcal{C}$.

Lemma 6.5. *There exists a linear time algorithm which, given a graph G and the list $\mathcal{C}$ of all 3-edge-cuts of G , constructs the tree H , along with the mappings ϕ and ψ .*

Proof. First, for each edge $e \in \mathcal{T}$, one can create a list $l(e)$ of all 3-edge-cuts containing e . We sort each such list decreasingly by the size of $P(c)$; this can be done using radix sort in linear time.

Lemma 6.3 guarantees an important property of each sorted list of cuts: if two cuts c_1, c_2 of the same list map to the edges e_1, e_2 in H , respectively, such that $e_1 <_H e_2$, then c_1 appears earlier in the list than c_2 .

Let $e_1, e_2, \dots, e_{n-1} \in E(\mathcal{T})$ be the sequence of edges visited by a depth-first search of $\mathcal{T}$. We note that this sequence of edges is consistent with the tree order $<_{\mathcal{T}}$; i.e., if $e_i <_{\mathcal{T}} e_j$ for some two edges in the sequence, then $i < j$. Let $e_i = u_i v_i$, where u_i is the vertex of $\mathcal{T}$ closer to the root r .

We will create the sought tree H iteratively. Initially, take H as the tree containing a single vertex $\psi(r)$. We consider the edges $e_1, e_2, \dots, e_{n-1}$ of $\mathcal{T}$ in this order, maintaining the following invariant after k iterations of the algorithm:

Invariant. H is a connected tree, rooted at $\psi(r)$, equal to the union of all paths $\phi(l(e_i))$ for $i \in \{1, 2, \dots, k\}$.

It is clear that after $n - 1$ iterations, H will be the rooted tree representing the structure of all 3-edge-cuts in G .

Consider the k -th iteration of the algorithm, $k \in \{1, 2, \dots, n - 1\}$. In this iteration, we need to add to H the path $\phi(l(e_k))$, which originates from $\psi(u_k)$ and terminates at $\psi(v_k)$.

To this end, we first notice that $\psi(u_k)$ already exists as a vertex of H : either $u_k = r$, which means that $\psi(u_k)$ is the root of H ; or u_k is the head v_t of some edge e_t earlier in the order, which implies that $\psi(u_k) = \psi(v_t)$ exists in H as one of the endpoints of $\phi(l(e_t))$. Hence, some prefix of $\phi(l(e_k))$ already exists in H , and it only remains to add the suffix of this path to H .

The considered path consists of two vertical parts: the first, from $\psi(u_k)$ to the lowest common ancestor of $\psi(u_k)$ and $\psi(v_k)$ towards the root $\psi(r)$, and the second, from the lowest common ancestor to $\psi(v_k)$ away from the root of H . Fortunately, the first part of the path is already included in H : since the tree is rooted, the entire vertical path from the root $\psi(r)$ to $\psi(u_k)$ is present in H . Then, the second part of the path is clearly formed by the edges of $l(e)$ not present in the first part of the path, sorted in the same order as $l(e)$ (Lemma 6.2).

Now, adding the path $\phi(l(e_k))$ to H is rather straightforward: first, starting from $\psi(u)$, we go up the tree H along the edges of H corresponding to the edge cuts containing e_k . Then, we start going down the tree: we iterate the list $l(e_k)$ of cuts, excluding the cuts that correspond to the edges visited in the first part of the traversal. For each such cut, we go down the tree along the edge corresponding to the cut (creating it, if necessary). This is summed up by the following implementation of a single iteration of the algorithm:

Algorithm 4. A single iteration of the algorithm

```

function ADDPATHTOH( $e$ )                                      $\triangleright e = (u, v)$  is an edge of  $\mathcal{T}$ 
   $x \leftarrow \psi(u)$ 
   $\triangleright$  Go up the tree along the first part of the path, marking the edges visited on the way.
  while  $e \in \phi^{-1}(\text{EdgeToParent}(x))$  do
     $\text{touched}(\phi^{-1}(\text{EdgeToParent}(x))) \leftarrow e$ 
     $x \leftarrow \text{parent}(x)$ 
     $\triangleright$  Go down the tree along the edges of  $l(e)$  not visited by the first loop.
  for  $c \in l(e)$  do
    if  $\text{touched}(c) \neq e$  then
      if  $\phi(c) = \perp$  then
         $\phi(c) \leftarrow$  a fresh edge in  $H$  from  $x$  to a new vertex, corresponding to the cut  $c$ 
      assert the shallower end of  $\phi(c)$  is equal to  $x$ 
       $x \leftarrow$  the deeper end of  $\phi(c)$ 
   $\psi(v) \leftarrow x$ 

```

We remark that each 3-edge-cut c is considered in only a constant number of calls to ADDPATHTOH: an edge of H corresponding to c is traversed by ADDPATHTOH(e) only if $e \in c$. Therefore, the total time complexity of all calls to ADDPATHTOH is $\mathcal{O}(m + |\mathcal{C}|) = \mathcal{O}(m + n)$. $\square$

This concludes the construction of a tree representing all 3-edge-cuts in G . As a result, each vertex of H , as long as it is not empty, contains a single 4-edge-connected component of G . Hence, this algorithm also computes the decomposition of a 3-edge-connected graph G into 4-edge-connected components in total linear time.

7 Open problems

As a natural open problem whose resolving would complement this result nicely, we suggest investigating whether it is possible to design an algorithm computing 4-vertex-connected components in linear time.

Problem 1. Given an undirected graph $G = (V, E)$, is it possible to find all 4-vertex-connected components of G in linear time?

Additionally, it might be worth investigating whether this result can be lifted to higher connectivities. Admittedly, this seems quite complicated as the algorithm presented in Sections 4 and 5 is crafted specifically for 4-edge-connectivities.

Problem 2. Given an undirected graph $G = (V, E)$, is it possible to find all 5-edge-connected components of G in linear time?

We also remark that our algorithm assumes the word RAM model in which we can perform any arithmetic and bitwise operations on $\mathcal{O}(\log n)$ -bit words in constant time; this is required by Theorem 4.1. The natural question is whether this assumption can be avoided.

Problem 3. Given an undirected graph $G = (V, E)$, is it possible to find all 4-edge-connected components of G in linear time in the pointer machine model?

References

- [1] N. Bhardwaj, A. M. Lovett, and B. Sandlund. A Simple Algorithm for Minimum Cuts in Near-Linear Time. In S. Albers, editor, *17th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2020, June 22-24, 2020, Tórshavn, Faroe Islands*, volume 162 of *LIPICs*, pages 12:1–12:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- [2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.
- [3] G. Di Battista and R. Tamassia. On-Line Graph Algorithms with SPQR-Trees. In M. S. Paterson, editor, *Automata, Languages and Programming*, pages 598–611, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.
- [4] Y. Dinitz. The 3-edge-components and a structural description of all 3-edge-cuts in a graph. In E. W. Mayr, editor, *Graph-Theoretic Concepts in Computer Science*, pages 145–157, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.
- [5] Y. Dinitz, A. V. Karzanov, and M. V. Lomonosov. On the structure of a family of minimum weighted cuts in a graph. *Studies in Discrete Optimization*, page 290–306, 1976.
- [6] Y. Dinitz and J. R. Westbrook. Maintaining the Classes of 4-Edge-Connectivity in a Graph On-Line. *Algorithmica*, 20(3):242–276, Mar 1998.

- [7] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, Sept. 1997.
- [8] T. Fleiner and A. Frank. A quick proof for the cactus representation of mincuts. 2009.
- [9] G. N. Frederickson. Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications. *SIAM Journal on Computing*, 14(4):781–798, 1985.
- [10] H. N. Gabow and R. E. Tarjan. A Linear-Time Algorithm for a Special Case of Disjoint Set Union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985.
- [11] Z. Galil and G. F. Italiano. Fully Dynamic Algorithms for Edge Connectivity Problems. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, STOC '91, page 317–327, New York, NY, USA, 1991. Association for Computing Machinery.
- [12] Z. Galil and G. F. Italiano. Reducing edge connectivity to vertex connectivity. *SIGACT News*, 22(1):57–61, Mar. 1991.
- [13] R. E. Gomory and T. C. Hu. Multi-Terminal Network Flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.
- [14] D. Harel and R. E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
- [15] R. Hariharan, T. Kavitha, and D. Panigrahi. Efficient Algorithms for Computing All Low s-t Edge Connectivities and Related Problems. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, page 127–136, USA, 2007. Society for Industrial and Applied Mathematics.
- [16] M. Henzinger, S. Rao, and D. Wang. *Local Flow Partitioning for Faster Edge Connectivity*, pages 1919–1938.
- [17] M. R. Henzinger. Fully dynamic biconnectivity in graphs. *Algorithmica*, 13(6):503–538, Jun 1995.
- [18] M. R. Henzinger and V. King. Randomized Dynamic Graph Algorithms with Poly-logarithmic Time per Operation. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '95, page 519–527, New York, NY, USA, 1995. Association for Computing Machinery.
- [19] M. R. Henzinger and M. Thorup. Sampling to Provide or to Bound: With Applications to Fully Dynamic Graph Algorithms. *Random Struct. Algorithms*, 11(4):369–379, Dec. 1997.
- [20] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, July 2001.
- [21] J. Holm and E. Rotenberg. Worst-Case Polylog Incremental SPQR-Trees: Embeddings, Planarity, and Triconnectivity. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '20, page 2378–2397, USA, 2020. Society for Industrial and Applied Mathematics.

- [22] J. Holm, E. Rotenberg, and M. Thorup. Dynamic Bridge-Finding in $\tilde{O}(\log^2 n)$ Amortized Time. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '18, page 35–52, USA, 2018. Society for Industrial and Applied Mathematics.
- [23] J. Hopcroft and R. Tarjan. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Commun. ACM*, 16(6):372–378, June 1973.
- [24] J. Hopcroft and R. Tarjan. Dividing a Graph into Triconnected Components. *SIAM Journal on Computing*, 2(3):135–158, 1973.
- [25] S.-E. Huang, D. Huang, T. Kopelowitz, and S. Pettie. Fully Dynamic Connectivity in $O(\log n(\log \log n)^2)$ Amortized Expected Time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, page 510–520, USA, 2017. Society for Industrial and Applied Mathematics.
- [26] A. Kanevsky, R. Tamassia, G. Di Battista, and J. Chen. On-Line Maintenance of the Four-Connected Components of a Graph. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 793–801, 1991.
- [27] B. M. Kapron, V. King, and B. Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, page 1131–1142, USA, 2013. Society for Industrial and Applied Mathematics.
- [28] D. R. Karger. Minimum Cuts in Near-Linear Time. *J. ACM*, 47(1):46–76, Jan. 2000.
- [29] K. Kawarabayashi and M. Thorup. Deterministic Global Minimum Cut of a Simple Graph in Near-Linear Time. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '15, page 665–674, New York, NY, USA, 2015. Association for Computing Machinery.
- [30] J. A. La Poutré and J. Westbrook. Dynamic 2-Connectivity with Backtracking. *SIAM J. Comput.*, 28(1):10–26, Feb. 1999.
- [31] J. A. La Poutré, J. van Leeuwen, and M. H. Overmars. Maintenance of 2- and 3-edge-connected components of graphs I. *Discrete Mathematics*, 114(1):329–359, 1993.
- [32] H. Nagamochi and T. Ibaraki. A linear time algorithm for computing 3-edge-connected components in a multigraph. *Japan Journal of Industrial and Applied Mathematics*, 9(2):163, Jun 1992.
- [33] N. Norouzi and Y. H. Tsin. A simple 3-edge connected component algorithm revisited. *Information Processing Letters*, 114(1):50–55, 2014.
- [34] R. Peng, B. Sandlund, and D. D. Sleator. Optimal Offline Dynamic 2, 3-Edge/Vertex Connectivity. In Z. Friggstad, J. Sack, and M. R. Salavatipour, editors, *Algorithms and Data Structures - 16th International Symposium, WADS 2019, Edmonton, AB, Canada, August 5-7, 2019, Proceedings*, volume 11646 of *Lecture Notes in Computer Science*, pages 553–565. Springer, 2019.
- [35] S. Pettie and L. Yin. The Structure of Minimum Vertex Cuts, 2021.

- [36] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
- [37] R. E. Tarjan and J. van Leeuwen. Worst-Case Analysis of Set Union Algorithms. *J. ACM*, 31(2):245–281, Mar. 1984.
- [38] M. Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, STOC '00, page 343–350, New York, NY, USA, 2000. Association for Computing Machinery.
- [39] Y. H. Tsin. A Simple 3-Edge-Connected Component Algorithm. *Theory of Computing Systems*, 40(2):125–142, Feb 2007.
- [40] Y. H. Tsin. Yet another optimal algorithm for 3-edge-connectivity. *Journal of Discrete Algorithms*, 7(1):130–146, 2009. Selected papers from the 1st International Workshop on Similarity Search and Applications (SISAP).
- [41] J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7(1):433–464, Jun 1992.
- [42] C. Wulff-Nilsen. Faster Deterministic Fully-Dynamic Graph Connectivity. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, page 1757–1769, USA, 2013. Society for Industrial and Applied Mathematics.

On Fully Dynamic Strongly Connected Components

Adam Karczmarz ✉ 

University of Warsaw, Poland
IDEAS NCBR, Warsaw, Poland

Marcin Smulewicz ✉

University of Warsaw, Poland

Abstract

We consider maintaining strongly connected components (SCCs) of a directed graph subject to edge insertions and deletions. For this problem, we show a randomized algebraic data structure with conditionally tight $O(n^{1.529})$ worst-case update time. The only previously described subquadratic update bound for this problem [Karczmarz, Mukherjee, and Sankowski, STOC'22] holds exclusively in the amortized sense.

For the less general dynamic strong connectivity problem, where one is only interested in maintaining whether the graph is strongly connected, we give an efficient deterministic black-box reduction to (arbitrary-pair) dynamic reachability. Consequently, for dynamic strong connectivity we match the best-known $O(n^{1.407})$ worst-case upper bound for dynamic reachability [van den Brand, Nanongkai, and Saranurak FOCS'19]. This is also conditionally optimal and improves upon the previous $O(n^{1.529})$ bound. Our reduction also yields the first fully dynamic algorithms for maintaining the minimum strong connectivity augmentation of a digraph.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases dynamic strongly connected components, dynamic strong connectivity, dynamic reachability

Digital Object Identifier 10.4230/LIPIcs.ESA.2023.68

Funding Adam Karczmarz: Partially supported by the ERC CoG grant TugbOAT no 772346 and the National Science Centre (NCN) grant no. 2022/47/D/ST6/02184.

1 Introduction

Two vertices of a directed graph $G = (V, E)$ are *strongly connected* if they can reach each other via a path in G . Pairwise strong connectivity is an equivalence relation and the *strongly connected components* of G are the equivalence classes of that relation. The graph G is called *strongly connected* if it has a single strongly connected component, or, equivalently, the transitive closure of G is a complete digraph. Testing strong connectivity and computing strongly connected components (SCCs) are among the most fundamental algorithmic problems on digraphs. Tarjan [24] famously showed a DFS-based linear-time algorithm for finding SCCs, which now, along with later alternatives [11, 23], is often taught in basic algorithms courses and appears in textbooks [9].

We study maintaining strong connectivity and strongly connected components in dynamic setting. A dynamic graph data structure is called *incremental* if it can handle edge insertions only, *decremental* if it can handle edge deletions only, and *fully dynamic* if it can handle both. When designing dynamic graph data structures, one is typically interested in optimizing both the (amortized or worst-case) update time of the data structure, and the query time. In the partially dynamic settings (i.e., incremental or decremental) one usually optimizes the total update time, i.e., the time needed to process the entire sequence of updates. In the following, let $n = |V|$ and $m = |E|$.



© Adam Karczmarz and Marcin Smulewicz;
licensed under Creative Commons License CC-BY 4.0

31st Annual European Symposium on Algorithms (ESA 2023).

Editors: Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman; Article No. 68;
pp. 68:1–68:15



Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper, we focus on the fully dynamic setting with single-edge updates. We are interested in maintaining the information about the *global* structure of SCCs of G , as opposed to supporting pairwise strong-connectivity queries. The information of interest may be either:

- (a) a single bit indicating whether G is strongly connected (let us call this variant SC),
- (b) the number of SCCs of G ($\#SCC$),
- (c) a representation of the SCCs allowing very efficient access to the individual SCCs, that is, computing the size in $O(\text{polylog } n)$ time, or reporting the elements in $O(\text{polylog } n)$ time per element ($SCCs$).

In particular, (c) can be achieved by maintaining an explicit $\Theta(n)$ -sized partition of V into SCCs, which also allows for $O(1)$ -time pairwise strong connectivity queries. On the other hand, using pairwise strong connectivity queries cannot easily provide any of the considered global information. In the following, for simplicity, we assume the desired information is explicitly recomputed after each update so that there is no need to consider query time. Thus, our sole goal is to optimize the *worst-case* update time.

Lower bounds. Note that if one is required to maintain the SCCs of a fully dynamic graph explicitly, then a single update can cause quite dramatic $\Omega(n)$ -sized change in the set of SCCs (consider a directed cycle). Abboud and Vassilevska Williams [1] showed that even for maintaining a single-bit information whether G has more than two SCCs (the $SC2$ problem), a data structure with $O(n^{1-\epsilon})$ amortized time¹ is unlikely, as it would break the Orthogonal Vectors conjecture, which is implied by SETH [15, 27]. The SETH-hardness of [1] does not seem to extend to SC though, which suggests that $\#SCC$ (or even $SC2$) might be a computationally harder problem.

The more recent OMv conjecture [14] implies that one cannot achieve $O(n^{1-\epsilon})$ update time even for SC . This suggests that for sparse graphs with $m = \tilde{O}(n)$ the trivial recompute-from-scratch approach might be the best possible approach for SC and $SCCs$.

van den Brand, Nanongkai, and Saranurak [26] developed a few extensions of the OMv conjecture and proved, based on those, that fully dynamic s, t -reachability (where $s, t \in V$ are fixed) currently requires $\Omega(n^{1.406})$ amortized update time, and fully dynamic single-source reachability (with only the source s fixed) currently requires $\Omega(n^{1.528})$ time per update.² This means that the state-of-the-art data structures [26, 20] for the respective variants are near-optimal. By known reductions (see, e.g., [1]), one obtains that SC requires $\Omega(n^{1.406})$ update time, whereas $SCCs$ require $\Omega(n^{1.528})$ update time.

Upper bounds. Abboud and Vassilevska Williams [1] showed that in order to have $\tilde{O}(n^{2-\epsilon})$ amortized update time for SC using fast matrix multiplication (FMM) is essential, thus ruling out non-trivial combinatorial approaches. And indeed, FMM-based algebraic dynamic reachability algorithms [20, 26] allow breaking the quadratic update bound. This is because

¹ Abboud and Vassilevska Williams [1] actually write that $\Omega(m^{1-o(1)})$ is the best bound one can hope for. However, since this bound is a function of m exclusively, it is required to hold only for *some* density regime, e.g., sparse graphs.

² More specifically, based on their OMv conjecture variants, van den Brand, Nanongkai, and Saranurak [26] proved, for any $\epsilon > 0$, an $\Omega(\min_{a,b \in [0,1]} (n^{a+b} + n^{\omega(1,a,1)-a} + n^{\omega(1,a,b)-b}) \cdot n^{-\epsilon})$ lower bound for dynamic s, t -reachability, and an $\Omega(n^{1+\rho-\epsilon})$ lower bound for dynamic single-source reachability (SSR), where $\rho < 0.529$ satisfies $1 + 2\rho = \omega(1, \rho, 1)$. Here $O(n^{\omega(a,b,c)})$ denotes the time needed to multiply an $n^a \times n^b$ boolean matrix by an $n^b \times n^c$ boolean matrix. These two conditional lower bounds become $\Omega(n^{1.406})$ and $\Omega(n^{1.528})$, respectively, assuming the current exponents of rectangular matrix multiplication. Furthermore, they become $\Omega(n^{1+1/4-\epsilon})$ and $\Omega(n^{1+1/2-\epsilon})$, respectively, assuming $\omega = 2$.

SC easily reduces to maintaining reachability to and from an arbitrary single vertex [26]. As a result, $O(n^{1.529})$ worst-case update time is possible in this case. This state-of-the-art bound does not match the $O(n^{1.406})$ lower bound [26], though. More generally, since the fully dynamic data structure of Sankowski [20] allows relatively cheap $O(n^{0.529})$ -time arbitrary-pair reachability queries³ at the cost of $O(n^{1.529})$ update time, this approach generalizes to maintaining some *at most* k distinct SCCs of G in $O(kn^{1.529})$ worst-case time per update. In particular, for $k = 2$ this trivially yields a solution to SC2 with $O(n^{1.529})$ worst-case update time. This matches the best known upper bound for SC. Consequently, the state-of-the-art bound for SC does not reflect the intuition of [1] that SC2 might be a harder problem.

The described approach, however, does not easily allow recomputing *all* SCCs of G , or even counting them, within subquadratic update time. In fact, the more general #SCC and SCCs problems do not seem to reduce to inspecting some $O(n^{2-\epsilon})$ -sized subset of the reachability matrix of G . Nevertheless, Karczmarz, Mukherjee, and Sankowski [17] recently showed that SCCs (and thus #SCC) can be maintained in $O(n^{1.529})$ amortized time per update. Their approach is a mix of combinatorial and algebraic methods and critically depends on the efficiency of a *decremental SCCs* data structure. Bernstein, Probst Gutenberg, and Wulff-Nilsen [7] gave a near-optimal data structure maintaining SCCs *explicitly* under edge deletions with $\tilde{O}(m)$ total update time. Although the $O(n^{1.529})$ amortized bound is near-optimal, the worst-case update time of the data structure of [7] might be $\Theta(n^2)$, and thus the worst-case update time of the fully dynamic SCCs data structure [17] may be as much as $\Theta(n^2)$ as well. To the best of our knowledge, no non-trivial worst-case bound for either #SCC or SCCs under fully dynamic edge updates has been described to date.

1.1 Our results

First of all, we close the gaps between the best known lower- and upper (worst-case) bounds for both fully dynamic strong connectivity (SC) and fully dynamic strongly connected components (SCC) in general directed graphs. See also Table 1. We achieve that by showing novel combinatorial reductions to the known fully dynamic arbitrary-pair reachability data structures [26, 20].

Fully dynamic strongly connected components. For fully dynamic SCCs we prove:

► **Theorem 1.** *Let G be a digraph. Suppose there is a fully dynamic reachability data structure $\mathcal{D}$ processing single-edge updates and arbitrary-pair queries on G in $U(n)$ and $Q(n)$ time respectively. Suppose the answers produced by $\mathcal{D}$ are correct with high probability⁴.*

Then one can explicitly maintain the SCCs of G subject to single-edge insertions and deletions in $\tilde{O}(U(n) + n \cdot Q(n))$ time per update (worst-case if the bounds $U(n), Q(n)$ are worst-case). The obtained data structure is Monte Carlo randomized. The answers produced are correct with high probability.

It is worth noting that the data structure trivially works against an adaptive adversary⁵ since the revealed information, the SCCs, depend only on the current graph G . As far as the adversarial model of the assumed data structure $\mathcal{D}$ is concerned, no special requirements

³ Throughout, we use the term *arbitrary-pair reachability* to refer to the situation when the endpoints s, t of a reachability pair of interest are a part of a query and may vary for different queries. The output of a query is a single bit indicating whether s can reach t in the (current) graph.

⁴ That is, with probability at least $1 - 1/n^c$, where the constant $c \geq 1$ can be set arbitrarily.

⁵ That can only see the output of the data structure, and not the random bits used by the data structure internally.

beyond high-probability correctness are needed. Indeed, the query interface of $\mathcal{D}$ does not leak any information about the potential random choices made by $\mathcal{D}$ internally (unless $\mathcal{D}$ errs, which we treat as a low-probability failure anyway) since the correct answers are uniquely determined by the graph maintained in $\mathcal{D}$.

Sankowski [20] showed a Monte Carlo randomized data structure with $U(n) = O(n^{1+\rho})$ and $Q(n) = O(n^\rho)$ (both worst-case), where $\rho < 0.529$ is such that $1 + 2\rho = \omega(1, \rho, 1)$. As a result, Theorem 1 implies a data structure with $O(n^{1.529})$ worst-case update bound for fully dynamic SCCs. To the best of our knowledge, we are the first to achieve a non-trivial $O(n^{2-\epsilon})$ *worst-case* update time for SCCs and even #SCC. Moreover, via a folklore⁶ reduction from fully dynamic single-source reachability and the conditional lower bound of [26], the obtained $O(n^{1.529})$ bound is tight for SCCs. Hence, our data structure constitutes the final answer to the problem (up to polylog factors), unless the *Mv-hinted Mv* variant of the OMv conjecture of [26, Conjecture 5.7] fails. That being said, it is not clear whether our bound is tight for the counting variant #SCC whose output is a single number, as opposed to n bits in SSR.

The worst-case update time guarantee allows applying Theorem 1 in the fault-tolerant model, where the goal is to recompute a graph property for a query set of at most k failed edges or vertices. Georgiadis, Italiano, and Parotsidis [12] showed that after optimal linear preprocessing, one can compute the SCCs of G after removing any single vertex/edge in optimal $O(n)$ time. Baswana, Choudhary, and Roditty [2] showed that after polynomial preprocessing, one can find the SCCs of G after removing k vertices/edges in $\tilde{O}(2^k \cdot n)$ time. Thus, the known results could only handle at most $\log n$ failures faster than recompute-from-scratch, for any density of G . Our result implies that for $m = \Omega(n^{1.53})$ one can recompute SCCs faster than from scratch even under *polynomially* many (that is, $k = O(m/n^{1.529})$) vertex/edge failures⁷. That being said, the purely combinatorial data structures [2, 12] are significantly faster if, e.g., k is constant.

It is worth noting that a reduction of computing SCCs to a reachability problem has been known in the parallel setting [21]. That static reduction does not seem to be applicable in the dynamic setting using algebraic techniques, though. More specifically, Schudy [21] reduces computing SCCs of G to a number of adaptively generated instances of *multi-source reachability* (which in turn reduces to the single-source case by adding a super-source) on induced subgraphs of G of total size $\tilde{O}(m)$. In the dynamic setting, spending time linear in m per update is prohibitive, and furthermore it is not clear how to efficiently simulate adaptively generated multi-source reachability queries for the following reasons. First, using the super-source trick would either require performing possibly $\Omega(n)$ edge updates (which are very costly, i.e., super-linear using algebraic methods) in the reachability data structure. If we wanted to avoid that, i.e., handle each source separately, we would possibly end up performing $\Omega(n^2)$ single-pair reachability queries. Our reduction for dynamic SCCs relies on being able to query $\tilde{O}(n)$ arbitrary reachability pairs after each update. Crucially, these pairs cannot be grouped into, e.g., $\text{polylog}(n)$ multi-source queries. Such an n -arbitrary-pairs reachability problem looks much more difficult even in the static setting and does not seem to be solvable in linear time like multi-source reachability. In fact, we believe no static bound beyond $O(\min(nm, n^\omega))$ is known.

⁶ It is enough to maintain a graph G' obtained from G by adding a super-sink t and edges vt for all $v \in V$. Then for a query source s , vertices reachable from s in G are those in the SCC of s in $G' + ts$.

⁷ Vertex failures can be easily simulated by splitting each $v \in V$ into $v_{\text{in}}, v_{\text{out}}$, that inherit the incoming and outgoing edges of v , resp, and introducing an edge $e_v = v_{\text{in}}v_{\text{out}}$. Then the failure of v is equivalent to the failure of e_v .

Strong connectivity. For the less general dynamic strong connectivity (SC), we give an even more efficient and *deterministic* black-box reduction to dynamic arbitrary-pair reachability.

► **Theorem 2.** *Let G be a digraph. Suppose there is a fully dynamic reachability data structure processing single-edge updates and arbitrary-pair reachability queries on G in at most $T(n)$ time. Then one can maintain whether G is strongly connected subject to single-edge insertions and deletions in $O(T(n) \log n)$ time per update (worst-case if the $T(n)$ bound is worst-case).*

van den Brand, Nanongkai, and Saranurak [26] showed a Monte Carlo randomized fully dynamic arbitrary-pair reachability data structure with $O(n^{1.407})$ worst-case update and query time. Therefore, Theorem 2 implies $O(n^{1.407})$ worst-case update bound for fully dynamic strong connectivity. This improves upon the known $O(n^{1.529})$ bound following from the trivial reduction to dynamic single-source reachability. By the conditional lower bound of [26], we essentially close the complexity of fully dynamic strong connectivity (up to polylogarithmic factors), unless the uMv-hinted uMv conjecture of [26, Conjecture 5.12] fails.

Whereas the obtained $O(n^{1.407})$ update bound is Monte Carlo randomized, this is only caused by the Schwartz-Zippel-lemma-style randomization inside the used reachability data structure. Obtaining a deterministic subquadratic fully dynamic reachability data structure would immediately imply a deterministic subquadratic bound for SC.

Interestingly, our reduction *does not* seem to generalize to maintaining whether G has more than two SCCs (SC2), which was the case for the trivial reduction. As a result, $O(n^{1.529})$ currently remains the sharpest update bound for SC2. At the same time the tightest conditional lower bound we have for SC2 is $\Omega(n^{1.407})$ because SC2 does not seem to be easily reducible from dynamic single-source reachability, which was the case for dynamic SCCs. This is a consequence of the fact that the output in the SCC problem is n numbers, whereas in SC2 it is just a single bit. Obtaining a faster dynamic algorithm for SC2 or coming up with a plausible lower bound beyond $O(n^{1.407})$ is an interesting next step.

■ **Table 1** Comparison of state-of-the art conditional lower- and upper bounds for the problems SC, SC2, #SCC, and SCC, and new upper bounds obtained via our reductions. Each problem in the table generalizes the preceding one. For SC and SCC, our obtained upper bounds are tight.

problem	known conditional lower bound	prev. worst-case update bound	previous amortized update bound	our new worst-case update bound
SC	$\Omega(n^{1.406})$ [1]+[26]	$O(n^{1.529})$ [26, 20]	same as worst-case	$O(n^{1.407})$ Theorem 2+[26]
SC2				—
#SCC	$\Omega(n^{1.528})$ [26] via SSR	$O(n^2)$ trivial	$O(n^{1.529})$ [17]	$O(n^{1.529})$ Theorem 1+[20]
SCC				

Generalizations. The reduction of Theorem 2 does generalize in two different ways. First, within the same update bound (see Theorem 11) we can actually maintain the exact size $\chi(G)$ of the *minimum strong connectivity augmentation* [10] of G . The minimum strong connectivity augmentation is a smallest possible set of edges that one needs to add to G to make it strongly connected. $\chi(G)$ can be seen as an alternative and “orthogonal” (to the number of SCCs) measure of how much G is *not* strongly connected. As shown by Eswaran and Tarjan [10], in the static setting, the value $\chi(G)$ and some minimum strong connectivity augmentation can be computed in linear time.

As is typical for algebraic data structures⁸, maintaining an *object* – some minimum strong augmentation in our case – is trickier than maintaining only the optimal *value* $\chi(G)$. Our algorithm can be nevertheless extended to maintain some $\chi(G)$ -sized augmentation Y within the same $O(n^{1.407})$ worst-case bound and against an adaptive adversary, as long as $\chi(G) = O(n^{0.703})$. On the other hand, for larger $\chi(G)$, e.g., $\chi(G) = \Theta(n)$, some minimum augmentation Y can be maintained in $O(n^{1.529})$ worst-case time per update.

Interestingly, the weighted version of the minimum strong connectivity augmentation problem, where the costs of different potential edges to be added vary, is NP-hard [10].

Moreover, the reduction behind Theorem 2 enables *universal* reachability queries, even if G is not strongly connected. That is, in $O(n^{1.407})$ time, we can decide whether a query vertex v can reach all vertices in G , or find some vertex that v cannot reach. See Theorem 14. The $O(n^{1.529})$ query time follows trivially (simply issue all the n possible reachability queries from v) from the fully dynamic single-source reachability data structure [20].

1.2 Further related work

Since the known lower bounds [1, 14] imply that fully dynamic (or partially dynamic with worst-case bounds) SCCs data structures are possible only for sufficiently dense graphs and using algebraic methods, most of the previous work regarding maintaining SCCs dealt with partially dynamic settings with the goal of optimizing total update time.

Maintaining SCCs has been studied in the incremental setting [5, 3, 13]. The current best known randomized solution for sparse graphs [5] has $\tilde{O}(m^{4/3})$ total update time, whereas the best known deterministic data structure [3] has $O(m^{3/2})$ total update time. For denser graphs, Bender et al. [3] gave a near-optimal deterministic data structure with $\tilde{O}(n^2)$ total update time. All these data structures maintain SCCs explicitly.

Decremental SCCs maintenance has been studied even more extensively [6, 7, 8, 18, 19]. Bernstein, Probst Gutenberg, and Wulff-Nilsen [7] gave a randomized data structure with near-optimal $\tilde{O}(m)$ total update time. The current best known deterministic total update time bound of $\tilde{O}(mn^{2/3})$ for general digraphs is due to Bernstein, Probst Gutenberg, and Saranurak [6]. For planar graphs, the near-optimal $\tilde{O}(n)$ total update time can be achieved deterministically [16].

In the case of (global) strong connectivity, incremental graph search from/to an arbitrary single source yields optimal $O(m)$ total update time. It is not clear if decremental strong connectivity is easier, in any sense, than decremental SCCs, and no specialized data structures beyond those for SCCs maintenance [7, 6] are known for decremental strong connectivity.

2 Preliminaries

In this paper we deal with *directed* graphs. We write $V(G)$ and $E(G)$ to denote the sets of vertices and edges of G , respectively. We omit G when the graph in consideration is clear from the context. A graph H is a *subgraph* of G , which we denote by $H \subseteq G$, if and only if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. We write $e = uv \in E(G)$ when referring to edges of G . By G^R we denote G with edges reversed.

A sequence of vertices $P = v_1 \dots v_k$, where $k \geq 1$ is called an $s \rightarrow t$ path in G if $s = v_1$, $v_k = t$ and there is an edge $v_i v_{i+1}$ in G for each $i = 1, \dots, k-1$. We sometimes view a path P as a subgraph of G with vertices $\{v_1, \dots, v_k\}$ and (possibly zero) edges $\{v_1 v_2, \dots, v_{k-1} v_k\}$.

⁸ For example, for dynamic reachability and exact distances, the known algebraic data structures [25, 26, 20] are polynomially faster than the known solutions to their respective path-reporting variants [4, 17].

For convenience, we sometimes consider a single edge uv a path. If P_1 is a $u \rightarrow v$ path and P_2 is a $v \rightarrow w$ path, we denote by $P_1 \cdot P_2$ (or simply $P_1 P_2$) a path obtained by concatenating P_1 with P_2 . A vertex $t \in V(G)$ is *reachable* from $s \in V(G)$ if there is an $s \rightarrow t$ path in G .

3 Fully dynamic strongly connected components

In this section we describe our fully dynamic strongly connected components data structure.

Suppose there exists a reachability data structure maintaining an n -vertex digraph subject to single-edge insertions and deletions and answering arbitrary-pair reachability queries. Denote by $U(n)$ and $Q(n)$ the update and query (resp.) time bounds of the assumed data structure.

We start by proving the following key observation, that tracking the vertices $v \in V$ strongly connected to *some* vertex t out of a chosen subset $T \subseteq V$ can be reduced to maintaining some n cells of the transitive closure matrix (reachability pairs) of a certain auxiliary graph G_T .

► **Lemma 3.** *Let $G = (V, E)$ be a digraph, and let $T \subseteq V$. Let G' be a graph with vertices $V' = \{v' : v \in V\}$, and edges $E' = \{u'v' : uv \in E\}$, i.e., G' is a copy of G with vertices primed. Consider a graph G_T obtained from $G \cup G'$ by adding edges tt' for each $t \in T$. Then, for any $v \in V$, v is strongly connected with some $t \in T$ in G if and only if there is a path $v \rightarrow v'$ in G_T .*

Proof. Suppose that v is strongly connected with some $t \in T$ in G . Then, there exists a path $P = v \rightarrow t$ in G , and a path $Q = t \rightarrow v$ in G . As a result, there is a corresponding path $Q' = t' \rightarrow v'$ in G' . Since $tt' \in E(G_T)$, we conclude there is a $v \rightarrow v'$ path $P \cdot tt' \cdot Q'$ in G_T .

Now suppose there is a $v \rightarrow v'$ path in G_T . Note that a path R starting in the G -part of G_T can only depart from G for G' through an edge tt' , $t \in T$, and it cannot go back to G . As a result, R can be expressed as $R_1 \cdot (tt') \cdot R'_2$, where $R_1 = v \rightarrow t$ is a path in G , and R'_2 is a $t' \rightarrow v'$ path in G' . But R'_2 has a corresponding path $R_2 = t \rightarrow v$ in G . The paths R_1, R_2 certify that v and t are strongly connected. Consequently, v is strongly connected to some vertex of T in G . ◀

To make use of Lemma 3 we will employ the *unique witness trick* (usually attributed to [22]) that have been used for computing witnesses of various matrix products, also in the dynamic setting [4]. However, in our case, we will use this technique with a conceptually very different purpose: to *select* a certain *root* of every SCC that ever appears in G throughout edge updates.

► **Lemma 4.** [22] *Let $X \subseteq V$ be a subset with $n/2^{k+1} \leq |X| \leq n/2^k$ for an integer $k \geq 0$. Let $S \subseteq V$ be a subset obtained by sampling 2^k elements of V uniformly at random and independently (with replacement). Then, with probability at least $1/6$, $|X \cap S| = 1$.*

Proof. The probability that only the i -th sampled vertex is a unique element of $X \cap S$ is

$$\frac{|X|}{n} \cdot \left(1 - \frac{|X|}{n}\right)^{2^k-1} \geq \frac{1}{2^{k+1}} \cdot \left(1 - \frac{1}{2^k}\right)^{2^k-1} \geq \frac{1}{2^{k+1}e}.$$

The probability that $|X \cap S| = 1$ is at least the probability of one of the above disjoint events happening, i.e., at least $2^k \cdot \frac{1}{2^{k+1}e} = \frac{1}{2e} \geq 1/6$. ◀

Let $\ell = (\gamma + 2) \cdot \log_{6/5} n$ for a constant $\gamma > 0$ that can be adjusted to control the error probability. Let B be the smallest integer such that $n < 2^B$, so that $B = O(\log n)$. For a non-negative integer m , put $[m] := \{0, \dots, m\}$. Let us identify V with the B -bit numbers $\{1, \dots, n\}$. Moreover, let

$$\mathcal{T} = \{T_{k,l,b} : k \in [B-1], l \in [\ell-1], b \in [B-1]\}$$

be a family of subsets of V obtained as follows. For $k \in [B-1], l \in [\ell-1]$ let $T_{k,l}$ be a subset of V obtained by sampling 2^k elements of V uniformly at random and independently, with replacement. Then, set $T_{k,l,b}$ to be a subset of numbers $v \in T_{k,l}$ such that the b -th (0-based) bit of v in the binary equals 1. Clearly, the family $\mathcal{T}$ has $O(\log^3 n)$ subsets.

First of all, we maintain the graph G itself in the assumed fully dynamic reachability data structure $\mathcal{D}$. For each $T \in \mathcal{T}$, we set up another assumed data structure $\mathcal{D}_T$ maintaining the graph G_T , as defined in Lemma 3. Note that each edge update to the graph G is reflected using just two edge updates to each graph G_T since G_T consists of two copies of G . As a result, every edge update can be processed in $O(U(n) \cdot |\mathcal{T}|) = \tilde{O}(U(n))$ time (worst-case if the bound $U(n)$ is worst-case).

We now describe how the SCCs of G are recomputed after an update. We build an auxiliary graph H whose construction follows. For each $v \in V$, $k \in [B-1]$, and $l \in [\ell-1]$, we do the following. For every $b \in [B-1]$, we query the data structure $\mathcal{D}_{T_{k,l,b}}$ whether there exists a path $v \rightarrow v'$ in $G_{T_{k,l,b}}$. Let $s_{v,k,l}$ be a number (vertex) such that the b -th bit is set to 1 if and only if the corresponding path in $G_{T_{k,l,b}}$ exists. If $s_{v,k,l} \in \{1, \dots, n\}$ (so that $s_{v,k,l}$ is indeed a vertex of G) and v is strongly connected with $s_{v,k,l}$ (which can be tested using two queries to $\mathcal{D}$), we add an undirected edge $\{v, s_{v,k,l}\}$ to H .

Observe that the graph H is constructed by performing $\tilde{O}(n)$ queries to the assumed data structures $\mathcal{D}$ and $\mathcal{D}_T$, $T \in \mathcal{T}$. The cost of these queries is $\tilde{O}(n \cdot Q(n))$ (worst-case if $Q(n)$ is worst-case).

Having constructed the undirected graph H , we output its connected components as the SCCs of G . Since H has $\tilde{O}(n)$ edges, this can be done in $\tilde{O}(n)$ additional worst-case time.

► **Lemma 5.** *With probability at least $1 - 1/n^\gamma$, the connected components of H and the strongly connected components of G are equal.*

Proof. By the construction, if two vertices $u, v \in V$ are connected by an edge (or more generally, a path) in H , then they are strongly connected in G . Hence, we only need to prove that if $u, v \in V$ are strongly connected in G , then they are connected in H as well.

Let C be the SCC of G containing both u and v . Let $z \in [B-1]$ be such an integer that we have $n/2^{z+1} \leq |C| \leq n/2^z$. By Lemma 4, for each $l \in [\ell-1]$, $T_{z,l}$ satisfies $|T_{z,l} \cap C| = 1$ with probability at least $1/6$. As a result, with probability at least $1 - (5/6)^\ell \geq 1 - 1/n^{\gamma+2}$, at least one of the sets $T_{z,l}$, say $T_{z,y}$, satisfies $|T_{z,y} \cap C| = 1$. Let s be the unique element of $T_{z,y} \cap C$. Observe that for $b \in [B-1]$, we have $T_{z,y,b} \cap C = \{s\}$ if the b -th bit of s equals 1 and $T_{z,y,b} \cap C = \emptyset$ otherwise. As a result, by Lemma 3, for both u and v the query to the data structure $\mathcal{D}_{T_{z,y,b}}$ will return true if and only if the b -th bit of s is 1. Equivalently, we will have $s = s_{v,k,l} = s_{u,k,l} \neq 0$. Since both edges $\{v, s\}$ and $\{u, s\}$ are added to H , u and v are indeed connected in H .

As there are at most n^2 pairs (u, v) , the implications hold with probability at least $1 - n^2 \cdot (1/n^{\gamma+2}) = 1 - 1/n^\gamma$. ◀

Note that the above lemma holds for any version of G since, unless the algorithm makes a mistake (which happens with low probability), we do not reveal any random bits to the adversary as the output is always unique. To summarize, we obtain the following.

► **Theorem 1.** *Let G be a digraph. Suppose there is a fully dynamic reachability data structure $\mathcal{D}$ processing single-edge updates and arbitrary-pair queries on G in $U(n)$ and $Q(n)$ time respectively. Suppose the answers produced by $\mathcal{D}$ are correct with high probability.*

Then one can explicitly maintain the SCCs of G subject to single-edge insertions and deletions in $\tilde{O}(U(n) + n \cdot Q(n))$ time per update (worst-case if the bounds $U(n), Q(n)$ are worst-case). The obtained data structure is Monte Carlo randomized. The answers produced are correct with high probability.

Sankowski [20] showed that there exists a fully dynamic reachability data structure with $U(n) = \tilde{O}(n^{1+\rho})$ worst-case update time and $Q(n) = \tilde{O}(n^\rho)$ query time, where $\rho < 0.529$ is such that $1 + 2\rho = \omega(1, \rho, 1)$. By Theorem 1 it follows that there exists a fully dynamic data structure maintaining SCCs explicitly with $\tilde{O}(n^{1+\rho}) = O(n^{1.529})$ worst-case update time.

4 Reducing dynamic strong connectivity to dynamic reachability

In this section we show how to maintain whether the graph G is strongly connected subject to edge insertions and deletions using a fully dynamic reachability data structure supporting single-edge updates and arbitrary-pair reachability queries. Our reduction is deterministic and incurs only polylogarithmic overhead. Again, assume that $n < 2^B$ for an integer $B = O(\log n)$. The vertices are numbered 1 through n and thus can be seen as B -bit numbers.

Let us define a *source strongly connected component* of G to be an SCC of G with no incoming edges from other SCCs of G . Clearly, if G is strongly connected, it has precisely one source SCC. The main idea is to maintain a *source set* $\mathcal{Z} \subseteq V$ such that $\mathcal{Z}$ contains *precisely one* vertex z of every source SCC of G . For example, if G is strongly connected, then $\mathcal{Z}$ can be any single-element subset of V . We also have the following simple property.

► **Observation 6.** *For every vertex $v \in V(G)$ there exists some source SCC S of G such that v is reachable from every $s \in S$.*

Proof. Let C be the SCC of v in G . Let G^* be the *condensation* of G obtained from G by contracting the SCCs. Then, G^* is a DAG. In a DAG, every vertex is reachable from a source (that is, a vertex with no incoming edges). In particular, C is reachable from some source S of G^* . Consequently, any vertex $s \in S$ can reach v in G by construction of G^* . ◀

The following lemma shows how a data structure maintaining some source set of G can be used to maintain strong connectivity of G .

► **Lemma 7.** *Let G^R be the reverse of G . Let $\mathcal{Z}$ be some source set of G and let $\mathcal{Z}^R$ be some source set of G^R . Then G is strongly connected if and only if $|\mathcal{Z}| = |\mathcal{Z}^R| = 1$ and $t \in \mathcal{Z}^R$ can reach $s \in \mathcal{Z}$ in G .*

Proof. Clearly, if G is strongly connected, then $\mathcal{Z} = \{s\}$, $\mathcal{Z}^R = \{t\}$ for some $s, t \in V$. Moreover, t can reach s in G .

Now suppose $\mathcal{Z} = \{s\}$ and $\mathcal{Z}^R = \{t\}$. Consider any $x, y \in V$. By Observation 6, y is reachable from s . Similarly, by Observation 6 applied to G^R , x is reachable from t in G^R , i.e., t is reachable from x in G . By the assumption, t can reach s in G . Thus, there exist a path $x \rightarrow t \rightarrow s \rightarrow y$ in G . Since the pair x, y was arbitrary, G is indeed strongly connected. ◀

By Lemma 7, we can maintain strong connectivity using a fully dynamic source set data structure built on G , a fully dynamic source set data structure built on G^R , and a fully dynamic reachability data structure built on G . Every edge update translates to a single edge update in the three data structures, and a single query to the reachability data structure.

Let us now focus on maintaining a source set $\mathcal{Z}$ of G subject to edge updates. We have:

► **Lemma 8.** *For any $w \in V$, there exists at most one $z \in \mathcal{Z}$ such that w can reach z in G . Moreover, w is in a source SCC (containing z) if and only if such a $z \in \mathcal{Z}$ exists.*

Proof. Let $z \in \mathcal{Z}$ and let C be the source SCC of G with $z \in C$. By the definition of a source SCC, the only vertices of V that can reach z are those in C . As a result, if w can reach z , then $w \in C$. Therefore, $w \notin C'$ for other source SCCs $C' \neq C$, and thus w cannot reach any other $z' \in \mathcal{Z}$, $z' \neq z$. On the other hand, if $w \in C$ for some source SCC C , then w can clearly reach the unique element of $\mathcal{Z} \cap C$. ◀

Maintained data structures. First of all, let $\bar{G}$ be the graph G extended with a super source $s \notin V(G)$ and edges sz for all $z \in \mathcal{Z}$. We store $\bar{G}$ in a fully dynamic reachability data structure $\mathcal{D}$. Note that using a single reachability query to $\mathcal{D}$ we can check whether some query vertex $v \in V$ is reachable from $\mathcal{Z}$ in $\bar{G}$. Since $G \subseteq \bar{G}$ and s has no incoming edges, the data structure $\mathcal{D}$ can also handle usual reachability queries in G .

For $b \in [B-1]$, let G_b be the graph G extended with a super sink $t \notin V(G)$, and edges zt for all $z \in \mathcal{Z}$ such that the b -th bit of the number z equals 1. We store the $O(\log n)$ graphs G_b in fully dynamic reachability data structures $\mathcal{D}_0, \dots, \mathcal{D}_{B-1}$. Note that since every $v \in V$ has at least one bit set, $\bigcup_{i=0}^{B-1} G_b$ contains edges zt for all $z \in \mathcal{Z}$. As a result, a vertex $w \in V$ can reach some vertex of $\mathcal{Z}$ if and only if w can reach t in $\bigcup_{i=0}^{B-1} G_b$. Moreover, by Lemma 8, if w can reach some vertex of $\mathcal{Z}$, then it can reach precisely one such $z \in \mathcal{Z}$. The individual data structures G_b can be used to find that z : observe that w can reach t (equivalently, reach z) in G_b precisely if and only if z has the b -th bit set. As a result, we have the following.

► **Lemma 9.** *For any $w \in V$ one can find the unique $z \in \mathcal{Z}$ that w can reach in G (if one exists) using $O(\log n)$ queries to the fully dynamic reachability data structures $\mathcal{D}_0, \dots, \mathcal{D}_{B-1}$.*

Whenever the graph G is updated or the set $\mathcal{Z}$ undergoes an update, the update is passed to all the $O(\log n)$ relevant data structures $\mathcal{D}$ and $\mathcal{D}_0, \dots, \mathcal{D}_{B-1}$ so that they reflect the current graphs $\bar{G}$ and $G_0, \dots, G_{B-1}$ respectively.

We now proceed with describing how the updates are processed. In the following, denote by G' the graph G after the edge update, whereas G denotes the graph before the edge update considered.

Edge insertions. First consider an insertion of edge uv . If there exists a path $u \rightarrow v$ in G (a single query to $\mathcal{D}$), the SCCs of G do not change due to the insertion. No source SCC gets an incoming edge from another SCC. Therefore, $\mathcal{Z}$ remains a valid source set.

Let us thus assume that there is no $u \rightarrow v$ path in G . In particular, u and v are not strongly connected in G . Let us denote by S_v the SCC containing v in G . Inserting uv cannot break any of the existing SCCs of G , but might cause some SCCs of G (in particular S_v) merge in G' . Let S^* denote the SCC of v in G' . We have $S_v \subseteq S^*$, but potentially $S^* = S_v$.

Let us now argue that S_v is the only SCC of G that might lose the status of a source SCC due to the insertion. Indeed, if $S_v \neq S^*$, then every other SCC $S' \neq S_v$ of G , such that $S' \subset S^*$, is reachable from S_v since the insertion of uv makes S_v and S' strongly connected. As a result, S' is not a source SCC of G . Moreover, the respective sets of incoming inter-SCC edges of other SCCs S'' of G such that $S'' \cap S^* = \emptyset$ do not change, so such S'' is a source SCC of G' if and only if S'' is a source SCC of G .

By Lemmas 8 and 9 we can test whether S_v is a source SCC (and possibly find the unique $z \in S_v \cap \mathcal{Z}$) by performing $O(\log n)$ queries to the data structures $\mathcal{D}_0, \dots, \mathcal{D}_{B-1}$. The next step is to remove z (if it exists) from $\mathcal{Z}$, and update all the data structures $\mathcal{D}, \mathcal{D}_0, \dots, \mathcal{D}_b$ accordingly. Next, we apply the insertion of uv to all these data structures so that they store

(supergraphs) of the updated graph G' . Observe that at this point the set $\mathcal{Z}$ might miss at most one element, if S^* turns out to be a source SCCs of G' . In order to detect whether this is the case we check whether $v \in S^*$ can be reached from $\mathcal{Z}$ in G' . Recall that this amounts to a single query to $\mathcal{D}$ about the existence of a path from s to u in $\bar{G}$. If so, S^* is reachable from some other SCC, and thus is not a source SCC. Otherwise, S^* is indeed a source SCC and thus we add to $\mathcal{Z}$ an arbitrary element of S^* , e.g., the vertex v , so that $\mathcal{Z}$ again becomes a valid source set of G' .

Edge deletions. Edge deletions can be handled essentially symmetrically to edge insertions. Suppose an edge $uv \in E(G)$ is deleted. Again, if the deletion does not make v unreachable from u (which can be tested by performing a single update and query on $\mathcal{D}$), neither the SCCs of G nor the source SCCs of G change. So suppose u cannot reach v after the deletion.

Let S'_v be the SCC containing v in G' and let S^* be the SCC containing v in G . We have $S'_v \subseteq S^*$ and potentially $S'_v = S^*$. If S^* is a source SCC in G , we can find $z \in S^* \cap \mathcal{Z}$ using $O(\log n)$ reachability queries from v to t in $\mathcal{D}_0, \dots, \mathcal{D}_{B-1}$ by Lemmas 8 and 9.

Observe that S'_v is the only SCC contained in S^* that might potentially be a source SCC of G' : if $S' \neq S'_v$ is an SCC of G' , and $S' \subset S^*$, then S' is surely reachable from S'_v in G' and thus S' is not a source SCC of G' . Similarly as we did when processing an insertion, we remove z (if exists) from $\mathcal{Z}$. At this point, $\mathcal{Z}$ might need inserting at most one element in order to become a valid source set of G' . This is the case precisely when S'_v is a source SCC of G' . We can check that, again, by issuing a single $s \rightarrow v$ reachability query to $\mathcal{D}$ after the deletion is reflected in $\mathcal{D}$. If S'_v happens to be a source SCC of G' , we add $v \in S'_v$ to $\mathcal{Z}$.

Note that processing an edge update causes at most two element updates to the set $\mathcal{Z}$. Each of these updates, and also the edge update itself, is reflected in $O(\log n)$ fully dynamic reachability data structures $\mathcal{D}, \mathcal{D}_0, \dots, \mathcal{D}_{B-1}$. Moreover, only $O(\log n)$ reachability queries are performed on these data structures. Consequently, we obtain the following.

► **Lemma 10.** *Suppose there is a fully dynamic reachability data structure processing single-edge updates and arbitrary-pair reachability queries on G in at most $T(n)$ time. Then one can maintain a source set $\mathcal{Z}$ of G explicitly subject to single-edge insertions and deletions in $O(T(n) \log n)$ time per update (worst-case if the $T(n)$ bound is worst-case).*

By Lemma 10 and the discussion after Lemma 7, we obtain:

► **Theorem 2.** *Let G be a digraph. Suppose there is a fully dynamic reachability data structure processing single-edge updates and arbitrary-pair reachability queries on G in at most $T(n)$ time. Then one can maintain whether G is strongly connected subject to single-edge insertions and deletions in $O(T(n) \log n)$ time per update (worst-case if the $T(n)$ bound is worst-case).*

The currently best known bound on the maximum of query time and update time of a fully dynamic arbitrary-pair reachability data structure is $T(n) = O(n^{1.407})$ worst-case (Monte Carlo randomized) due to [26]. Consequently, Theorem 2 combined with [26] yields $O(n^{1.407})$ worst-case update time for fully dynamic strong connectivity.

Strong connectivity augmentation. Due to a result of Eswaran and Tarjan [10], using Lemma 10 we can actually prove a more general result.

► **Theorem 11.** *Let G be a digraph. Let $T(n)$ be defined as in Theorem 2. Then one can maintain the size $\chi(G)$ of a minimum strong connectivity augmentation of G subject to single-edge insertions and deletions in $O(T(n) \log n)$ time per update. (worst-case if the $T(n)$ bound is worst-case).*

Proof. Let $\mathcal{Z}$ and $\mathcal{Z}^R$ be source sets of G and G^R respectively. Define a *sink SCC* to be an SCC of G that is a source SCC in G^R . Eswaran and Tarjan [10] prove that if G is not strongly connected, then $\chi(G)$ equals $\max(s, t) + q$, where s denotes the number of source SCCs that are not sink SCCs, t denotes the number of sink SCCs that are not source SCCs, and q denotes the number of “isolated SCCs” that are both source and sink SCCs. If G is strongly connected then $\chi(G) = 0$ holds trivially.

Note that with s, t, q defined like this, we have $|\mathcal{Z}| = s + q$ and $|\mathcal{Z}^R| = t + q$. Hence, if G is not strongly connected, then $\max(|\mathcal{Z}|, |\mathcal{Z}^R|) = \max(s, t) + q = \chi(G)$. As a result, it is enough to maintain some sets $\mathcal{Z}$ and $\mathcal{Z}^R$, and test whether G is strongly connected using Lemmas 10 and 7. Clearly, we need only $O(T(n) \log n)$ time for this. ◀

Again, by combining Theorem 11 with [26], we obtain that the size $\chi(G)$ of a minimum strong connectivity augmentation can be maintained in $O(n^{1.407})$ worst-case time per update.

Theorem 11 does not immediately yield any actual set Y of edges such that $|Y| = \chi(G)$ and $G + Y$ is strongly connected. We now discuss how such an augmentation Y can be maintained. To this end we now sketch the method [10] of obtaining an augmentation with $\max(|\mathcal{Z}|, |\mathcal{Z}^R|)$ edges in case G is not strongly connected. Observe that in such a case, the quantity $\max(|\mathcal{Z}|, |\mathcal{Z}^R|)$ is clearly a lower bound on the size of Y .

Suppose wlog. that $|\mathcal{Z}| \geq |\mathcal{Z}^R|$. For each $z \in \mathcal{Z}$, let $t_z \in \mathcal{Z}^R$ be arbitrary such that z can reach t_z in G ; note that at least one such t_z always exists. Similarly, for each $z' \in \mathcal{Z}^R$, let $s_{z'} \in \mathcal{Z}$ be arbitrary such that z' is reachable from $s_{z'}$ in G .

Consider a set of edges $F = \{zt_z : z \in \mathcal{Z}\} \cup \{s_{z'}z' : z' \in \mathcal{Z}^R\}$. Let M be a maximal matching in F (possibly allowing self-loops), that is, a maximal subset $\{y_1y'_1, \dots, y_ky'_k\} \subseteq F$ such that all $y_1, \dots, y_k$ are distinct and all $y'_1, \dots, y'_k$ are distinct.

Put $A := \{y_1, \dots, y_k\}$, and $B = \{y'_1, \dots, y'_k\}$. Moreover, put $\mathcal{Z} \setminus A = \{z_1, \dots, z_p\}$ and $\mathcal{Z}^R \setminus B = \{z'_1, \dots, z'_q\}$, where $p \geq q$. By the maximality of M , we have that for all $z \in \mathcal{Z} \setminus A$, $t_z \in B$, that is, z can reach some vertex from B in G . Similarly, for all $z' \in \mathcal{Z}^R \setminus B$, z' is reachable from a vertex of A .

Put $y_{k+1} := y_1$. Consider an augmentation:

$$Y = \{y'_iy_{i+1} : i = 1, \dots, k\} \cup \{z'_{\max(i,q)}z_i : i = 1, \dots, p\}.$$

We have $|Y| = k + p = |\mathcal{Z}| = \max(|\mathcal{Z}|, |\mathcal{Z}^R|)$. Eswaran and Tarjan [10] argue rather easily that $G + Y$ is strongly connected: (1) the source and sink vertices in $A \cup B$ are pairwise strongly connected since they are arranged in a cycle, (2) every vertex in $\mathcal{Z}^R \setminus B$ can reach $A \cup B$ via $\mathcal{Z} \setminus A$ and the edges in Y , and (3) every vertex in $\mathcal{Z} \setminus A$ can be reached from $A \cup B$ via $\mathcal{Z}^R \setminus B$ and the edges in Y .

The above construction reduces, in $O(n)$ time, the problem of maintaining an optimal Y to maintaining:

- (1) for each vertex $z \in \mathcal{Z}$, an arbitrary vertex $t_z \in \mathcal{Z}^R$ reachable from z in G ,
- (2) for each vertex $z' \in \mathcal{Z}^R$, an arbitrary vertex $s_{z'} \in \mathcal{Z}$ that can reach z' in G .

If $\chi(G) = \max(|\mathcal{Z}|, |\mathcal{Z}^R|)$ is small enough, we can achieve the above by simply maintaining the $(\mathcal{Z} \cup \mathcal{Z}^R) \times (\mathcal{Z} \cup \mathcal{Z}^R)$ submatrix of the transitive closure of G . van den Brand, Forster, and Nazari [25] showed the following.

► **Theorem 12.** [25] *Let G be a directed graph. Let S be a dynamic subset of V be such that $|S| = O(n^{0.85})$ at all times. There exists a data structure explicitly maintaining reachability between all-pairs of vertices in S subject to fully dynamic single-edge updates to G and single-element additions/deletions to S in $O(n^{1.407} + |S|^2)$ worst-case time per update.*

Recall that if G is subject to fully dynamic single-edge updates, then $\mathcal{Z}$ and $\mathcal{Z}^R$ undergo at most two element updates per update to G . As a result, by putting $S = \mathcal{Z} \cup \mathcal{Z}^R$ in the above theorem and combining with Lemma 10, we can maintain a minimum strong connectivity augmentation Y in $O(n^{1.407})$ time as long as $\chi(G)^2 = O(n^{1.407})$.

If $\chi(G) = \Theta(n)$, however, the above method would have quadratic update time. We now argue that if we expect $\chi(G)$ to be large, then some desired sets $\{t_z \in \mathcal{Z}^R : z \in \mathcal{Z}\}$, and $\{s_{z'} \in \mathcal{Z} : z' \in \mathcal{Z}^R\}$ can be maintained in $O(n^{1.529})$ worst-case time per update anyway. This is possible using the below simple *existential reachability* data structure.

► **Lemma 13.** *Let G be a directed graph. Suppose there is a fully dynamic reachability data structure processing single-edge updates and arbitrary-pair reachability queries on G in $U(n) = \text{poly}(n)$ and $Q(n) = \text{poly}(n)$ time respectively.*

Let $X \subseteq V$ be a dynamic subset. There is a data structure maintaining G subject to fully dynamic edge updates and X subject to single-element insertions and deletions with $\tilde{O}(U(n))$ update time supporting the following queries in $\tilde{O}(Q(n))$ -time. Given a query vertex $s \in V$, find some vertex $x \in X$ that s can reach in G (if such an x exists).

Proof sketch. Suppose wlog. that the vertices of G are identified with $\{1, \dots, n\}$ and n is a power of two. Let G' be obtained from G as follows. First, we augment G with an auxiliary full binary tree $\mathcal{T}$ with n leaves labeled $l_1, \dots, l_n$ from left to right, so that $V(\mathcal{T}) \cap V(G) = \emptyset$. The edges in $\mathcal{T}$ are directed from children to their parents. Second, for every $x \in X$, we add an edge xl_x to G' . Note that the graph G' has at most $3n$ vertices.

We maintain the graph G' using the assumed reachability data structure $\mathcal{D}$. Whenever G is subject to an edge insertion or deletion, that update is also applied to G' . Similarly, if an element x is inserted/deleted from X , this is reflected in G' by adding/deleting the edge xl_x .

The binary tree $\mathcal{T}$ allows locating some (or even the minimum-label) vertex of X that a query vertex s can reach within $O(\log n)$ queries to $\mathcal{D}$. Indeed, note that s can reach a vertex $w \in V(\mathcal{T})$ in G' if and only if s can reach in G some vertex $x \in X$ such that w is an ancestor of the leaf $l_x \in V(\mathcal{T})$. In particular, s can reach X in G if and only if s can reach the root of $\mathcal{T}$ in G' . If this is the case, the search starts in the root of $\mathcal{T}$ and descends down the tree maintaining the invariant that one of the leaves in the current subtree is reachable from s in G' . Once we reach a leaf l_x , we report $x \in X$ as reachable from s . ◀

With Lemma 13 in hand (applied twice, to the graph G with $X := \mathcal{Z}^R$ and to the graph G^R with set $X := \mathcal{Z}$), using fully dynamic reachability data structure of [20] with $O(n^{1.529})$ worst-case update time and $O(n^{0.529})$ query time, we can find the desired sets by issuing $O(n)$ existential reachability queries. Recall that the sets $\mathcal{Z}$ and $\mathcal{Z}^R$ undergo $O(1)$ updates per edge update to G , so the worst-case update time is $O(n^{1.529})$.

Finally, we note that the described algorithms for maintaining the minimum strong connectivity augmentation Y work against an adaptive adversary. Indeed, they are deterministic if provided with the required reachability information about G . That in turn is uniquely defined and maintained correctly with high probability by the data structures of [25, 20].

Universal reachability. By maintaining the source set $\mathcal{Z}$, we can also achieve the following.

► **Theorem 14.** *Let G be a digraph. Let $T(n)$ be defined as in Theorem 2. There exists a data structure maintaining G under single-edge insertions and deletions and answering queries whether a given vertex v can reach all vertices in G (and if not, providing an example of an unreachable vertex) with $\tilde{O}(T(n))$ update and query time.*

Proof. Let $\mathcal{Z}$ be again a source set of G . Clearly, if $|\mathcal{Z}| > 1$, then no vertex of G can reach all other vertices. Otherwise, if $|\mathcal{Z}| = 1$, then some v can reach all of V if and only if v can reach (or, equivalently, is strongly connected to) the only element $z \in \mathcal{Z}$. As a result, given $\mathcal{Z}$ and the data structure $\mathcal{D}$, we can check whether v can reach all vertices in G by issuing a $v \rightarrow z$ query to $\mathcal{D}$ in $\tilde{O}(T(n))$ time. If v cannot reach some vertex of G , then it cannot reach some $z' \in \mathcal{Z}$. Since there is at most one $z \in \mathcal{Z}$ that v can reach, to find some unreachable $z' \in \mathcal{Z}$, it is enough to issue two queries to $\mathcal{D}$. ◀

References

- 1 Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014*, pages 434–443. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.53.
- 2 Surender Baswana, Keerti Choudhary, and Liam Roditty. An efficient strongly connected components algorithm in the fault tolerant model. *Algorithmica*, 81(3):967–985, 2019. doi:10.1007/s00453-018-0452-3.
- 3 Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A new approach to incremental cycle detection and related problems. *ACM Trans. Algorithms*, 12(2):14:1–14:22, 2016. doi:10.1145/2756553.
- 4 Thiago Bergamaschi, Monika Henzinger, Maximilian Probst Gutenberg, Virginia Vassilevska Williams, and Nicole Wein. New techniques and fine-grained hardness for dynamic near-additive spanners. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*, pages 1836–1855. SIAM, 2021. doi:10.1137/1.9781611976465.110.
- 5 Aaron Bernstein, Aditi Dudeja, and Seth Pettie. Incremental SCC maintenance in sparse graphs. In *29th Annual European Symposium on Algorithms, ESA 2021*, volume 204 of *LIPICs*, pages 14:1–14:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ESA.2021.14.
- 6 Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental reachability, scc, and shortest paths via directed expanders and congestion balancing. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*, pages 1123–1134. IEEE, 2020. doi:10.1109/FOCS46700.2020.00108.
- 7 Aaron Bernstein, Maximilian Probst, and Christian Wulff-Nilsen. Decremental strongly-connected components and single-source reachability in near-linear time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, pages 365–376. ACM, 2019. doi:10.1145/3313276.3316335.
- 8 Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Jakub Łącki, and Nikos Parotsidis. Decremental single-source reachability and strongly connected components in $\tilde{O}(m\sqrt{n})$ total update time. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016*, pages 315–324. IEEE Computer Society, 2016. doi:10.1109/FOCS.2016.42.
- 9 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: <http://mitpress.mit.edu/books/introduction-algorithms>.
- 10 Kapali P. Eswaran and Robert Endre Tarjan. Augmentation problems. *SIAM J. Comput.*, 5(4):653–665, 1976. doi:10.1137/0205044.
- 11 Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Inf. Process. Lett.*, 74(3-4):107–114, 2000. doi:10.1016/S0020-0190(00)00051-X.
- 12 Loukas Georgiadis, Giuseppe F. Italiano, and Nikos Parotsidis. Strong connectivity in directed graphs under failures, with applications. *SIAM J. Comput.*, 49(5):865–926, 2020. doi:10.1137/19M1258530.

- 13 Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert Endre Tarjan. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Trans. Algorithms*, 8(1):3:1–3:33, 2012. doi:10.1145/2071379.2071382.
- 14 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015*, pages 21–30. ACM, 2015. doi:10.1145/2746539.2746609.
- 15 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001. doi:10.1006/jcss.2000.1727.
- 16 Giuseppe F. Italiano, Adam Karczmarz, Jakub Łącki, and Piotr Sankowski. Decremental single-source reachability in planar digraphs. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017*, pages 1108–1121. ACM, 2017. doi:10.1145/3055399.3055480.
- 17 Adam Karczmarz, Anish Mukherjee, and Piotr Sankowski. Subquadratic dynamic path reporting in directed graphs against an adaptive adversary. In *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1643–1656. ACM, 2022. doi:10.1145/3519935.3520058.
- 18 Jakub Łącki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Trans. Algorithms*, 9(3):27:1–27:15, 2013. doi:10.1145/2483699.2483707.
- 19 Liam Roditty and Uri Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.*, 37(5):1455–1471, 2008. doi:10.1137/060650271.
- 20 Piotr Sankowski. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *45th Symposium on Foundations of Computer Science FOCS 2004*, pages 509–517. IEEE Computer Society, 2004. doi:10.1109/FOCS.2004.25.
- 21 Warren Schudy. Finding strongly connected components in parallel using $o(\log^2 n)$ reachability queries. In *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 146–151. ACM, 2008. doi:10.1145/1378533.1378560.
- 22 Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51(3):400–403, 1995. doi:10.1006/jcss.1995.1078.
- 23 M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981. doi:10.1016/0898-1221(81)90008-0.
- 24 Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. doi:10.1137/0201010.
- 25 Jan van den Brand, Sebastian Forster, and Yasamin Nazari. Fast deterministic fully dynamic distance approximation. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022*, pages 1011–1022. IEEE, 2022. doi:10.1109/FOCS54457.2022.00099.
- 26 Jan van den Brand, Danupon Nanongkai, and Thatchaphol Saranurak. Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019*, pages 456–480. IEEE Computer Society, 2019. doi:10.1109/FOCS.2019.00036.
- 27 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2-3):357–365, 2005. doi:10.1016/j.tcs.2005.09.023.

Decreasing the maximum average degree by deleting an independent set or a d -degenerate subgraph*

Wojciech Nadara[†] Marcin Smulewicz[‡]

Institute of Informatics
University of Warsaw
Poland

`{w.nadara,m.smulewicz}@mimuw.edu.pl`

Submitted: Mar 21, 2020; Accepted: Dec 30, 2021; Published: Mar 11, 2022

© The authors. Released under the CC BY-ND license (International 4.0).

Abstract

The maximum average degree $\text{mad}(G)$ of a graph G is the maximum over all subgraphs of G , of the average degree of the subgraph. In this paper, we prove that for every G and positive integer k such that $\text{mad}(G) \geq k$ there exists $S \subseteq V(G)$ such that $\text{mad}(G - S) \leq \text{mad}(G) - k$ and $G[S]$ is $(k - 1)$ -degenerate. Moreover, such S can be computed in polynomial time. In particular, if G contains at least one edge then there exists an independent set I in G such that $\text{mad}(G - I) \leq \text{mad}(G) - 1$ and if G contains a cycle then there exists an induced forest F such that $\text{mad}(G - F) \leq \text{mad}(G) - 2$. As a side result, we also obtain a subexponential bound on the diameter of reconfiguration graphs of generalized colourings of graphs with bounded value of their mad .

Mathematics Subject Classifications: 05C07, 05C10, 05C15, 05C20, 05C21, 05C85

1 Introduction

The maximum average degree (abbreviated as mad) of a graph is a heavily studied notion. Multiple results show that a lower or upper bound on mad implies the existence of a particular partition of vertices of G , e.g., [5, 12, 13, 15, 24]. Another class of results considers edge partitions, including [2, 7, 17, 21, 22, 23]. In these directions, particular attention has been paid to planar graphs, where, due to the inequality $(\text{mad}(G) - 2)(g(G) - 2) < 4$, an upper bound on mad can be inferred from a lower bound on the girth.

*This research is a part of projects that have received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme Grant Agreement 714704 (W. Nadara) and 677651 (M. Smulewicz).

[†]Supported by European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme Grant Agreement 714704.

[‡]Supported by European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme Grant Agreement 677651.

A graph parameter $f(G)$ is called *partitionable* [3, 25, 29] if for every undirected simple graph G and positive real numbers a and b such that $f(G) < a + b$, the vertex set $V(G)$ can be partitioned into A and B so that $f(G[A]) < a$ and $f(G[B]) < b$. It is quite simple to prove that degeneracy [26], maximum degree [25] and treewidth [14] are all partitionable parameters. Hendrey, Norin and Wood asked whether mad is also partitionable as part of the open problems for Barbados workshop [1, Problem #14]. Such a result would agree exactly with or even improve many existing results, for example, the ones mentioned in [15, 24]. We answer this question positively for cases $a = 1$ and $a = 2$. It is a consequence of a following theorem which is the main result of this paper.

Theorem 1. *For every undirected simple graph G and a positive integer k such that $\text{mad}(G) \geq k$ there exists $S \subseteq V(G)$ such that $G[S]$ is $(k-1)$ -degenerate and $\text{mad}(G-S) \leq \text{mad}(G) - k$. Moreover such S can be computed in polynomial time.*

Up to our knowledge this is the first theorem of a kind where we are given a graph with bounded value of its mad where we partition its vertex set into some parts so that their values of mad are smaller, however they need not be bounded by absolute constant. This is opposed to all results where every resulting part induces a forest or is an independent set or has maximum degree 1, etc.

Our results can be applied as a tool for directly deriving many results for some specific sparse graph classes, for example planar graphs with constraints on girth. It seems that our results do not show as much expressive power as it is possible to get on such restrictive graph classes (where arguments specifically adjusted to the researched restricted graph classes can be used), which is a price for deriving them from a more general theorem. However, our results can be seen as a nice way of unifying these results and there are cases where using our results improves the state of the art.

Our results imply a positive answer for the open problem presented in [16] (Problem 2 from the final remarks), which implies a subexponential bound on the diameter of reconfiguration graphs of $(k+2)$ -colourings for graphs G with maximum average degree strictly less than $k+1$. However, this bound has already been improved in [18] to a polynomial bound depending on the value of $\text{mad}(G)$ in a slightly less general setting. Nevertheless, we are able to get a novel analogous result for reconfiguration graphs of H -colourings, in particular for circular colourings. This line of research is motivated by Cereceda's conjecture [10] that states that reconfiguration graphs of $(k+2)$ -colourings for k -degenerate graphs have a quadratic diameter. Its polynomial version was proved in [8].

The rest of the paper is organized as follows. In Section 2, we introduce a few useful notions. In Section 3, we state our main result and provide its proof. In Section 4, we present our reconfiguration graphs results. In Section 5, we present some direct consequences of our results and conclude this paper. In Appendix A, we present detailed proofs deferred from Section 4.

2 Preliminaries

Theorems proved in this paper will be about simple undirected graphs, however multiple directed graphs will show up throughout the proofs.

An undirected edge between vertices u and v will be denoted as uv . A directed edge from u to v will be denoted as $\overrightarrow{uv}$.

If G is a graph and A is a subset of its vertices, then by $G[A]$ we denote the subgraph of G induced on vertices of A . The length of the shortest cycle in a graph G is called girth and will be denoted as $g(G)$. If G is a forest we set that $g(G) = \infty$. The maximum degree of a vertex in a graph is denoted $\Delta(G)$. The set of neighbours of a vertex v is denoted by $N_G(v)$ (or $N(v)$ if clear from the context) and the closed neighbourhood of v , that is $N_G(v) \cup \{v\}$ is denoted by $N_G[v]$. For $Y \subseteq V(G)$ we additionally denote $N_G[Y] := \bigcup_{v \in Y} N_G[v]$. By $\overline{G}$ we denote the complement of G , that is the graph on the same set of vertices, where for $u, v \in V(G)$, such that $u \neq v$, we have $uv \in E(G) \Leftrightarrow uv \notin E(\overline{G})$.

The maximum average degree of a given graph G is defined as follows:

$$\text{mad}(G) := \max_{H \subseteq G, H \neq \emptyset} \frac{2|E(H)|}{|V(H)|},$$

where $E(H)$ and $V(H)$ are respectively the set of edges in H and the set of vertices of H . We assume that mad of a graph with an empty vertex set is $-\infty$.

We say that undirected graph G is k -degenerate if each of its subgraphs contains a vertex of degree at most k . Degeneracy of a graph is the smallest value of k such that this graph is k -degenerate.

Let us note that class of 0-degenerate graphs is exactly the same class of graphs as graphs with $\text{mad}(G) < 1$, because both are just edgeless graphs. Moreover class of 1-degenerate graphs is exactly the same class of graphs as graphs with $\text{mad}(G) < 2$, because both are just forests.

3 Proof of Theorem 1

In order to prove Theorem 1 we are going to investigate a flow network that allows us to determine the value of mad in polynomial time. An example of such network can be found in [20], however we are going to use one adjusted to our own use.

Let us define a flow network $F(G, c)$ for given undirected graph G and any nonnegative real number c . The network will consist of one node for each $v \in V(G)$, one node for

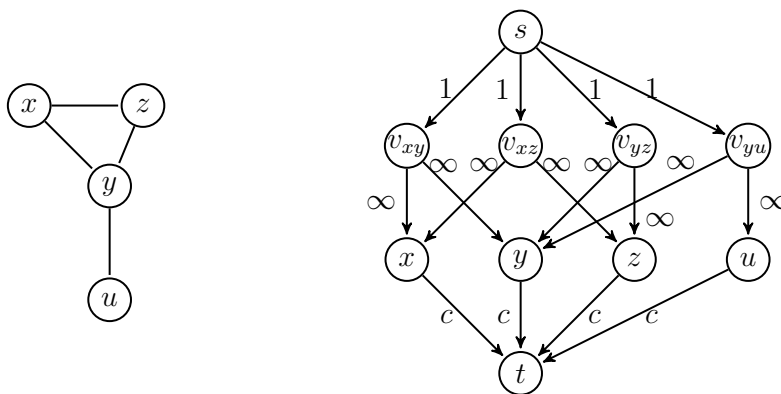


Figure 1: Example of graph G and flow network $F(G, c)$ corresponding to it.

each $e \in E(G)$ denoted as v_e and two special nodes s and t , respectively source and sink. There will be three layers of directed edges in $F(G, c)$:

- The first layer – Edges of capacity one from s to each node v_e .
- The second layer – Edges of infinite capacity from each v_e where $e = uw \in E(G)$ to u and to w .
- The third layer – Edges of capacity c from each $v \in V(G)$ to t .

Lemma 2. *For any graph G and any real number c , maximum flow between s and t in $F(G, c)$ is equal to $|E(G)|$ if and only if $2c \geq \text{mad}(G)$.*

Proof. By the max-flow min-cut theorem we know that maximum flow in a graph G is equal to the minimum cut, so we are going to investigate structure of $s - t$ cuts in this graph. We refer to cuts as sets of edges. The set of all edges from first layer form an inclusion-wise minimal cut of weight $|E(G)|$. Since edges in the second layer have infinite capacities they surely do not belong to any minimum cut, so if maximum flow is smaller than $|E(G)|$ then there exists a minimum cut with some edges in third layer. Let us fix some minimal cut $C \subseteq E(F(G, c))$ and let W be the nonempty subset of $V(G)$ of all vertices w such that $\overrightarrow{wt}$ belongs to C . Let $H = G[W]$. If $e \notin E(H)$ then $\overrightarrow{sv_e}$ has to belong to C . Observe that all mentioned edges, that is $\overrightarrow{wt}$ for $w \in W$ and $\overrightarrow{sv_e}$ for $e \notin E(H)$ already form an $s - t$ cut. Its weight is $c|V(H)| + |E(G)| - |E(H)|$. If this value is less than $|E(G)|$ then we know that maximum flow in this graph is less than $|E(G)|$. However, if for any H this value is not smaller than $|E(G)|$ then we know that maxflow in this graph is $|E(G)|$.

We get that maxflow in this graph is smaller than $|E(G)|$ if and only if there exists $H \subseteq G$ such that $c|V(H)| + |E(G)| - |E(H)| < |E(G)| \Leftrightarrow c|V(H)| < |E(H)| \Leftrightarrow c < \frac{|E(H)|}{|V(H)|}$. The maximum value of $\frac{|E(H)|}{|V(H)|}$ equals $\frac{\text{mad}(G)}{2}$, so we get that maxflow in $F(G, c)$ is equal to $|E(G)|$ if and only if $c \geq \frac{\text{mad}(G)}{2}$, as desired. $\square$

Let us note that by using Lemma 2, observing that $\text{mad}(G) = \frac{a}{b}$ for some $a, b \in \mathbb{Z}$ and $a \leq n^2, b \leq n$ and knowing that we can compute maximum flow in polynomial time, we can conclude that $\text{mad}(G)$ can be computed in polynomial time.

Let us fix any graph G and denote $F := F(G, \frac{\text{mad}(G)}{2})$. Let us define a directed graph G_f for a given $s - t$ flow f in F of capacity $|E(G)|$ by directing some of edges from G and discarding the rest. Flow f routes one unit of flow through each v_{uw} . Node v_{uw} has two outgoing edges to u and to w . If f sends more than $\frac{1}{2}$ unit of flow to w then in G_f we put directed edge $\overrightarrow{uw}$, similarly if f sends more than $\frac{1}{2}$ unit of flow to u we put edge $\overrightarrow{wu}$. Otherwise if f sends exactly $\frac{1}{2}$ unit to both u and w we simply discard this edge.

Lemma 3. *There exists flow f of capacity $|E(G)|$ in F such that G_f is acyclic. Moreover, it can be determined in polynomial time.*

Proof. From Lemma 2 we know that there exists at least one flow f between s and t of capacity $|E(G)|$. Let us take f such that number of edges in G_f is as small as possible. Suppose there is a cycle in G_f on vertices $c_1, c_2, \dots, c_k$ respectively. Denote $c_{k+1} := c_1$ as we are dealing with a cycle. Let x be the minimum amount of flow that f sends through

some edge $\overrightarrow{v_{c_i c_{i+1}} c_{i+1}}$ for some valid i . From definition of G_f we deduce that $x > \frac{1}{2}$. Let us define f' by decreasing flow f on edges $\overrightarrow{v_{c_i c_{i+1}} c_{i+1}}$ and increasing it on edges $\overrightarrow{v_{c_i c_{i+1}} c_i}$ by $x - \frac{1}{2}$. The amount of flow leaving and entering each vertex remains unchanged hence f' is also a flow. Moreover, f' still satisfies the capacity constraints. Flow f' for at least one vertex $v_{c_i c_{i+1}}$ sends exactly $\frac{1}{2}$ unit of flow through both edges outgoing from it, so at least one edge on the cycle is no longer present in $G_{f'}$ and edges outside the cycle remain unchanged when compared to G_f . This contradicts the assumption that G_f has the smallest possible number of edges, which implies the existence of such an f .

In order to compute such f in polynomial time let us take any f of capacity $|E(G)|$ in $F(G, \frac{\text{mad}(G)}{2})$ (let us remind the reader that we can determine the value of $\text{mad}(G)$ in the polynomial time). If G_f contains a cycle, we can detect one, determine the corresponding value of x and adjust f in the manner described in previous paragraph to remove this cycle. The number of edges in $G_{f'}$ is strictly smaller than in G_f , so we will not do this more than $|E(G)|$ times, which gives us an algorithm performing a polynomial number of operations. In order to omit dealing with rational numbers we can multiply all capacities in F by $2b$, where $\text{mad}(G) = \frac{a}{b}$ for some coprime integers a, b . That concludes the description of a polynomial time algorithm determining the desired f . $\square$

Let us fix f from the above lemma. We will present an algorithm in which:

- the routine $NoInEdges(H_f)$ returns any vertex from directed acyclic graph H_f which has no incoming edges (as the graph is acyclic there always exists at least one such vertex)
- the routine $KNeighborhood(H, S, k)$ takes as input a given graph H , a subset of its vertices S and an integer k , and returns the set of all vertices from H outside of S adjacent to at least k vertices from S .

Algorithm 1

```

function SOLVE( $H, H_f, k$ )
   $S \leftarrow \emptyset$ 
  while  $H_f \neq \emptyset$  do
     $x \leftarrow NoInEdges(H_f)$ 
     $S \leftarrow S \cup \{x\}$ 
     $H_f \leftarrow H_f - \{x\} - KNeighborhood(H, S, k)$ 
  return  $S$ 

```

Theorem 4. *For positive integer k such that $\text{mad}(G) \geq k$ algorithm $Solve(G, G_f, k)$ returns a set $S \subseteq V(G)$ such that $G[S]$ is a $(k-1)$ -degenerate and that $\text{mad}(G-S) \leq \text{mad}(G) - k$.*

Proof. First we argue that the graph induced on the set of vertices returned by the algorithm is $(k-1)$ -degenerate. In each iteration the vertex x picked by the algorithm is adjacent to at most $k-1$ already picked vertices. So $G[S]$ is $(k-1)$ -degenerate indeed.

To show that $\text{mad}(G-S) \leq \text{mad}(G) - k$ we just have to find a flow f' in graph $F' := F(G-S, \frac{\text{mad}(G)}{2} - \frac{k}{2})$ of value $|E(G-S)|$ thanks to Lemma 2. Observe that F'

is a subgraph of F with capacities of edges on the third layer reduced by $\frac{k}{2}$. The flow f' has to saturate all edges from the first layer in order to have value $|E(G - S)|$. On the second layer we define f' using f , for each edge from the second layer of F' flow f' will send exactly the same amount of flow as f on corresponding edge in F . Now we just have to argue that the amount of flow sent by f' to any node between the second and third layer in F' is bounded by $\frac{\text{mad}(G)}{2} - \frac{k}{2}$ i.e. capacity of edge going from that node to sink. Each such node corresponds to vertex from $G - S$, so let us take arbitrary vertex $u \in V(G - S)$. During execution of the algorithm vertex u has been removed from H_f as incident to some k vertices already picked to S . Denote them $x_1, \dots, x_k$ and let us consider arbitrary x_i . When the algorithm picked x_i from H_f , there were no incoming edges to x_i . In particular, in H_f there was no edge $\overrightarrow{ux_i}$. At that time u still belonged to H_f , so there was no edge $\overrightarrow{ux_i}$ even in G_f . Since u and x_i are adjacent in G , there was either an edge $\overrightarrow{x_i u}$ in G_f which means that flow f sends more than $\frac{1}{2}$ unit of flow from v_{ux_i} to u in F or there was no $\overrightarrow{x_i u}$ and $\overrightarrow{ux_i}$ which means that flow f sends exactly $\frac{1}{2}$ unit of flow from v_{ux_i} to u in F . Through node u in F flow f sends at most $\frac{\text{mad}(G)}{2}$ units of flow and for every $1 \leq i \leq k$ at least $\frac{1}{2}$ unit of flow comes from v_{ux_i} to u . Therefore flow going through u is decreased by at least $\frac{1}{2}$ unit per each x_i in F' what implies that f' sends at most $\frac{\text{mad}(G)}{2} - \frac{k}{2}$ units of flow to vertex u in F' . $\square$

What is more, procedure $\text{Solve}(G, G_f, k)$ can be trivially implemented in a polynomial time. Theorem 1 directly follows from Theorem 4. As two notable special cases we mention following corollaries:

Theorem 5. *For every undirected simple graph G there exists $I \subseteq V(G)$ such that I is an independent set and $\text{mad}(G - I) \leq \text{mad}(G) - 1$. Moreover such I can be computed in polynomial time.*

Theorem 6. *For every undirected simple graph G there exists $F \subseteq V(G)$ such that $G[F]$ is a forest and $\text{mad}(G - F) \leq \text{mad}(G) - 2$. Moreover such F can be computed in polynomial time.*

4 Reconfiguration graphs results

The reconfiguration graph $R_k(G)$ for a positive integer k and a graph G is the graph whose vertex set is the set of k -colourings of G and there is an edge between two colourings if and only if they differ by colour of exactly one vertex.

Let us recall the Cereceda's conjecture [10].

Conjecture 7. Let k and $l \geq k + 2$ be positive integers and let G be a k -degenerate graph on n vertices. Then $R_l(G)$ has diameter $\mathcal{O}(n^2)$.

As we have already mentioned, our results imply the positive answer to the last remaining step in the outline of the proof by Eiben and Feghali [16] of the following fact:

Theorem 8. *Let $k \geq 2$ and $l \geq k + 2$ be integers and let G be a graph on n vertices such that $\text{mad}(G) < k + 1$. Then $R_l(G)$ has diameter $k^{\mathcal{O}(k^2 \sqrt{n})}$.*

Let us remind that any graph with $\text{mad}(G) < k + 1$ is k -degenerate, hence the setting of Theorem 8 can be seen as an easier setting of Conjecture 7. The obtained bound on the diameter is worse as well. Nonetheless, Theorem 8 seems interesting as previous proofs regarding the connectivity of such reconfiguration graphs yield only exponential bounds. However, this bound under a slightly less general assumption has already been improved by Feghali [18] to the polynomial one. Nevertheless, the idea of the proof of Theorem 8 transfers over to a novel analogous result for reconfiguration graphs of colouring viewed as homomorphism to a given graph H , which we shall present now.

For a given graph H , the H -colouring of a graph G is any homomorphism $f : V(G) \rightarrow V(H)$, that is, a function f such that if $uv \in E(G)$, then $f(u)f(v) \in E(H)$. We call $f(v)$ the colour of v . In particular, if H is loopless and $uv \in E(H)$, then it must hold that $f(u) \neq f(v)$. The H -reconfiguration graph of G is the graph whose vertex set consists of all H -colourings of G and two colourings are adjacent if and only if they differ by colour of exactly one vertex.

The main result of this section is the following theorem:

Theorem 9. *Let $k \geq 2$ be a positive integer and let $G = (V, E)$ be a graph on n vertices with $\text{mad}(G) < k + 1$ and let H be a graph such that $\Delta(\overline{H}) \leq d$ and $|V(H)| \geq (d + 1)(k + 1) + 1$. Let α and β be two H -colourings of G . It is possible to get β from α by a sequence of $k^{\mathcal{O}(k^2\sqrt{n})}$ recolourings.*

Note that if we set $H = K_l$ and $d = 0$ we get the exact statement of Theorem 8, as regular colouring is exactly H -colouring for clique H . Moreover, to prove that fact we mainly follow the outline of the proof by Eiben and Feghali [16] of Theorem 8. Since the proof requires following closely proofs of all the intermediate lemmas and generalizing them to H -colourings, we defer this proof to the Appendix.

As a motivation for such generalization, we use the notion of circular colourings. We focus on the following description of the circular colourings [9], also known as (p, q) -colouring [30].

Definition 10. Let a and b , where $a \geq 2b$, be positive integers. The *circular clique* $G_{a,b}$ has the vertex set $\{0, 1, \dots, a - 1\}$ where ij is an edge when $b \leq |i - j| \leq a - b$. A homomorphism $\phi : G \rightarrow G_{a,b}$ is called a *circular colouring* in general, and an (a, b) -colouring of G for the specific pair (a, b) .

We remark that $G_{a,1}$ is isomorphic to K_a and so an $(a, 1)$ -colouring is simply an a -colouring.

Brewster and Noel [9] have proved that for a given graph G and positive integers a, b if $\frac{a}{b} \geq 2l + 2$, where l is the degeneracy of G , then the reconfiguration graph of (a, b) -colourings of G is connected. However, the bound on its diameter that follows from their proof is exponential. Based on our H -colourings result, we are able to deduce the following bound on this diameter.

Corollary 11. *Let $k \geq 2$ and $G = (V, E)$ be a graph on n vertices with $\text{mad}(G) < k + 1$. Let $a \geq 2b$ be positive integers. If $\frac{a}{b} \geq 2k + 2$, then the (a, b) -reconfiguration graph of G is connected and has diameter $k^{\mathcal{O}(k^2\sqrt{n})}$.*

Proof. We use Theorem 9 for $H := G_{a,b}$. The complement of $G_{a,b}$ is $(2b - 2)$ -regular, hence we set $d := 2b - 2$. If $\frac{a}{b} \geq 2k + 2$, then $V(H) = a \geq b(2k + 2) = (d + 2)(k + 1) \geq$

$(d+1)(k+1)+1$, hence all assumptions of that theorem are satisfied and the conclusion follows. $\square$

5 Conclusions and open problems

Our main results imply many results for some specific classes of graphs as a direct consequence and here we mention a few of them.

Following folklore fact will come in handy in deriving some of the consequences:

Fact 12. *For every planar graph G we have $(\text{mad}(G) - 2)(g(G) - 2) < 4$.*

Based on Theorems 5, we are able to improve Theorem 1 from [15] and one of its consequences.

Theorem 13 ([15]). *Let M be a real number such that $M < 3$. Let $d \geq 0$ be an integer and let G be a graph with $\text{mad}(G) < M$. If $d \geq \frac{2}{3-M} - 2$, then $V(G)$ can be partitioned into $A \uplus B$ such that $G[A]$ is an independent set and $G[B]$ is a forest with maximum degree at most d .*

We are able to strengthen this to the following theorem:

Theorem 14. *Let M be a real number such that $M < 3$. Let $d \geq 0$ be an integer and let G be a graph with $\text{mad}(G) < M$. If $d \geq \frac{2}{3-M} - 2$, then $V(G)$ can be partitioned into $A \uplus B$ such that $G[A]$ is an independent set and $G[B]$ is a forest whose connected components have size at most $d+1$.*

Proof. By using Theorem 5 one can partition $V(G)$ into $A \uplus B$ such that $G[A]$ is an independent set and $\text{mad}(G[B]) < M - 1$. Let us bound $M - 1$ from above using assumed inequalities:

$$d \geq \frac{2}{3-M} - 2 \Rightarrow d+2 \geq \frac{2}{3-M} \Rightarrow 3-M \geq \frac{2}{d+2} \Rightarrow M-1 \leq 2 - \frac{2}{d+2}.$$

$G[B]$ does not contain a cycle because $\text{mad}(G[B]) < M - 1 \leq 2 - \frac{2}{d+2} < 2$. Assume that $G[B]$ contains a tree T on $d+2$ vertices as a subgraph. Then $\text{mad}(G[B]) \geq \frac{2|E(T)|}{|V(T)|} = \frac{2(d+1)}{d+2} = 2 - \frac{2}{d+2} \geq M - 1 > \text{mad}(G[B])$. Note that if $G[B]$ contains a component which is a tree on at least $d+2$ vertices then it contains tree on exactly $d+2$ vertices as a subgraph, hence shown contradiction finishes the proof that connected components of $G[B]$ are trees on at most $d+1$ vertices. $\square$

Dross et al. [15] use their theorem and Fact 12 to deduce the following corollary:

Corollary 15. *For every planar graph G with $g(G) \geq 10$, the vertex set $V(G)$ can be partitioned into $A \uplus B$ such that A is an independent set and $G[B]$ is a forest with maximum degree 2.*

We are able to strengthen this to the following corollary:

Corollary 16. *For every planar graph G with $g(G) \geq 10$, the vertex set $V(G)$ can be partitioned into $A \uplus B$ such that A is an independent set and $G[B]$ is a forest whose connected components have size at most 3.*

Borodin et al. proved in [6] that the vertex set of any planar graph with $g(G) \geq 7$ admits a partition into an independent set and a set that induces graph with maximum degree at most 4. Dross et al. proved in [15] that the vertex set of any planar graph with $g(G) \geq 7$ admits a partition into an independent set and a set that induces forest of max degree at most 5. In Corollary 17 we add another partition result for the class of planar graphs with girth at least 7.

Corollary 17. *For every planar graph G with $g(G) \geq 7$, the vertex set $V(G)$ can be partitioned into $A \uplus B$ such that A is an independent set and $G[B]$ is a forest where every connected component has at most 9 vertices.*

Proof. Since $g(G) \geq 7$ we deduce that $\text{mad}(G) < 1 + \frac{9}{5}$, so based on Theorem 5 we get that there exist A and B such that $V(G) = A \uplus B$, A is an independent set and $\text{mad}(G[B]) < \frac{9}{5}$. It can be readily verified that class of graphs with value of their mad smaller than $\frac{9}{5}$ is class of graphs which are forests with connected components of size at most 9. $\square$

Recently, independently of our work, Cranston and Yancey [13] improved Corollaries 16 and 17. Namely, they claim that every planar graph G of girth at least 9 (resp. 8, 7) has a partition of $V(G)$ into an independent set I and a set F such that $G[F]$ is a forest with each component of order at most 3 (resp. 4, 6).

Apart from that, based on Theorem 5 and 6 and Fact 12 we are able to deduce following corollaries:

Corollary 18. *For every planar graph G , the vertex set $V(G)$ can be partitioned into $A \uplus B \uplus C$ such that $G[A], G[B], G[C]$ are forests.*

Proof. Every planar graph satisfies $\text{mad}(G) < 6$, so using Theorem 6 we can partition $V(G)$ into A and D such that $\text{mad}(G[A]) < 2$ and $\text{mad}(G[D]) < 4$ and then using Theorem 6 again we can partition D into B and C such that $\text{mad}(G[B]) < 2$ and $\text{mad}(G[C]) < 2$. Hence $G[A], G[B], G[C]$ are forests. $\square$

Corollary 19. *For every planar graph G without triangles, the vertex set $V(G)$ can be partitioned into $A \uplus B$ such that $G[A], G[B]$ are forests.*

Proof. Based on Fact 12 we know that if G has no triangles then $g(G) \geq 4 \Rightarrow g(G) - 2 \geq 2 \Rightarrow \text{mad}(G) < 4$. Therefore using Theorem 6 we deduce that there exist A, B such that $V(G) = A \uplus B$ and $G[A], G[B]$ are forests. $\square$

Corollary 20. *For every planar graph G without cycles of length 3 and 4, the vertex set $V(G)$ can be partitioned into $A \uplus B$ such that $G[A]$ is a forest and $\Delta(G[B]) \leq 1$.*

Proof. Since $g(G) \geq 5$ we deduce that $\text{mad}(G) < 2 + \frac{4}{3}$, so based on Theorem 6 we get that there exist A and B such that $V(G) = A \uplus B$ and $\text{mad}(G[A]) < 2$ and $\text{mad}(G[B]) < \frac{4}{3}$. Therefore $G[A]$ is a forest and $\Delta(G[B]) \leq 1$, because if $G[B]$ contains a vertex with degree ≥ 2 then this vertex together with its two neighbours induce a graph with mad at least $\frac{4}{3}$. $\square$

Corollary 21. *For every planar graph G with $g(G) \geq 6$ its vertex set $V(G)$ can be partitioned into $A \uplus B$ such that $G[A]$ is a forest and B is an independent set.*

Proof. Since $g(G) \geq 6$ we deduce that $\text{mad}(G) < 3$, so based on either Theorem 6 or Theorem 5 we get that there exist A and B such that $V(G) = A \uplus B$ and $\text{mad}(G[A]) < 2$ and $\text{mad}(G[B]) < 1$. Therefore $G[A]$ is a forest and B is an independent set. $\square$

However, Corollaries 18, 19, 20 and 21 have already been proven and even improved before. Corollary 18 was proven in [11] and later improved in [27]. An improved version of Corollary 19 was proven in [28]. Theorem improving both Corollaries 20 and 21 was proven in [4].

5.1 Open problem

As the main open problem in the area of partitionability of graphs with bounded mad , we recall the following conjecture.

Conjecture 22. For every graph G and positive real numbers c_1, c_2 if $\text{mad}(G) < c_1 + c_2$ then there exists a partition of the vertex set $V(G) = A \uplus B$ such that $\text{mad}(G[A]) < c_1$ and $\text{mad}(G[B]) < c_2$.

Our main result shows that this conjecture is true for $c_2 \in \{1, 2\}$. Moreover, since for positive k we have that k -degenerate graphs fulfill $\text{mad}(G) < 2k$ we can deduce that for every integer $k \geq 2$ and a graph G that satisfies $\text{mad}(G) < c_1 + k$ there exists a partition of the vertex set $V(G) = A \uplus B$ such that $\text{mad}(G[A]) < c_1$ and $\text{mad}(G[B]) < 2k - 2$.

Acknowledgements

The majority of this research was conducted during the Structural Graph Theory workshop in Gułtowy, Poland in June 2019. We thank the organizers and the other workshop participants for creating a productive working atmosphere. We would also like to thank Bartosz Walczak for discussions on this topic and anonymous referees of the Electronic Journal of Combinatorics for their valuable comments.

References

- [1] *Open problems for the Barbados graph theory workshop 2019*. Edited by L. Cook and S. Spirkl, Available at: <https://sites.google.com/site/sophiespirkl/open-problems/2019-open-problems-for-the-barbados-graph-theory-workshop>.
- [2] A. Bassa, J. Burns, J. Campbell, A. Deshpande, J. Farley, M. Halsey, S.-Y. Ho, D. Kleitman, S. Michalakis, P.-O. Persson, P. Pylyavskyy, L. Rademacher, A. Riehl, M. Rios, J. Samuel, B. Tenner, A. Vijayarathy, and L. Zhao. Partitioning a planar graph of girth 10 into a forest and a matching. *Studies in Applied Mathematics*, 124:213 – 228, 01 2010.
- [3] B. Bollobás and B. Manvel. Optimal vertex partitions. *Bulletin of the London Mathematical Society*, 11(2):113–116, 1979.
- [4] O. V. Borodin and A. Glebov. On the partition of a planar graph of girth 5 into an empty and an acyclic subgraph. *Diskretnyj Analiz i Issledovanie Operatsij. Seriya 1*, 01 2001.
- [5] O. V. Borodin and A. V. Kostochka. Vertex decompositions of sparse graphs into an independent vertex set and a subgraph of maximum degree at most 1. *Siberian Mathematical Journal*, 52(5):796, Oct 2011.

- [6] O. V. Borodin and A. V. Kostochka. Defective 2-colorings of sparse graphs. *Journal of Combinatorial Theory, Series B*, 104:72 – 80, 2014.
- [7] O. V. Borodin, A. V. Kostochka, N. N. Sheikh, and G. Yu. Decomposing a planar graph with girth 9 into a forest and a matching. *European Journal of Combinatorics*, 29(5):1235–1241, 2008.
- [8] N. Bousquet and M. Heinrich. A polynomial version of cereceda’s conjecture. *Journal of Combinatorial Theory, Series B*, 155:1–16, 2022.
- [9] R. Brewster and J. Noel. Mixing homomorphisms, recolorings, and extending circular precolorings. *Journal of Graph Theory*, 80(3):173–198, 2015.
- [10] L. Cereceda. Mixing graph colourings. PhD thesis, London School of Economics, 2007.
- [11] G. Chartrand, H. V. Kronk, and C. E. Wall. The point-arboricity of a graph. *Israel Journal of Mathematics*, 6(2):169–175, Apr 1968.
- [12] M. Chen, W. Yu, and W. Wang. On the vertex partitions of sparse graphs into an independent vertex set and a forest with bounded maximum degree. *Applied Mathematics and Computation*, 326:117–123, 2018.
- [13] D. W. Cranston and M. P. Yancey. Vertex partitions into an independent set and a forest with each component small. *SIAM Journal on Discrete Mathematics*, 35(3):1769–1791, 2021.
- [14] G. Ding, B. Oporowski, D. P. Sanders, and D. Vertigan. Partitioning graphs of bounded tree-width. *Combinatorica*, 18(1):1–12, Jan 1998.
- [15] F. Dross, M. Montassier, and A. Pinlou. Partitioning sparse graphs into an independent set and a forest of bounded degree. *Electronic Journal of Combinatorics*, 25(1):#P1.45, 2018.
- [16] E. Eiben and C. Feghali. Toward Cereceda’s conjecture for planar graphs. *Journal of Graph Theory*, 94(2):267–277, 2020.
- [17] G. Fan, Y. Li, N. Song, and D. Yang. Decomposing a graph into pseudoforests with one having bounded degree. *Journal of Combinatorial Theory, Series B*, 115:72 – 95, 2015.
- [18] C. Feghali. Reconfiguring colorings of graphs with bounded maximum average degree. *Journal of Combinatorial Theory, Series B*, 147:133–138, 2021.
- [19] C. Feghali, M. Johnson, and D. Paulusma. A reconfigurations analogue of Brooks’ theorem and its consequences. *Journal of graph theory.*, 83(4):340–358, December 2016.
- [20] A. V. Goldberg. Finding a maximum density subgraph. Technical Report UCB/CSD-84-171, EECS Department, University of California, Berkeley, 1984.
- [21] L. Grout and B. Moore. The pseudoforest analogue for the strong nine dragon tree conjecture is true. *Journal of Combinatorial Theory, Series B*, 145:433 – 449, 2020.
- [22] W. He, X. Hou, K. Lih, J. Shao, W. Wang, and X. Zhu. Edge-partitions of planar graphs and their game coloring numbers. *Journal of Graph Theory*, 41(4):307–317, 2002.
- [23] S.-J. Kim, A. V. Kostochka, D. B. West, H. Wu, and X. Zhu. Decomposition of sparse graphs into forests and a graph with bounded degree. *Journal of Graph Theory*, 74(4):369–391, 2013.
- [24] M. Kopreski and G. Yu. Maximum average degree and relaxed coloring. *Discrete Mathematics*, 340(10):2528–2530, 2017.
- [25] L. Lovász. On decomposition of graphs. *Studia Sci. Math. Hungar.*, 1:237–238, 1966.
- [26] P. Mihók. Minimal reducible bounds for the class of k -degenerate graphs. *Discrete Mathematics*, 236(1):273 – 279, 2001. Graph Theory.
- [27] K. S. Poh. On the linear vertex-arboricity of a planar graph. *Journal of Graph Theory*, 14(1):73–75, 1990.

- [28] A. Raspaud and W. Wang. On the vertex-arboricity of planar graphs. *European Journal of Combinatorics*, 29(4):1064 – 1075, 2008. Homomorphisms: Structure and Highlights.
- [29] D. R. Wood. Acyclic, star and oriented colourings of graph subdivisions. *Discrete Mathematics and Theoretical Computer Science*, 7(1):37–50, 2005.
- [30] X. Zhu. Recent developments in circular colouring of graphs. In M. Klazar, J. Kratochvíl, M. Loeb, J. Matoušek, P. Valtr, and R. Thomas, editors, *Topics in Discrete Mathematics*, pages 497–550, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

A Proof of the bound on the diameter of reconfiguration graphs of H -colourings of graphs with bounded maximum average degree

The overall structure of proof of Theorem 9 consists of combining the generalization of Lemma 2 from [16] with Theorem 5 in the same way that Lemmas 8, 9 and 10 in [19] are combined to obtain Theorem 6 in [19]. It is the generalization of the outline of the analogous result for standard colourings from [16].

Lemma 23. *Let $k, d \geq 0$, and let G be the graph on n vertices such that $\text{mad}(G) < k + 1$. Let $\{u_1, \dots, u_s\}$ be the set of vertices of G of degree at least $k + 2$. Let H be a graph such that $\Delta(\overline{H}) \leq d$ and $|V(H)| \geq (d + 1)(k + 1) + 1$ and let α be a H -colouring of G . Let F be any subset of $V(H)$ of size at most $d + 1$. It is possible to reconfigure α to some H -colouring α' of G such that $\alpha'(V(G)) \subseteq V(H) - F$ by using at most $n^2 \prod_{i=1}^s \deg(u_i)$ recolourings.*

Let us note that the case $d = 0$ coincides with Lemma 1 from [16] with the slight change in the assumption on G . The proof presented here is a generalization of its proof.

Proof. Since $\text{mad}(G) < k + 1$, we know that G is k -degenerate, hence we can fix its k -degenerate ordering $\sigma = v_1, \dots, v_n$ and without loss of generality, let u_i appear before u_j in σ whenever $i < j$. In the following, we will describe an algorithm $\text{Recolour}(h, F_h)$, which given an index $h \in [n]$ and the subset $F_h \subseteq V(H)$ of forbidden colours for v_h such that $|F_h| \leq d + 1$, outputs a sequence of recolourings with the following properties:

- for $i < h$, v_i is not recoloured,
- for $i \geq h$, v_i is recoloured at most $\prod_{j=i}^s \deg(u_j)$ times, where u_l is the first vertex of degree at least $k + 2$ with index at least h in σ
- v_h ends up with a colour from $V(H) - F_h$ (in particular, if $f(v_h) \in V(H) - F_h$, then an empty sequence is a feasible one, where $f(v_h)$ denotes the current colour of v_h)

Notice that the algorithm takes at most $n \prod_{i=1}^s \deg(u_i)$ recolourings to recolour v_h . Hence, by repeatedly calling $\text{Recolour}(i, F)$ for $i = 1, \dots, n$, we obtain the colouring α' in which colours from F do not appear by using at most $n^2 \prod_{i=1}^s \deg(u_i)$ recolourings, as required.

Given $h \in [n]$, k -degenerate ordering $\sigma = v_1, \dots, v_n$ of G and the set F_h of forbidden colours, the algorithm $\text{Recolour}(h, F_h)$ works as follows:

1. If $f(v_h) \notin F_h$, then terminate.

2. If v_h has degree at most k , then

- Let c be a colour not belonging to $N_{\overline{H}}[f(N(v_h))] \cup F_h$. Such colour exists since $\Delta(\overline{H}) \leq d$, so $|N_{\overline{H}}[f(N(v_h))] \cup F_h| \leq (d+1)k + (d+1) < |V(H)|$.
- Recolour v_h to c .

3. If v_h has degree at least $k+1$, then

- Let c be a colour not belonging to $N_{\overline{H}}[f(Z)] \cup F_h$, where Z is the set consisting of first k neighbours of v_h in ordering σ . Such colour exists since $\Delta(\overline{H}) \leq d$, so $|N_{\overline{H}}[f(Z)] \cup F_h| \leq (d+1)k + (d+1) < |V(H)|$.
- Let $v_{i_1}, \dots, v_{i_t}$ be the neighbours of v_h outside Z with $i_1 < i_2 < \dots < i_t$. Note that $h < i_1$, since G is k -degenerate.
- For each $j \in [t]$ in the ascending order call $\text{Recolour}(i_j, N_{\overline{H}}[c])$.
- Recolour v_h to c .

It is clear that this algorithm is correct. To estimate the number of used recolourings, it is sufficient to observe that the recursion branches only on vertices of degree at least $k+2$. $\square$

Lemma 24. *Let $k \geq 1, d \geq 0$ and let G be a graph on n vertices and with $\text{mad}(G) < k+1$. Let H be a graph such that $\Delta(\overline{H}) \leq d$ and $|V(H)| \geq (d+1)(k+1)+1$ and let α be a H -colouring of G . Let F be a subset of $V(H)$ of size at most $d+1$. It is possible to reconfigure α to some H -colouring α' of G such that $\alpha'(V(G)) \subseteq V(H) - F$ by $\max(k, 2)^{\mathcal{O}(k^2\sqrt{n})}$ recolourings.*

Let us note again that the case $d = 0$ coincides with Lemma 2 from [16] and that the proof presented here is a generalization of its proof.

Proof. We will call H -colourings of G *small* if they do not use colours from F . We will denote $k' = \max(k, 2)$.

We shall prove by the induction on the size $n := |V(G)|$ that we can reconfigure α to a small H -colouring α' , such that each vertex in G is recoloured at most $n^2 \cdot k'^{9k'^2\sqrt{n}}$ times, which implies the lemma.

As the base case we distinguish graphs P on p vertices such that $\text{mad}(P) < k+1$ that contain at most $2(k+1)\sqrt{p}$ vertices of degree at most k . Let $\{u_1, \dots, u_s\}$ be a set of vertices of P of degree at least $k+2$. In the case of such graphs $\prod_{i=1}^s \deg(u_i)$ is bounded from above by $(2(k+1))^{2k(k+1)\sqrt{p}} \leq (k^3)^{3k'^2\sqrt{p}} = k'^{9k'^2\sqrt{p}}$ as proven by Eiben and Feghali as part of the proof of Lemma 2 in [16]. Thus, in this case we can use algorithm from Lemma 23 and prove that we can get α' using at most $p^2 \prod_{i=1}^s \deg(u_i) \leq p^2 k'^{9k'^2\sqrt{p}}$ recolourings in total, in particular the number of recolourings of each particular vertex is bounded by $p^2 k'^{9k'^2\sqrt{p}}$.

For the inductive step, suppose that G contains more than $2(k+1)\sqrt{n}$ vertices of degree at most k and that we can reconfigure any subgraph P of G with $p < n$ vertices to some small H -colouring α_P such that each vertex gets recoloured at most $p^2 k'^{9k'^2\sqrt{p}}$ times. Let S be an independent set in G of size at least $2\sqrt{n}$ containing only vertices of degree at most k . Note that G is k -degenerate, so it can be partitioned into $k+1$ independent

sets and one of them has to contain at least $2\sqrt{n}$ vertices of degree at most k , so such S exists. Using an inductive argument, we can recolour the graph $P = G - S$ to some small H -colouring through a recolouring sequence R . We can extend this sequence of recolourings in P to a sequence in G in the following way. Let a recolouring of $v \in V(P)$ from c_1 to c_2 be one of the recolouring operations from R and let $u \in S$ be a neighbour of v . Let C be the sum of $\{c_1, c_2\}$ and colours present in the neighbours of u in P other than v . We have that $|C| \leq k + 1$, so $N_{\overline{H}}[C] \leq (d + 1)(k + 1) < |V(H)|$, so there exists a colour $c \in V(H) - N_{\overline{H}}[C]$. In the reconfiguration sequence extended to G we put an operation of recolouring u to c before recolouring v . This way we get a valid recolouring sequence for G . Finally, we recolour each vertex $u \in S$ to any colour not belonging to $F \cup N_{\overline{H}}[f(N_G(u))]$. It is possible as $|F \cup N_{\overline{H}}[f(N_G(u))]| \leq (d + 1)(k + 1) < |V(H)|$. For each vertex outside S we recoloured it exactly as many times as in R , whereas for each vertex in S we recoloured it at most as many times as all its neighbours were recoloured in total in R plus one.

Let $g(n)$ be the maximum number of times that a vertex of a graph on n vertices is recoloured in this process. In the base case we require at most $n^2 k^{9k^2/2} \sqrt{n}$ recolourings, while in the inductive step case we require at most $k \cdot g(\lfloor n - 2\sqrt{n} \rfloor) + 1$ recolourings, hence we have $g(n) \leq \max(n^2 \cdot k^{9k^2/2} \sqrt{n}, k \cdot g(\lfloor n - 2\sqrt{n} \rfloor) + 1)$. However, simple calculations show that $k \cdot k^{9k^2/2} \sqrt{n - 2\sqrt{n}} + 1 \leq k^{9k^2/2} \sqrt{n}$ (as $\sqrt{n} > \sqrt{n - 2\sqrt{n}} + 1$), so $n^2 \cdot k \cdot k^{9k^2/2} \sqrt{n - 2\sqrt{n}} + 1 \leq n^2 \cdot k^{9k^2/2} \sqrt{n}$, hence $g(n) \leq n^2 \cdot k^{9k^2/2} \sqrt{n}$, which concludes this proof that each vertex can be recoloured at most $n^2 \cdot k^{9k^2/2} \sqrt{n}$ times. Thus, in total we get a sequence of at most $n^3 \cdot k^{9k^2/2} \sqrt{n} = \max(2, k)^{\mathcal{O}(k^2 \sqrt{n})}$ recolourings. $\square$

Remark. If we were a bit more meticulous, then in Lemma 23 it would be possible to get the bound of $\mathcal{O}(n^2 \prod_{i=1}^s (deg(u_i) - k))$ instead of $\mathcal{O}(n^2 \prod_{i=1}^s deg(u_i))$. In consequence, it would be possible to improve the bound from Lemma 24 to $2^{\mathcal{O}(k^2 \sqrt{n})}$, but proving that would lead to significantly longer exposition and the improvement would not be that significant, hence we decided not to present it.

Now we are ready to prove Theorem 9.

Proof. Let $k \geq 0$ be a positive integer and let $G = (V, E)$ be a graph on n vertices with $\text{mad}(G) < k + 1$ and let H be a graph such that $\Delta(\overline{H}) \leq d$ and $|V(H)| \geq (d + 1)(k + 1) + 1$. We will define a H -colouring of G called γ and then prove that any H -colouring δ can be reconfigured to γ by a sequence of at most $\max(2, k)^{\mathcal{O}(\max(k^2, 1)\sqrt{n})}$ recolourings, which clearly implies the thesis by using twice this statement for $\delta = \alpha$ and for $\delta = \beta$.

We will prove this inductively on k . The base case $k = 0$ is trivial: if $\text{mad}(G) < 1$, then G is an edgeless graph and every colouring is valid. We can set γ as an arbitrary colouring of G . Now we can recolour each vertex directly from colour from δ to colour from γ using at most n recolourings in total.

Now we present the inductive step. Let us consider an independent set I from Theorem 5 such that $\text{mad}(G - I) < k$ and any colour $u \in V(H)$. For each $v \in I$ we set $\gamma(v) = u$. Thanks to Lemma 24 for $F := N_{\overline{H}}[u]$, we are able to reconfigure δ to a colouring δ' that does not use colours from $N_{\overline{H}}[u]$. Afterwards, we reconfigure δ' to δ'' by recolouring each vertex from I to the colour u . We define $G' = G - I$ and $H' = H - N_{\overline{H}}[u]$. Graphs G' and H' meet the assumptions of the inductive hypothesis for $k' = k - 1$. So there exists γ' independent of δ'' and a sequence of recolourings R' configuring δ'' to γ' . Now we just

need to concatenate all those reconfiguration sequences and set $\gamma(v) = \gamma'(v)$ for $v \in G'$. This concatenation yields a valid reconfiguration sequence because colour c is connected to all colours from H' .

To build the final reconfiguration sequence we used the construction from Lemma 24 exactly k times. Apart from that, each vertex is recoloured at most once, to its final colour, so in total we used at most $n + n \cdot \max(2, k)^{\mathcal{O}(k^2\sqrt{n})} \leq \max(2, k)^{\mathcal{O}(\max(k^2, 1)\sqrt{n})}$ recolourings, as desired. $\square$

It is worth noting that all parts of this proof were constructive, hence we can compute such sequence of recolourings in $k^{\mathcal{O}(k^2\sqrt{n})}$ time and polynomial space.