

The Pennsylvania State University
The Graduate School

**FPT ALGORITHMS FOR NP-COMPLETE GRAPH PROBLEMS
PARAMETERIZED ON TREEWIDTH AND TREE-DEPTH**

A Dissertation in
Computer Science and Engineering
by
Mahdi Belbasi

© 2022 Mahdi Belbasi

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2022

The dissertation of Mahdi Belbasi was reviewed and approved by the following:

Martin Fürer

Professor of Computer Science and Engineering

Dissertation Advisor

Chair of Committee

Antonio Blanca

Assistant Professor of Computer Science and Engineering

Paul Medvedev

Associate Professor of Computer Science and Engineering

Associate Professor of Biochemistry and Molecular Biology

Jan Reimann

Associate Professor of Mathematics

Chitaranjan Das

Department Head of Computer Science and Engineering

Abstract

Many important graph problems are NP-complete. The complexity analysis relies on the worst-case scenario. However, most of the times the input has a nice “structure” so that we can solve the problem much faster, compared to the other inputs of the same size. In parameterized complexity, we classify the problems finer than the traditional setting. In order to do so, we define the complexity as a function of the set of the (input or output) parameters and the size of the input itself rather than as a function of the size of the input alone.

Fixed-parameter tractable problems are the problems that can be solved in time $\mathcal{O}(f(k)\text{poly}(n))$, where $f(\cdot)$ is a computable function, k is the parameter, and n is the input size. If the parameter is fixed, then the problem becomes tractable since the dependence on n is only polynomial.

Many NP-complete problems are known to be FPT. This means, if the parameter is bounded, then we can solve them relatively fast, which is the case in many applications. Courcelle’s metatheorem states that any graph property which can be described in the monadic second-order logic of the graphs can be decided in linear time if the treewidth (the parameter in this case) is bounded.

There are three concerns that we address. First, we need to prove that the treewidth is bounded. The bad news is that the treewidth problem itself is NP-complete but FPT. Then, we have to find a tree decomposition with bounded treewidth to be able to apply dynamic programming. Finally, when we apply dynamic programming, most of the time we end up using exponential space.

In this work, we solve all the above issues and start by presenting a 5-approximation algorithm solving the treewidth problem in time $\mathcal{O}(2^{8.765k}n \log n)$, which can be used in application¹. Later on we further improve this result by introducing new notion called “leftmost separator”. The improved algorithm runs in time $\mathcal{O}(2^{6.755k}(\log k)\sqrt{kn} \log n)$. Both of our algorithms check if the treewidth is at most k . If not, they output the subgraph that has higher treewidth. Otherwise, they generate a tree decomposition with width at most $5k$. In the last paper on this problem, we provide an algorithm that is a 2-approximation of treewidth with the time complexity of $\mathcal{O}^*(2^{8k})n$, which is state of the art. This method is based on a method employed by Korhonen [1].

Then, we present FPT algorithms to count the number of solutions to Minimum Dominating Set, Hamiltonian Cycle, and Travelling Salesperson which use only polynomial

¹The current fast FPT algorithms have huge coefficient for k in the exponent that makes them not applicable in practice.

space rather than exponential by an approach called “dynamic algebraization”.

Table of Contents

List of Figures	viii
List of Tables	x
List of Symbols	xi
Acknowledgments	xii
Chapter 1	
Introduction	1
1.1 Notation and Background	1
1.1.1 Tree Decomposition	2
Chapter 2	
An $\mathcal{O}^*(2^{7.61k})n \log n$ -time	
5-Approximation of Treewidth; An Improvement Of Reed's	
Treewidth Approximation	5
2.1 Introduction	5
2.1.1 Previously Known Results	6
2.1.2 Our Contribution	6
2.2 Preliminaries	8
2.2.1 Tree Decomposition	8
2.3 Analysis of Reed's Algorithm	8
2.3.1 Summary of Reed's Algorithm	9
2.3.2 Centroids and Separators	11
2.3.3 Algorithm to Find a Weakly Balanced Separation	13
2.3.3.1 Split(G, X, Y, k)	14
2.3.3.2 DFS-Trees(G, k)	14
2.3.3.3 Separation(G, k)	14
2.3.4 The Correctness of Reed's Algorithm	14
2.3.5 Running Time of Reed's Algorithm	17
2.4 Our Improved Algorithm	18
2.4.1 Relax the balancing requirement	18
2.4.2 Main Improvement	19

2.4.3	The Correctness of Our Algorithm	21
2.5	Running Time of Our Algorithm	26
2.6	Conclusion	31
Chapter 3		
	Finding All Leftmost Separators of Size $\leq k$	32
3.1	Introduction	32
3.1.1	Notation	32
3.1.2	Our Contributions	37
3.2	Finding the Leftmost Minimum Size $(X, Y, \leq k)^G$ -Separator	37
3.3	Finding All Minimal Leftmost $(X, Y, \leq k)$ -Separators	41
3.4	Application to Treewidth Approximation	48
3.4.1	Summary of Reed's algorithm	50
3.4.2	Further Previous Improvement	52
3.4.3	Our Improvement	53
Chapter 4		
	Faster 2-Approximation of the Treewidth	57
4.1	Introduction	58
4.2	A Summary of Korhonen's Algorithm	61
4.2.1	Splitting a Splittable Bag	63
4.2.1.1	Finding a Minimum Split	63
4.2.1.2	Performing the Split	64
4.3	Dynamic programming	66
4.3.1	The Details of the Dynamic Programming Algorithm	71
4.4	Our Improved $\mathcal{O}^*(2^{8k}) n$ -time 2-Approximation Algorithm	73
4.4.1	Runtime	77
4.5	Conclusion	78
Chapter 5		
	Saving Space by Dynamic Algebraization Based on Tree Decomposition: Minimum Dominating Set	80
5.1	Introduction	80
5.1.1	Previous Work	83
5.2	Notations	83
5.3	Saving Space Using Zeta and Möbius Transforms	86
5.3.1	Counting the dominating sets of all sizes	87
5.3.2	Finding The Minimum Dominating Set	91
5.4	Analyzing Time and Space Complexity of the Algorithm	94
5.5	Extension	95
5.6	Conclusion	96

Chapter 6	
Space-Efficient Parameterized Algorithms for Counting the Number of solutions to Hamiltonian cycles, and Traveling Salesper- son by Dynamic Algebraization	97
6.1 Preliminaries	100
6.1.1 Algebraic tools to save space	100
6.2 Counting the number of Hamiltonian cycles	102
6.2.1 Computing $f_x[X]$	104
6.2.2 Computing $\zeta f_x[X]$	105
6.3 The traveling Salesman problem	110
6.3.1 Computing the Hamiltonian Cycles of All Costs	110
6.3.2 Conclusion	111
Chapter 7	
Conclusion	112
Bibliography	113

List of Figures

1.1	Graph $G = (V, E)$ is a graph with vertices $\{a, b, c, d, e, f, g, h, i\}$ and τ_1 and τ_2 are two tree decompositions of G with width 3 and 4, respectively. Notice that in both decompsotions all the vertices are present. Furthermore, in both of them, endpoints of every edge are present in at least one bag (for instance look at the edge $\{g, h\}$ with red color). Also, the subgraph induced on every vertex of G is a subtree in both cases; check the green and purple subtrees corresponding to vertices b and c , respectively. Here $tw(G) = 4$ since there is a clique induced by $\{a, b, c, d\}$ and henceforth τ_1 is an optimal tree decompositon while τ_2 is not.	3
2.1	For a given graph G , let $V(G)$ be the set of its vertices. Let $G_6 = G[V(G_3) \cup V(G_4) \cup V(G_5) \cup \{c\}]$ (the entire rightmost green subtree rooted at c), and assume $ V(G_1) , V(G_2) $, and $ V(G_6) \geq \frac{n}{24k}$, and $ V(G_3) , V(G_4) $, and $ V(G_5) < \frac{n}{24k}$. Here, d, e , and f are NOT representatives but a, b , and c are.	9
2.2	Error, i.e. part of a subtree not on the side of its representative	10
2.3	How to fix the cases after a representative goes to the separator. Here, v and q are representatives and v is going to be sent into the separator. . .	20
3.1	Here $k = 5$. Separators S_1, S_2, S_3 , and S are all important but only S is leftmost	34
3.2	Augmenting Paths	39
3.3	Leftmost separators vs leftmost minimum size separator	40
3.4	Here, $k = 6$. We illustrate two out of 42 minimal leftmost $(X, Y, \leq 6)^{\hat{G}}$ -separators. Also, there are $1 + 2 + 5 + 14 + 42 = 64$ important $(X, Y, \leq 6)^{\hat{G}}$ -separators.	47

4.1	Neighbor-reduced tree decomposition before adding the intersection nodes and converting it into a grouped tree decomposition	68
4.2	Intersection node	68
4.3	Grouped Tree Decomposition	70
6.1	$H[V_x]$ is a set of paths with endpoints in B_x	103
6.2	Forget node x forgetting vertex v with the child c	105
6.3	Join node x with two children c_1 , and c_2	106

List of Tables

2.1	The history of previous algorithm for the treewidth approximation problem	7
4.1	An extended history of constant-factor FPT approximation of treewidth .	61
5.1	Previous work done on MDS, with a focus on time-space trade-off.	83

List of Symbols

G	Graph
$\mathcal{T}$	Tree decomposition
k	The width of a tree decomposition
tw	The treewidth
d	The depth of a tree decomposition
td	The tree-depth
$\mathcal{O}$	Capital Greek letter Omicron (big-O) to denote the upper bound on asymptotic growth
$\mathcal{O}^*$	The upper bound on asymptotic growth embedding polynomial parts
Θ	Greek letter theta to denote the exact asymptotic growth

Acknowledgments

First, I want to thank my advisor, Martin Fürer, who has had a significant impact on my development as a researcher. His great efforts and enthusiasm towards the subject continue to inspire me, and I could not have completed this work without his guidance. Martin Fürer is an amazing advisor, teacher, and human whose care and genuine support made this possible. I would also like to thank my committee members, Antonio Blanca, Paul Medvedev, and Jan Reimann, who provided thoughtful feedback and support throughout my graduate experience. I am especially grateful for summer research opportunities with Paul Medvedev and Antonio Blanca that introduced me to new applications in the field. I am equally grateful for Jan Reimann's constructive input on my thesis.

I am also thankful for the loving support of my family in Iran. I greatly value video chats with my parents and sisters each week, who provide relentless support and motivation. Their continued encouragement was essential to the completion of my thesis.

Lastly, I want to thank everyone else who has helped me throughout these years. I hope you know that I will never forget your help.

Dedication

I want to dedicate this thesis to my late sister, Mansoureh, who tragically lost her life to medical malpractice in 2017. She donated her organs to save four lives. I love you, sister.

Chapter 1 |

Introduction

1.1 Notation and Background

In this section, we review some terminology and define some new ones. Furthermore, we include a survey on the previous results. First, we review main notation of the algorithm design and analysis as well as graph theory.

Throughout this entire document, by a graph we refer to an undirected graph, unless specified otherwise.

Definition 1.1.1. Let $\mathcal{U}$ be an unbounded subset of $\mathbb{R}^+$ and $f : \mathcal{U} \mapsto \mathbb{C}$. Furthermore, let $N_0 \in \mathbb{N}$ and $g : \mathcal{U} \mapsto \mathbb{R}$ such that $g(n) > 0$, for all $n \geq N_0$. We say:

- $f(n) = o(g(n))$ if for any $c > 0$, there exists $N_c \in \mathbb{N}$ such that $|f(n)| < cg(n)$ for all $n > N_c$. i.e. $\lim_{n \rightarrow \infty} \frac{|f(n)|}{g(n)} = 0$.
- $f(n) = \mathcal{O}(g(n))$ if there exist $c > 0$ and $N_c \in \mathbb{N}$ such that $|f(n)| \leq cg(n)$ for all $n > N_c$. i.e. $\lim_{n \rightarrow \infty} \frac{|f(n)|}{g(n)} < \infty$.
- $f(n) = \omega(g(n))$ if for any $c > 0$, there exists $N_c \in \mathbb{N}$ such that $|f(n)| > cg(n)$ for all $n > N_c$. i.e. $\lim_{n \rightarrow \infty} \frac{|f(n)|}{g(n)} = \infty$.
- $f(n) = \Omega(g(n))$ if there exist $c > 0$ and $N_c \in \mathbb{N}$ such that $|f(n)| \geq cg(n)$ for all $n > N_c$. i.e. $\lim_{n \rightarrow \infty} \frac{|f(n)|}{g(n)} > 0$.
- $f(n) = \Theta(g(n))$ if $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$ ¹.

For more details, refer to [3].

¹Based on Knuth's notation [2]

1.1.1 Tree Decomposition

Tree decomposition is a data structure to decompose a given graph as a tree, where every node of the decomposed tree is a subset of the vertices of the original graph. Often times, the graph problems that are NP-complete are easy to solve when the input is a tree or “close” to a tree. Tree decomposition enables us to formally define what does this closeness mean and how to measure it. For that purpose the notion of treewidth is introduced.

Definition 1.1.2. A *tree decomposition* of a graph $G = (V, E)$ is a tree $\mathcal{T} = (V_{\mathcal{T}}, E_{\mathcal{T}})$ such that each node x in $V_{\mathcal{T}}$ is associated with a set B_x (called the bag of x) of vertices in G , and $\mathcal{T}$ has the following properties:

- $\bigcup_{x \in V_{\mathcal{T}}} B_x = V(G)$
- $\forall \{u, v\} \in E, \exists x \in V_{\mathcal{T}} : u, v \in B_x$
- $\forall x, y \in V_{\mathcal{T}}, \forall z \in V_{\mathcal{T}}$ on the path connecting $x, y \in V_{\mathcal{T}}: B_x \cap B_y \subseteq B_z$.

Historically, the *width* of tree decomposition $\mathcal{T}$ is defined to be the size of its largest bag minus one and is denoted by $w(\mathcal{T})$. The reason for deducting one, which is arbitrary, is to ensure that the width of every tree is 1.

Definition 1.1.3. The *treewidth* of a given graph G is the minimum width of all possible tree decompositions of G and is denoted by $tw(G)$. i.e.,

$$tw(G) = \min_{\mathcal{T}: \text{ a tree decomposition of } G} \{w(\mathcal{T})\}.$$

Before moving on, let us mention few famous remarks without the proof.

Remark 1.1.1. The connected graph G has treewidth 1 iff G is a tree.

Remark 1.1.2. Every graph with treewidth k has a vertex of degree $\leq k$.

Remark 1.1.3. Graph G with n vertices has treewidth $n - 1$ if G is a complete graph.

Figure 1.1 shows a graph and few of its possible tree decompositions.

Notice that in Figure 1.1, the treewidth of G should be at least 4 per Remark 1.1.3 and τ_1 has width 4. Hence, τ_1 is an optimal tree decomposition of G .

Courcelle’s metatheorem states that many graph properties are “easy” to decide when the input graphs has bounded treewidth.

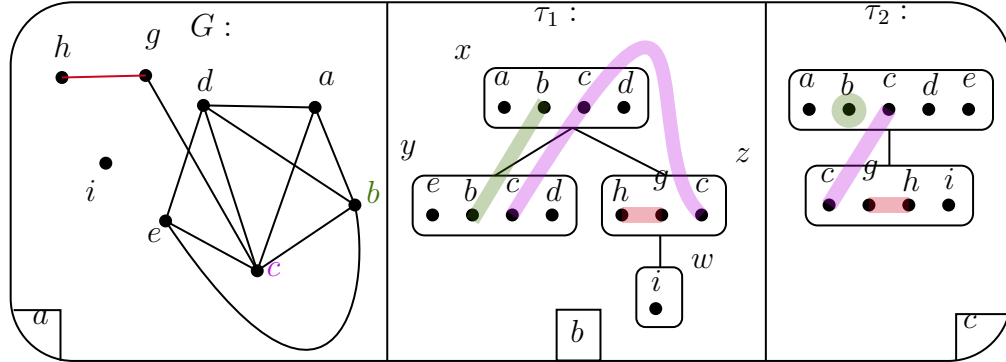


Figure 1.1. Graph $G = (V, E)$ is a graph with vertices $\{a, b, c, d, e, f, g, h, i\}$ and τ_1 and τ_2 are two tree decompositions of G with width 3 and 4, respectively. Notice that in both decompositions all the vertices are present. Furthermore, in both of them, endpoints of every edge are present in at least one bag (for instance look at the edge $\{g, h\}$ with red color). Also, the subgraph induced on every vertex of G is a subtree in both cases; check the green and purple subtrees corresponding to vertices b and c , respectively. Here $tw(G) = 4$ since there is a clique induced by $\{a, b, c, d\}$ and henceforth τ_1 is an optimal tree decomposition while τ_2 is not.

Theorem 1.1.1 (Courcelle '90 [4]). Given any graph property $P(\cdot)$ that is definable in the monadic second-order (MSO_1) logic of graphs, there a Turing machine M deciding P in linear time if the input graph has bounded treewidth.

For the definition of MSO_1 and more details refer to [5].

Courcelle's metatheorem is why finding an optimal or near-optimal tree decomposition is important. First, we have to check if the treewidth is bounded. Then, if that is the case, we need to find an optimal tree decomposition to actually decide those MSO_1 -definable graph properties by using dynamic programming. However, there are two issues that arise here:

- **First Issue:** Checking whether the treewidth is bounded and finding an optimal tree decomposition are both proven to be NP-complete problems [6].
- **Second Issue:** Even if we assume access to an optimal tree decomposition, applying dynamic programming of tree decompositions often results in exponential space complexity, in terms of the treewidth. Quoting from Woeginger: "Algorithms with exponential space complexities are absolutely useless for real life applications" [7].

In this document, we focused on addressing these issues and coming up with better solutions to fix them as much as possible. In Sections 2, 3, and 4 we focus on the first issue and in Sections 5 and 6 we explain approaches to fix the second issue.

To address the first issue and to fix it, my collaborator(s) and we have a series of efficient Fixed Parameter Tractable (FPT) algorithms solving treewidth approximation. Even though the exact treewidth problem is NP-complete, the constant approximation is in FPT class and can be solved very faster.

Definition 1.1.4. Problem A is said to be *FPT-solvable* if given input I of size n and a parameter k , there exist an algorithm solving A on input I in time $f(k)n^{\mathcal{O}(1)}$, where $f(\cdot)$ is a (Turing²) computable function. Furthermore, such an algorithm is called an *FPT algorithm*.

²A Turing machine can compute it in finite time.

Chapter 2 |

An $\mathcal{O}^* (2^{7.61k}) n \log n$ -time 5-Approximation of Treewidth; An Improvement Of Reed's Treewidth Approximation

This chapter is based on a joint work with Martin Fürer [8].

2.1 Introduction

Since the 1970s and early 1980s, when the notions of treewidth and tree decomposition were introduced [9–11], they have played important roles in computer science [12]. In a nutshell, treewidth is a parameter of a graph that measures how similar it is to a tree. One of the main reasons that the tree decomposition is widely studied is that many NP-complete problems have efficient algorithms for graphs with small treewidth. In fact, Courcelle's metatheorem states that every graph property definable in monadic second-order logic of graphs can be solved in linear time on graphs of bounded treewidth [4]. The first step of solving such problems is to find an optimal or near-optimal tree decomposition. However, finding an optimal tree decomposition itself is NP-hard [6]. In this work, we propose an algorithm which is based on Reed's algorithm [13] to approximate the treewidth and find an approximately optimal tree decomposition.

Definition 2.1.1. A graph problem is *fixed parameter tractable (FPT)* if it can be solved in time $\mathcal{O}(f(k) n^{\mathcal{O}(1)})$, where f is a computable function, k is a parameter of the graph, and n is the input graph size.

2.1.1 Previously Known Results

In this work, we are interested in algorithms which run fast (polynomial in terms of the number of vertices) for graphs with bounded treewidth. One of the first algorithms given for this problem goes back to the same paper where treewidth has been shown to be NP-complete. Arnborg et al. [6] gave an algorithm which runs in time $\mathcal{O}(n^{k+2})$. In 1995, Robertson and Seymour [14] gave a quadratic time FPT approximation algorithm. Later on, Lagergren introduced an 8-approximation algorithm with the time complexity of $2^{\mathcal{O}(k \log k)} n \log^2 n$ [15]. In 1992, Reed [13] improved these algorithms to have an algorithm running in time $2^{\mathcal{O}(k \log k)} n \log n$. In this paper, we formally show that the approximation ratio of Reed's algorithm is 7 or 5, depending on the frequency of the split by volume. We show that this algorithm runs in time $\mathcal{O}(2^{24k} k! n \log n)$, in order to be able to compare it to our algorithm. Like the algorithms of Robertson and Seymour, Reed's algorithm is recursive. In [14], they find a separator that partitions G into two parts but they do not force the separator to partition the entire graph in a balanced fashion. Reed finds a separator which partitions the graph in a balanced way to obtain time $\mathcal{O}(n \log n)$ for bounded k . This paper focuses on this algorithm. Later, Bodlaender gave an exact algorithm which runs in $2^{\mathcal{O}(k^3)} n$ [16]. Even though we focus only on constant-factor approximation algorithms, it is worth mentioning the $(\log k)$ -approximation algorithm by Amir [17] and the $\sqrt{\log k}$ -approximation algorithm by Feige et al. [18]. Later in 2016, Bodlaender et al. [19] gave two constant factor approximation algorithms which run in $2^{\mathcal{O}(k)} \mathcal{O}(n \log n)$ and $2^{\mathcal{O}(k)} \mathcal{O}(n)$. The former one is a 3-approximation and the latter one is a 5-approximation. Although it is a great result from a theoretical point of view, it uses a sophisticated data structure and the constant factor hidden in $\mathcal{O}(k)$ in the exponent is not claimed to be practical. Very recently, Korhonen gave a 2-approximation algorithm for the same problem running in time $2^{\mathcal{O}(k)} n$ [1]. They improve the hidden coefficient of k in the exponent. In their algorithm, the coefficient of k in the exponent is 10.7549, in the worst-case scenario. Here, we sacrifice the dependence on n and drop the coefficient of k in the exponent to only 7.61. Table 4.1 summarizes the history of previously known algorithm for treewidth problem.

2.1.2 Our Contribution

First in Section 2.3, we analyze Reed's algorithm [13] in detail. Reed has focused on the dependence on n because his goal was to come up with an $\mathcal{O}(n \log n)$ -time algorithm,

Reference	Approximation Ratio	Dependence on k	Dependence on n	Comments
Arnborg et al. (1987) [6]	1	$\mathcal{O}(n^{k+2})$		not FPT
Robertson & Seymour (1995) [14]	4	$\mathcal{O}(3^{3k})$	n^2	
Lagergren (1996) [15]	8	$2^{\mathcal{O}(k \log k)}$	$n \log^2 n$	
Reed (1992) [13]	5 (or 7)	$2^{24k} k!$ (Sec. 2.3)	$\mathcal{O}(n \log n)$	
Bodlaender (1996) [16]	1			
Feige et al. (2008) [18]				not a constant-factor approximation
Amir (2010) [17]	$\mathcal{O}(\log k)$	$\mathcal{O}(k \log k)$	n^4	not a constant-factor approximation
	4	$\mathcal{O}(2^{4.38k} k)$	n^2	
	4.5	$\mathcal{O}(2^{3k} k^{1.5})$	n^2	
	$\frac{11}{3}$	$\mathcal{O}(2^{3.6982k} k^3)$	$n^3 \log^4 n$	
Fomin (2015) [20]	1	$\mathcal{O}(1)$	1.7347^n	not FPT
Bodlaender et al. (2016) [19]	3	$2^{\mathcal{O}(k)}$	$n \log n$	the coefficients of k are not stated
	5	$2^{\mathcal{O}(k)}$	n	
Belbasi & Fürer (2021) [21]	5	$2^{8.766k}$	$n \log n$	improved to $2^{7.61k} n$
Korhonen (2021) [1]	2	$2^{10.7549k}$	n	relatively low coefficient of k in the exponent, good approximation ratio, and only linear
Belbasi & Fürer (2021) [22]	5	$2^{6.755k}$	$n \log n$	extra $\log n$ in the running time compared to [1] but has smaller coefficient of k in the exponent
Belbasi, Fürer & Kumar (2022)	2	$\mathcal{O}^*(2^{8k})$	n	state of the art

Table 2.1. The history of previous algorithm for the treewidth approximation problem

for fixed k . The “fastest”¹ algorithm at that time, was Lagergren’s $\mathcal{O}(n \log^2 n)$ -time algorithm [15]. We show that the dependence on k in Reed’s algorithm is $2^{\mathcal{O}(k \log k)}$. Furthermore, we give a proof for the approximation ratio of Reed’s algorithm by filling in the details.

Then, we propose two improvements and prove that the approximation ratio stays at 5. One of our improvements focuses on the notion of “balanced split”. We call a split balanced, if we get two parts of volume $1 - \epsilon$ and ϵ (or better). Then, the running time of our algorithm has a factor of $1/\epsilon$. For instance, if we set $\epsilon = \frac{1}{100}$, a generous estimation shows that the dependence on k in our $\mathcal{O}(f(k)n \log n)$ -time algorithm is $k^2 2^{8.87k}$, instead

¹Considering the dependence on both n and k .

of $2^{24k}(k+1)!$ in Reed's algorithm. Here the asymptotic notation is a bit misleading from a practical point of view, as $2^{24k} = o(k!)$, even though $k!$ is reasonable for small k , while 2^{24k} is not. Then, we further improve the running time upper bound and show that our algorithm runs in time $\mathcal{O}(2^{7.61k}k^2n \log n)$. In the end, the main aim of this paper is to produce a constant-factor approximation algorithm that runs in time $2^{ck}n \log n$ with c as small as possible.

2.2 Preliminaries

2.2.1 Tree Decomposition

Definition 2.2.1. A *tree decomposition* of a graph $G = (V, E)$, is a tree $\mathcal{T} = (V_{\mathcal{T}}, E_{\mathcal{T}})$ such that each node x in $V_{\mathcal{T}}$ is associated with a set B_x (called the bag of x) of vertices in G , and such that $\mathcal{T}$ has the following properties:

- The union of all bags is equal to V . In other words, for each $v \in V$, there exists at least one node $x \in V_{\mathcal{T}}$ with B_x containing v .
- For every edge $\{u, v\} \in E$, there exists a node x such that $u, v \in B_x$.
- For any nodes $x, y \in V_{\mathcal{T}}$, and any node $z \in V_{\mathcal{T}}$ belonging to the path connecting x and y in $\mathcal{T}$, $B_x \cap B_y \subseteq B_z$.

In this paper we use a variation of tree decomposition where the adjacent bags differ in at most one vertex (converting can happen in linear time).

The *width of a tree decomposition* is the size of its largest bag minus one. The *treewidth* of a graph G is the minimum width over all tree decompositions of G called $tw(G)$. Observe that the treewidth of a tree is 1. In the following, we reserve the letter k for the $tw(G)+1$.

We have to mention that Bodlaender et al. [19] filled in some details on Reed's algorithm. We need to be more detailed because we do not use Reed's algorithm as a black box. That is why we first analyze Reed's algorithm precisely (Section 2.3), fill in the blanks, and then introduce our improvements of his algorithm (Section 2.4).

2.3 Analysis of Reed's Algorithm

In 1992, Reed gave an elegant algorithm [13] to either construct a tree decomposition of width at most $7k$ or $5k$ of a given graph G , or declare that the treewidth is greater than

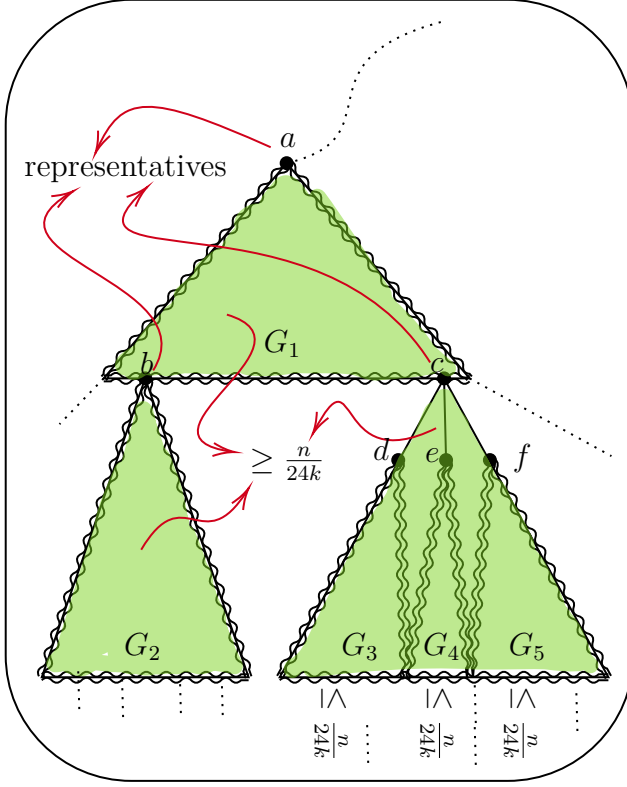


Figure 2.1. For a given graph G , let $V(G)$ be the set of its vertices. Let $G_6 = G[V(G_3) \cup V(G_4) \cup V(G_5) \cup \{c\}]$ (the entire rightmost green subtree rooted at c), and assume $|V(G_1)|, |V(G_2)|$, and $|V(G_6)| \geq \frac{n}{24k}$, and $|V(G_3)|, |V(G_4)|$, and $|V(G_5)| < \frac{n}{24k}$. Here, d, e , and f are NOT representatives but a, b , and c are.

k and output a subgraph which is a bottleneck (no separator of size $\leq k$).

2.3.1 Summary of Reed's Algorithm

In Reed's algorithm, one of the main tasks is to find a “balanced” separator S that splits the graph $G - S$ into two subgraphs with sets of vertices $X, Y \subseteq V(G)$. Once a balanced separator is found, the algorithm recursively finds a tree decomposition for $G[X \cup S]$ (the subgraph induced by $X \cup S$) and $G[Y \cup S]$.

The main task is to find a balanced separator. Instead of branching on every vertex (going to X, Y , or S , which would be exponential in n), Reed forms groups of vertices and

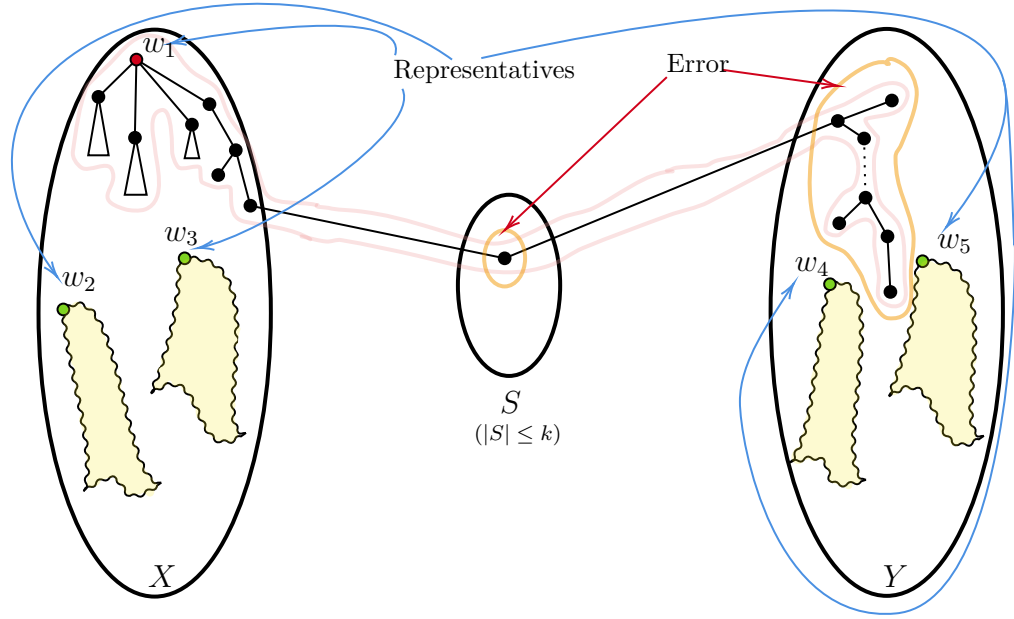


Figure 2.2. Error, i.e. part of a subtree not on the side of its representative

works with the representatives of the groups. Then, he branches on the representatives. Reed does a DFS and finds the deepest vertex v whose subtree has at least $\frac{n}{24k}$ vertices² (See Figure 2.1). We call such a vertex a “representative”. He defines the weight of the representative v as the size of its subtree rooted at v in the DFS, denoted by $w(v)$. The idea here, is that if a representative goes to either X or Y , most of its descendants will go to the same set. The reason is that if a descendant goes to the other side, the path connecting the representative to the descendant has at least one vertex in the separator. However, we know that the separator cannot have more than k vertices. So, not many vertices will go to the wrong set; not more than $\frac{n}{24}$ vertices, in total. This is because every subtree that partially goes to the other side should go through the separator and have one vertex there. Hence, not more than k subtrees rooted at children of the representatives can go through the separator, which results in at most $\frac{n}{24}$ vertices on the wrong side. This nice property allows Reed to work with the set of representatives (which is much smaller) rather than all the vertices (see Figure 2.2).

Now, one might think, why not just check all the possibilities of the representatives going to X, Y , or S ? The reason that this simple idea does not work is that if a representative

²later, we talk about this threshold

goes to the separator, its entire subtree of arbitrary size can go anywhere and we do not have any control over it. Reed handles this problem by deciding whether any representative is going to the separator, at the very beginning of the algorithm. If so, he just places such a representative (namely v) into S (and not its subtree) and starts forming a new group of representatives by running a new DFS on $G - \{v\}$. So, the other representatives might change. Also, since one vertex has been placed into the separator, now $k \leftarrow k - 1$. However, if none of the representatives goes to the separator, he branches on placing them left (X) or right (Y). This is the high-level idea of Reed's algorithm. Let's start with presenting and reviewing some definitions.

2.3.2 Centroids and Separators

For an undirected graph $G = (V, E)$ and a subset W of the vertices, $G[W]$ is the subgraph induced by W . For the sake of simplicity throughout this paper, let $G - W$ be $G[V \setminus W]$ and $G - v$ be $G - \{v\}$ for any $W \subseteq V(G)$ and any $v \in V(G)$.

Also, in a weighted graph, a non-negative integer weight $w(v)$ is defined for each vertex v . For a subset W of the vertices, the weight $w(W)$ is simply the sum of the weights of all vertices in W . Furthermore, the total weight or the weight of G is the weight of V .

Definition 2.3.1. A centroid of a weighted tree T is a node x such that none of the trees in the forest $T - x$ has more than half the total weight.

Definition 2.3.2. A tree decomposition is called a *good tree decomposition* if the adjacent bags differ in at most one vertex.

For good tree decompositions we choose a stronger version of centroid.

Lemma 2.3.1. Any given tree decomposition $\mathcal{T}$ with width k can be converted to an equivalent good tree decomposition $\mathcal{T}^*$ with width k of the same graph in linear time.

Proof. Let x and y be two adjacent nodes of $\mathcal{T}$ with associated bags B_x , and B_y , respectively.

Let $B_x = \{v_1, \dots, v_t\}$, and $B_y = \{u_1, \dots, u_s\}$. Notice that $s, t \leq k$.

Now, let $V^- = B_x \setminus B_y$, and $U^+ = B_y \setminus B_x$. Let $D = V^- \cup U^+ = \{d_1, \dots, d_{|V^-|}, d'_1, \dots, d'_{|U^+|}\}$.

The idea is to add $|D|$ intermediate nodes (namely $x_1, x_2, \dots, x_{|V^-|}, y_1, y_2, \dots, y_{|U^+|}$) between x and y such that we start by deleting vertices of V^- one by one and then adding vertices of U^+ one at a time.

Set

$$\begin{cases} B_{x_1} = B_x \setminus \{d_1\} \\ B_{x_i} = B_{x_{i-1}} \setminus \{d_i\} & \forall i \in \{2, \dots, |V^-|\} \\ B_{y_1} = B_{x_{|V^-|}} \cup \{d'_1\} \\ B_{y_i} = B_{y_{i-1}} \cup \{d'_i\} & \forall i \in \{2, \dots, |U^+|\}. \end{cases} \quad (2.1)$$

□

Definition 2.3.3. A strong centroid of a good tree decomposition τ of a graph $G = (V, E)$ with respect to $W \subseteq V$ is a node x of τ such that none of the connected components of $G - B_x$ contains more than $\frac{1}{2}|W \setminus B_x|$ vertices of W .

The following lemma shows the existence of a strong centroid for any given $W \subseteq V$.

Lemma 2.3.2. For every good tree decomposition $(\mathcal{T}, \{B_x : x \in V_{\mathcal{T}}\})$ of a graph $G = (V, E)$ and every subset $W \subseteq V$, there exist a strong centroid with respect to W .

Proof. If a node x is not a strong centroid with respect to W , then let C_x be the set of vertices in the unique connected component of $G - B_x$ containing more than $\frac{1}{2}|W \setminus B_x|$ vertices of X . In the forest obtained from the tree $\mathcal{T}$ by removing x , there is a tree $\mathcal{T}_x$ with the property that the union of all bags in $\mathcal{T}_x$ contains all the vertices of C_x . Now, we define a set F of directed tree edges by $(x, y) \in F$ if all the following conditions hold:

- x is not a centroid
- y is a neighbor of x in $\mathcal{T}$,
- y is a node in $\mathcal{T}_x$.

Now we show that there is a node x with out-degree 0 in $(V_{\mathcal{T}}, F)$. Such an x is a centroid, and we are done. Otherwise, F contains (x, y) and (y, x) for some $x, y \in V_{\mathcal{T}}$. W.l.o.g., $B_y = B_x \cup \{v\}$ for some $v \in V \setminus B_x$. Note that $\mathcal{T}_x$ and $\mathcal{T}_y$ are disjoint. Furthermore, any vertex that is in a bag of $\mathcal{T}_x$ and in a bag of $\mathcal{T}_y$ is also in B_x and B_y . Thus also C_x and C_y are disjoint.

Furthermore, $W \cap C_x \subseteq W \setminus B_x$ and $W \cap C_y \subseteq W \setminus B_y \subseteq W \setminus B_x$. As $W \cap C_x$ and $W \cap C_y$ are disjoint, they cannot both have more than $\frac{1}{2}|W \setminus B_x|$ vertices, which is a contradiction.

Hence, there exists a node x which is a strong centroid. □

We use the definitions of balanced W -separator and weakly balanced W separation from the book of Flum and Grohe [23].

Definition 2.3.4. Let $G = (V, E)$ be a graph and $W \subseteq V$. A *balanced W -separator* is a set $S \subseteq V$ such that every connected component of $G - S$ has at most $\frac{1}{2}|W|$ vertices.

Lemma 2.3.3. [23, Lemma 11.16] Let $G = (V, E)$ be a graph of treewidth at most $k - 1$ and $W \subseteq V$. Then there exists a balanced W -separator of G of size at most k .

We say that a separator S separates $X \subseteq V$ from $Y \subseteq V$ if $C \cap X = \emptyset$ or $C \cap Y = \emptyset$ for every connected component C of $G - S$.

Definition 2.3.5. Let $G = (V, E)$ be a graph and $W \subseteq V$. A *weakly balanced separation* of W is a triple (X, S, Y) , where $X, Y \subseteq W$, $S \subseteq V$ are pairwise disjoint sets such that:

- $|S| \leq k$
- $W = X \cup (S \cap W) \cup Y$
- S separates X from Y
- $0 < |X|, |Y| \leq \frac{2}{3}|W|$.

Lemma 2.3.4. [23, Lemma 11.19] For $k \geq 3$, let $G = (V, E)$ be a graph of treewidth at most $k - 1$ and $W \subseteq V$ with $|W| \geq 2k + 1$. Then there exists a weakly balanced separation of W of size at most k .

Even though Lemma 3.4.3 is sufficient for us, one can make it stronger such that it holds for $|W| \geq k + 1$.

Theorem 2.3.1. [23, Corollary 11.22] For a graph of treewidth at most $k - 1$ with a given set $W \subseteq V$ of size $|W| = 3k - 2$, a weakly balanced separation of W of size $\leq k$ can be found in time $O(3^{3k}k^2n)$.

2.3.3 Algorithm to Find a Weakly Balanced Separation

$\text{Separation}(G, k)$ is the main part of Reed's algorithm. It finds a separator of size at most k in G using the procedures $\text{Split}(G, X, Y, k)$ and $\text{DFS-Trees}(G, k)$. We explain each of these procedures.

2.3.3.1 Split(G, X, Y, k)

For X, Y disjoint subsets of V , $\text{Split}(G, X, Y, k)$ finds a separator S of size at most k in G which is strictly between X and Y . Split reports failure if no such separator exists (described in Lemma 11.20 of [23]).

2.3.3.2 DFS-Trees(G, k)

$\text{DFS-Trees}(G, k)$ (Algorithm 1 using Algorithm 2) computes a DFS tree and partitions it into smaller DFS trees with the following properties.

- the size (number of vertices) of the smaller trees is at least $s = n/24k$, and
- all subtrees rooted at children of the roots of the trees in the partition have size less than $n/24k$.

$\text{DFS-Trees}(G, k)$ collects the set W' consisting of all the roots of the trees in the partition. These roots are representatives of the vertices in their small DFS tree. Therefore, the weight $w[v]$ for $v \in W'$ is the number of vertices in the small tree with root v .

2.3.3.3 Separation(G, k)

$\text{Separation}(G, k)$ is the recursive procedure that splits according to the number of vertices (Algorithm 3). Note that when any vertex v is placed into the separator S , then the procedure Separation removes that vertex v from the graph and starts from scratch. The idea is that when we place a root of a small tree (a representative) left or right, then we want to put the whole small tree there. But when a representative is placed into the separator, then its tree does not go there. At this point a new collection of trees is formed.

2.3.4 The Correctness of Reed's Algorithm

If the treewidth is at most $k - 1$, then there is a good tree decomposition of G of width $k - 1$. Let x be a centroid in it. The connected components of $G[V \setminus B_x]$ can be partitioned into 2 parts L and R , such that no part has more than $\frac{2}{3}|V|$ vertices.

Note that for the correctness proof, we do not have to find this tree decomposition. It is sufficient to know that it exists. We can assume, that we have fixed such a tree decomposition, a centroid x and the sets L and R .

Every $W' \subseteq V$ is partitioned into parts in L , the separator $S = B_x$, and R .

Algorithm 1 Construct Small DFS-Trees

Ensure: Roots of DFS-Trees (representatives) with sizes of their strict subtrees $< |V|/(24k)$

procedure DFS-TREES(G, k) $\triangleright G$ is a connected graph.

$s = \frac{|V|}{24k}$ $\triangleright s$: the size bound for splitting off a small tree.

$W' = \emptyset$

for all $v \in V$ **do**

color[v] = WHITE

end for

Pick any vertex u of G .

count = DFS-visit(G, u)

$w[x] = w[x] + \text{count}$, where x is the last representative added to W' **return** (W' , $w[v]$ for all $v \in W'$)

end procedure

Algorithm 2 Main recursive procedure of DFS-Trees

procedure DFS-VISIT(G, u)

color[u] = GRAY

count = 1

for all v adjacent to u **do**

if color[v] == WHITE $\triangleright$ The white vertex v has been discovered. **then**

count = count + DFS-Visit(G, v)

end if

end for

if count $\geq s$ **then**

$W' = W' \cup \{u\}$

$w[u] = \text{count}$

count = 0

end if return count

end procedure

One of the many branches of the procedure Separation(G, k) picks the set W' , tries this partition of it, and succeeds, unless a previously taken branch has already succeeded. First, the procedure decides which part of W' goes into S , one vertex v at a time. This vertex v is removed from G , but otherwise, we still consider the same tree decomposition. $|B_x|$ has now decreased by 1, as v is removed from it.

We then consider the case that none of the remaining vertices in W' are in the separator. The procedure decides which part $X \subseteq W'$ goes into L (line 12). Now the weight of each part is at most $(\frac{2}{3} + \frac{1}{24})n$ as at most k small subtrees can have some of their vertices on the wrong side. And this is at most k times at most $\frac{n}{24k}$ vertices.

Algorithm 3 Main recursive procedure in Reed's algorithm

Ensure: A weakly balanced separation (X, S, Y) of G of size $\leq k$

procedure SEPARATION(G, k)

if $k > 0$ **then**

$(W', w[v] : \forall v \in W') = \text{DFS-Trees}(G, k)$

end if

for all $v \in W'$ $\triangleright$ Here v is placed into separator S . **do**

$(X, S, Y) = \text{SEPARATION}(G - v, k - 1)$

if $\neg \text{failure}$ **then return** $(X, S \cup \{v\}, Y)$

end if

end for $\triangleright$ The set of vertices W' is partitioned into $X \subseteq L$ and $Y \subseteq R = W' \setminus L$.

for all $X \subseteq W'$ $\triangleright$ Here no vertex is put into S . **do**

if $(\frac{1}{3} - \frac{1}{24})n \leq w(X) \leq (\frac{2}{3} + \frac{1}{24})n$ **then**

 Split($G, X, W' \setminus X, k$)

if $\neg \text{failure}$ **then return** (X, S, Y)

end if

end if

end for **return** failure

end procedure

On the branch of the procedure Separation(G, k) which finds this partition of W' , there is the separator B_x of size at most k between X and Y . Our algorithm cannot guarantee to find this separator B_x , but it will find some separator S of size at most k between X and Y . Again up to $\frac{1}{24}n$ vertices can be on the opposite side of their representatives. Now the larger side can contain at most $(\frac{2}{3} + 2 \cdot \frac{1}{24})n = \frac{3}{4}n$ vertices. Thus we have a somewhat balanced partition (a constant fraction on each side).

We assume the $\mathcal{O}(3^{3k}k^2n^2)$ 4-approximation algorithm of Robertson-Seymour [14] (see Proposition 11.14 of Flum and Grohe [23]) is known. It handles a set $W \subseteq V$ of size $3k - 2$. Working with (G, V) , it finds a separator S of size at most k to split $G - S$ into two parts L and R with both, $|W \cap X|$ and $|W \cap Y|$ at most $\frac{2}{3}|W|$. A tree node with bag $W \cup S$ is formed. Two recursive calls continue with $G[L \cup S], (W \cap (L \cup S))$ and $G[R \cup S], (W \cap (R \cup S))$, respectively.

Reed's algorithm does the same steps to handle W . However, in order to decrease the dependence on n of the running time from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$, Reed intersperses these balanced partitions of W with balanced partitions of V (Algorithm 3). In each case, it would be desirable that V and W simultaneously split in a balanced way. However, during the traditional splitting of W , the graph might be split very unbalanced, and during the new splitting of V , the set W might not be split at all.

Reed's algorithm can alternate between splitting W as in the $\mathcal{O}(3^{3k}k^2n^2)$ algorithm and splitting V . Now W can be of size at most $6k$. On each side, we have at most $(\frac{2}{3} \cdot 6k) = 4k$. Splitting by W as well as splitting by V adds k to the new W . Thus, we are back at $6k$. The constructed tree decomposition has then width at most $7k$. But we show that this can be actually a 5-approximation algorithm. We do not need to alternate between splitting W and V . Splitting V is a costly procedure. We can do it only after every $\log_{\frac{3}{2}} k$ steps and we still spend $\mathcal{O}(n \log n)$ time.

We start with W of size at most $4k$ ($3k$ and k_{excess} as excess). Initially, $k_{\text{excess}} = k$. Each time we split W , we get

$$|W| \leq \frac{2}{3} \cdot 3k + \underbrace{k}_{\text{adding separator}} + \frac{2}{3}k_{\text{excess}} = 3k + \frac{2}{3}k_{\text{excess}},$$

and then update $k_{\text{excess}} \leftarrow \lfloor \frac{2}{3}k_{\text{excess}} \rfloor$. The excess drops by a factor of $\frac{2}{3}$. After $\log_{\frac{3}{2}} k$ steps, the excess becomes zero and then we can split by V . At this point, $|W|$ could increase to $4k$ again ($3k$ was the size of W before this step, and when we split by V , we have to include the separator as well). In the end, we end up with $|W| \leq 4k$ and we add the separator to the root bag, which means the largest bag has size at most $5k$. Therefore, it is a 5-approximation algorithm. Reed mentions $5k$ in his paper, but he does not give any details. We think based on what it has been described in Reed's algorithm, it seems we should have $7k$. However, if splitting by V does not happen very often, we can achieve $5k$.

2.3.5 Running Time of Reed's Algorithm

Let $T(n, k)$ be the running time of the procedure $SEPARATION(G, k)$ for $G = (V, E)$ and $n = |V|$. Let n' and k' be the current bound on the graph size and current separator capacity. Initially $n' = n$ and $k' = k$. We have the following recurrence for Reed's algorithm.

$$T(n', k') \leq 24k'T(n' - 1, k' - 1) + 2^{24k'} \underbrace{c(k' + 1)kn'}_{\text{flow algorithm}}, \quad (2.2)$$

for some $c > 0$. It is difficult to obtain a good solution, but by induction on k' we get the following loose upper bound.

$$T(n', k') \leq 3c2^{24k'}k'!kn. \quad (2.3)$$

For $k' = 0$, this bound is valid. For $k' \geq 1$, we have:

$$\begin{aligned}
T(n', k') &\leq 24k'T(n' - 1, k' - 1) + c2^{24k'}(k' + 1)kn' \\
&\leq 3c24k'2^{24(k'-1)}(k' - 1)!kn + c2^{24k'}(k' + 1)kn' \quad \text{by induction hypothesis} \\
&= c2^{24k'}kn\left(\frac{3 \cdot 24k'!}{2^{24}} + k' + 1\right) \leq 3c2^{24k'}knk'!\left(\frac{24}{2^{24}} + \frac{k' + 1}{3k'!}\right) \\
&\leq 3c2^{24k'}k'!kn.
\end{aligned}$$

Even though, this is not a tight bound, we have $T(n, k) \geq c'24^k k!(n - k)$, which is $2^{\Omega(k \log k)}n$.

$$\begin{aligned}
T(n, 0) &\geq c'n \\
T(n, k) &\geq 24kT(n - 1, k - 1) \\
&\geq c'24k24^{k-1}(k - 1)!(n - k) \text{ by inductive hypothesis} \\
&\geq c'24^k k!(n - k)
\end{aligned}$$

2.4 Our Improved Algorithm

In this section, we discuss how to improve Reed's algorithm. The dependence on k in the running time of Reed's algorithm is huge. We decrease this factor significantly to make the algorithm more applicable. We introduce two main modifications. First, we work with a larger cut-off threshold than Reed's $\frac{|V|}{24k}$. Such an improvement can be achieved by replacing the arbitrary $3/4$ bound by $1 - \epsilon$. But even more is possible by arguing about the weights of connected components instead of the weights of the parts of a bipartition.

The second improvement is to avoid branching on whether there is a representative going into the separator or not. Reed branches on these two cases at the beginning, while we branch 3-fold for every representative.

Note that the second improvement is not obvious. Reed had a good reason to avoid 3-fold branching. If a representative of tree is put into the separator, then, we lose control of the unbounded set of additional vertices of the tree.

2.4.1 Relax the balancing requirement

Reed's argument starts with a weakly balanced separation by volume (i.e. according to V) that is known to exist. The larger side has at most $2/3$ of the volume, but it might have up to $2/3 + 1/24$ of the weight. The algorithmic split by this weight partition

might pick a set with another $1/24$ fraction more volume. Thus the worst kind of volume split found is now $3/4$ to $1/4$. Recall that these differences are bounded this way for the following reason. When the weight carrying root of a tree is on one side, some of its small subtrees rooted at the children can be partially on the other side. But the separator prevents more than k small subtrees to have any part on a different side than the root, and each small subtree contains less than $\frac{|V|}{24k}$ vertices. Instead of $3/4$, one can choose any number strictly between $2/3$ and 1 . If $1 - \epsilon$ is chosen, then the constant 24 is replaced by $\frac{1}{((1-\epsilon)-2/3)/2} = \frac{6}{1-3\epsilon} \leq 6 + 24\epsilon$ for $\epsilon < 1/12$.

More improvement is possible by a modification of the analysis. We start with a restricted balanced separator by volume. No connected component of $G - S$ has more than half the vertices. We can afford it to contain up to a fraction of $3/4 - \epsilon/2$ of the weight. There is still a partition of $G - S$ into left and right with a ratio of at most $3/4 - \epsilon/2$ to $1/4 - \epsilon/2$ by weight. The separator found for such a partition by weight creates a partition by volume with a worst case ratio of $1 - \epsilon$ to ϵ . This time, the constant 24 is replaced by $\frac{1}{((1-\epsilon)-1/2)/2} = \frac{4}{1-2\epsilon} \leq 4 + 12\epsilon$ for $\epsilon < 1/6$.

2.4.2 Main Improvement

The other improvement is to allow the representatives (the roots of the subtrees) to go either, left, right, or into the separator. Once a representative v goes into the separator, we change its weight to 0 . We also delete v from G and unmark all of the vertices in its subtree, so that they can be searched again. Then, we continue the DFS from the parent of v . There is one complication here that we have to take care of; what happens if $G - v$ gets disconnected?

Three cases might happen. We cover all the cases and show that they can be handled. Let x, y , and z be three types of children of v with the small subtrees (with size $< \frac{n}{Ck}$) τ_x , τ_y , and τ_z , rooted respectively (Fig. 2.3(a)). The cases are:

- There is a back edge from a vertex in τ_x to an ancestor of v . If we delete v from the tree, the vertices of τ_x will still be searched because they are connected to an ancestor of v . So, we just need to color them white once more. Hence, we do not need to worry about this case.
- There is a vertex p in τ_y with subtree τ_q attached to it with q as its root such that q is a representative below τ_y which has not gone into the separator (this is a bottom-up approach). Even though this case seems to be troublesome, we can fix it. Let $\tau_{p,y}$ be τ_y rooted at p (dangling from p). Make p a child of q . Now, the

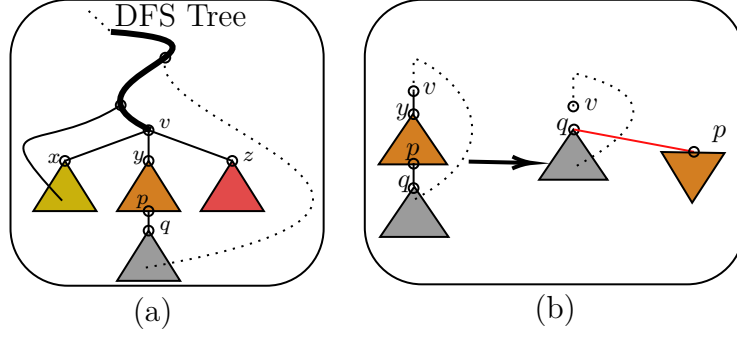


Figure 2.3. How to fix the cases after a representative goes to the separator. Here, v and q are representatives and v is going to be sent into the separator.

problem has been fixed (Fig. 2.3(b)). The same reasoning applies when v is a root and its subtree is too small.

Notice that in this case, it is possible that there is a back edge to some vertex higher up as well but it does not change anything here. We handle it by attaching to the subtree below it. One could handle it like the previous possibility. Both are fine.

- There is no back edge from τ_z to an ancestor of v , and there is no subtree below. So deleting v makes τ_z disconnected from the entire tree, and this is only for our advantage.

Now, the question is how to distinguish between these three cases?

We just need to distinguish between case 2 and the other two cases (together) because case 1 and 3 are handled the same way.

Cases 1 and 3 are actually the simple ones. We just need to color their vertices white and then we are done. So, we have to find a way to recognize case 2. For that, each vertex u in $\tau(y)$ saves a possible representative with an edge to some $p \in \tau(y)$ such that p is an ancestor of u (defined as *rep_connected* in the pseudocode). If no such representative exists, simply set its value to *nil*. Notice that we just need to store one representative connected down below since it is sufficient to handle this case as we just described. Each vertex passes that information to its parent. So, if *rep_connected*[y] $\neq \text{nil}$, we know that the case 2 has happened. Otherwise, case 1 or 3 has happened and we do not need to know which one since they are handled the same way. Now, let's go back to the algorithm description.

The main difference here is that Reed branches in the beginning and considers two cases. In the first case, none of the representatives goes to the separator, and in the second case at least one goes to the separator. In the second case, Reed's algorithm

branches into at most $24k$ (upper bound for the number of subtrees). We want to avoid these branches and each time only branch into three cases. Assume we want to decide where to put v (a representative with weight $w(v)$). Let L , S , and R be the left, the separator, and the right sets, respectively. If we put v into L (or R), usually most of the vertices in its subtree will be in L (or R) as well. In case v goes to S , we release the other vertices of its tree to be searched again (as we described above and also as it is mentioned in the pseudocode). We have to mention that unlike Reed, we do not decide at the beginning if at least one vertex is going to the separator. Instead, we consider this case for every representative only when we handle that representative (remember that we handle the representative in a serial way).

Here, we include the pseudocodes of our algorithms (4 and 5), but before that we introduce the global variables.

Global Variables

- G : the input graph
- $root$: a fixed vertex to start all the DFS visits
- W' : the list containing all the representatives (that do not go into the separator)
- $\mathcal{W}$: the list containing the weight of every representative
- $decision$: the string that dictates where we should place the representatives.
- S : the separator
- k : an upper bound on the capacity of the separator in the beginning.
- k' : an upper bound on the capacity of the separator at any given time.

The following section includes additional explanations on Algorithms 4 and 5 as well the proof of correctness.

2.4.3 The Correctness of Our Algorithm

We use a subroutine of Robrtson-Seymour's 4-approximation algorithm [14] to find a balanced separator of any given set W . We call this splitting "split by W ". If we continue just by splitting W , we end up having n^2 -dependence in the running time in worst case

Algorithm 4 Caption

```

procedure INIT_DFS( )
   $\mathcal{R\_tree\_info}$  = (list of vertices  $\mathcal{R\_vertices}$ , vertex  $rep\_connected$ )
   $s = \lfloor (1/4 - \epsilon/2) \frac{|V|}{k} \rfloor$   $\triangleright s$  : the size bound for splitting off an  $\mathcal{R}$ -tree.
   $decision = \text{" "}$   $\triangleright$  No representative should go into the separator
  while TRUE do
    Initialize  $W'$  and  $\mathcal{W}$  with empty lists  $\triangleright W'$  is the (ordered)
    list of the representatives and  $\mathcal{W}$  is the (ordered) list of their corresponding weights.
     $\mathcal{W} = (w[i]_{1 \leq i \leq |W'|})$ , where  $w[i]$  is the weight of the  $i$ -th representative.
     $k' = k$ 
    for all  $v \in V$  do
       $color[v] = \text{WHITE}$ 
    end for
     $vertex\_rep = \text{DFS\_VISIT}(\text{root})$ 
     $x.\mathcal{R\_vertices} = x.\mathcal{R\_vertices} \circ vertex\_rep.\mathcal{R\_vertices}$ , where  $x$  is the last
    representative added to  $W'$   $\triangleright$  If the topmost subtree has smaller size than  $s$ , we hook it
    onto the last representative  $x$ .
    if  $x.rep\_connected == nil$  then
       $x.rep\_connected = vertex\_rep.rep\_connected$ 
    end if
    if  $k' == 0$  then
      if connected components of  $G - S$  can form a balanced bipartition  $(L, R)$ 
    then return  $(L, R)$ 
    end if
    else
      try all partitions of  $W'$  into  $X$  and  $Y$  to see if there exists a balanced
      separator.
      if such a partition exists then return  $(L, R)$   $\triangleright L$  is  $X$  along with all the
      vertices represented by vertices of  $X$  (analogously  $R$  corresponds to  $Y$ ).
    end if
    end if  $\triangleright$  Now, we update the decision to look for the next possibility
     $decision = \text{lexicographically next string of at most the same length.}$   $\triangleright$  For
    instance, 0011001 follows 011000 and 101 follows 1001111
    if no such string exists then return "Treewidth is  $> k$ "
    end if
    Drop the "0"s to the right of the rightmost "1" in  $decision$ .  $\triangleright$  We always remove
    the rightmost "0"s and add them back only if necessary (in the DFS-Visit() procedure)
  end while
end procedure

```

Algorithm 5 Main recursive procedure of DFS-Trees

Ensure: $\mathcal{R}_{\text{tree_info}}$ about the current partial $\mathcal{R}_{\text{tree}}$

procedure DFS-VISIT(u)

$\text{this} = \text{new } \mathcal{R}_{\text{tree_info}}(\{u\}, \text{nil})$

$\text{color}[u] = \text{GRAY}$

while there exists v adjacent to u such that $\text{color}[v] == \text{WHITE}$ **do**

$\text{child} = \text{DFS_VISIT}(v)$

$\text{this}.\mathcal{R}_{\text{vertices}} = \text{this}.\mathcal{R}_{\text{vertices}} \circ \text{child}.\mathcal{R}_{\text{vertices}}$

$\text{color}[v] = \text{BLACK}$

if $\text{this.rep_connected} == \text{nil}$ **then**

$\text{this.rep_connected} = \text{child.rep_connected}$ $\triangleright$ Each vertex remembers one of the possible representatives to which one of its descendants in the current $\mathcal{R}_{\text{tree}}$ is connected to and passes it to its parent.

end if

end while

$\text{length} = |W'| + (k - k')$

if $|\text{this}.\mathcal{R}_{\text{vertices}}| \geq s$ **then**

if $\text{len}(\text{decision}) < \text{length}$ **then** $\triangleright 0$ means place it left/right. Also, we assume that the decision is padded with unlimited zeros to the right of decision that we do not show after the last 1, where 1 means place into the separator

$\text{decision} = \text{decision} \circ "0"$ $\triangleright$ where $\circ$ is string concatenation

end if

if $\text{decision}[\text{length}] == 0$ **then** $\triangleright "0"$ means place it left/right

$W' = W' \cup \{\text{this}\}$ **return** $(\emptyset, u)$

else $\triangleright$ i.e. $\text{decision}[\text{length}] == 1$ which means send this into the separator
 $k' = k' - 1$

if $\text{this.rep_connected} \neq \text{nil}$ **then** $\triangleright$ Handling the middle case of Figure 2.3
 $w[\text{this.rep_connected}] = w[\text{this.rep_connected}] + |\text{this}.\mathcal{R}_{\text{vertices}}|$ **re-**

turn this

else

for all $z \in \text{this}.\mathcal{R}_{\text{nodes}}$ **do**

$\text{color}[z] = \text{WHITE}$ $\triangleright$ Either the left or the right case of Figure 2.3 has happened. In either case we do not need to worry.

end for **return** $(\emptyset, \text{nil})$

end if

end if

end if

end procedure

scenario. In order to avoid this, after every $\log_{\frac{3}{2}} k$ such splittings, we split based on V (the reasoning can be found in Section 2.3.4). Splitting by V is handled by finding the representatives and then splitting the representatives in a balanced way. As we mentioned earlier, at most $\frac{n}{C_\epsilon}$ vertices would be on the opposite side of their representatives. Hence, this way of splitting splits the entire graph in a balanced way without actually deciding on every vertex (if there exists a balanced separator at all). Below we explain the latter part in more detail.

We do a DFS and the first time that we post-visit a vertex with at least $\frac{n}{C_\epsilon k}$ vertices in its subtree, we call that vertex a representative which represents the vertices of its subtree. Then, we decide what should we do with that representative.

Algorithm 4 initiates the process by calling the DFS on G starting from a fixed vertex $root$. The reason that we fix the starting point is that we want to have a deterministic DFS search so that when we change our “*decision*”, the DFS follows the same order.

Decision is a string $\in \{0, 1\}^*$ that tells us what to do after we find a representative. If one *decision* was not good (i.e. did not give us a balanced separator), we check the next *decision*.

We start with the decision that no representative should go into the separator (*decision* = “”). Notice that *decision* is a string. The reason is that we want to keep the leading 0’s. Decision either is empty or ends with a 1 (we stop at the right-most 1 and assume the string is padded with the unlimited³ number of 0s to the right). If necessary, we keep adding those 0’s to the right.

If $decision[i] = 0$, we send the i -th representative left/right. If it was 1, we send that vertex into the separator and do not consider it a representative anymore. If $|decision| < s$, this means it was supposed to be 0 (because the string can be padded with unlimited 0s). Then, we add 0 to the R.H.S. of the *decision* and send this representative left/right. Sending left/right means just place it in W' , for now. At the end of this iteration, check all the possible ways of splitting W' into X and Y to find a balanced separator. If such a split exist, return (L, R) , where $L = X \cup \{\text{any vertex represented by a representative in } X\}$ (analogously R corresponds to Y). If none exist, check the next *decision* and start over.

Then, the while loop on line 6 checks the entire search space. For any fixed *decision*, we deterministically know what should happen to the representatives that we find. Each time we have a new decision, we set the variables back to their original values and restart searching for a balanced separator.

In line 12, we call the DFS_Visit on $root$. DFS_Visit returns a set of vertices (either

³In fact, $C_\epsilon k$ is sufficient

the subtree that is small to form a group represented by a representative or $\emptyset$ if they form a group) and the information whether any vertex in the set returned in the first argument is connected to another representative down below (we will see why we need this, later). We refer to this pair of information as $\mathcal{R_tree_info}$ (check line 3 of Algorithm 4). There is a chance that the last vertex post-visited (which is *root*) is not a representative, meaning that the size of its subtree is smaller than the threshold s . In that case, we hook the subtree rooted at *root* to the last representative that we have found (similar to part b of Figure 2.3). This happens in line 13.

The main process happens in DFS_VISIT. Whenever it post-visits a vertex, it checks whether its subtree is big enough. If the size of the current subtree hits the threshold, it forms a group and decides whether the current representative should go left/right or into the separator based on *decision* (0 means send it into the separator and 1 means send it left/right). Otherwise, it passes the set of nodes to the parent along with a possible representative connected to its subtree down below.

In line 28 of Algorithm 4, we update the decision if the current decision has failed.

The next *decision* is the next string in lexicographical order with at most the same length. For example

$$\begin{cases} 1 & \text{follows } 0 \\ 100111 & \text{follows } 100110111 \\ 001011 & \text{follows } 0010101111111 \end{cases}$$

The reason that we do not check all the decision space is that if *decision* leads us to a failure, there is no reason to check a string with *decision* as one of its prefixes. Notice that every time we run out of the characters on the R.H.S. on *decision*, we just concatenate a ‘0’ to the R.H.S. of it.

If $tw(G) \leq k$, then there exists a tree decomposition of G namely $\mathcal{T}$ with width (at most) k . Then based on Lemma 3.4.1, there exists a strong centroid x in $\mathcal{T}$. Both $\mathcal{T}$ and x are unknown, but they do exist (if we knew, we already had a tree decomposition of width k and a balanced separator). The connected components of $G \setminus B_x$ can form a bipartition (L, R) such that no part has more than $\frac{1}{2}n$ vertices. For the sake of argument, fix $\mathcal{T}$, x , L , and R . Every $W' \subseteq V$ is partitioned into parts in L (name it X), parts in separator $S = B_x$, and parts in R (name it Y). The weight of each part is at most $(\frac{1}{2} + \frac{1}{C_\epsilon})n$. One of the branches picks W' . On the branch that picks W' , the separator B_x lies between X and Y . There is no guarantee that we find this separator, but the algorithm finds at least one separator of size $\leq k$ separating X from Y in a balanced

way. Once more, up to $\frac{1}{C_\epsilon}n$ vertices are on the opposite side. Now, the larger side has at most $\frac{1}{2}n + \frac{2}{C_\epsilon}n$ vertices and the smaller side has at least $\frac{1}{2}n - \frac{2}{C_\epsilon}n$ vertices. As we argued, we set $C_\epsilon \leftarrow \frac{4}{1-2\epsilon}$. This means in worst case scenario, our algorithm finds a $(\epsilon, 1 - \epsilon)$ -separation. Each part has a constant fraction of the vertices. Hence, we have a somewhat balanced separator.

The approximation ratio analysis is similar to what we described in Section 2.3.4. We start with $|W| \leq 4k$. It can be written as $3k + k$. Let's call the second term, k_{excess} . Initially, $k_{excess} = k$. Each time we split W , the separator separates W into two parts with the largest part having at $2/3|W|$. We include the separator itself in both sides before recursing. This means

$$|W| \leq \frac{2}{3}(3k + k_{excess}) + \underbrace{k}_{\text{upper bound on the size of the separator}} = 3k + \overbrace{\frac{2}{3}k_{excess}}^{\text{new } k_{excess}}.$$

Each time, k_{excess} might drop by a factor of $2/3$. After $\log_{\frac{3}{2}} k$ iterations, k_{excess} might drop to zero. At this point, we do one split by V . We have to add the separator to both parts and this means $|W|$, which was upper bounded by $3k$, might go up to $4k$ once more. Now, we can do $\log_{\frac{3}{2}} k$ splits by W again and continue as described.

Notice that throughout the entire process $|W| \leq 4k$. Ultimately, in order to merge the two tree decompositions that we found for two subproblems, we add the separator to both sides. This means the largest bag can have up to $5k$ vertices and hence, the approximation ratio is 5.

2.5 Running Time of Our Algorithm

Let G , k , and t be the initial graph, the bound on the size of the separator, and the number of representatives, respectively. t is at most $C_\epsilon k$ since the cut-off threshold for the volume of the subtrees was $\frac{n}{C_\epsilon k}$. While proceeding with the algorithm at each step, let G' , k' , and t' be the current graph, the current bound on the size (capacity) of the separator, and a bound on the number of representatives still to find, respectively. Each time we send some vertex to the separator, we decrease the capacity by one. We assume a worst case, where the number of $\mathcal{R}$ -trees is t , and the separator found has size k (other cases only speed up the running time). Then, the recurrence for the running time to find

a separator of size at most k in G is:

$$T(t, k) \leq T\left(t \frac{k-1}{k}, k-1\right) + 2T(t-1, k) + Qkn + \mathcal{O}(1), \text{ for } t, k > 0, \quad (2.4)$$

where Q is the constant factor of the DFS. On the R.H.S., the first term handles the case where the representative goes to the separator. Therefore, k decreases by 1, and the number of subtrees becomes at most $t \frac{k-1}{k}$ (delete that vertex and continue the DFS). The second term handles the case that the current representative does not go into the separator but left or right. Here, the capacity of the separator is unchanged but the number of subtrees decreases by 1. The third term is the upper bound of the exact running time of the DFS. And the last term $\mathcal{O}(1)$ is the overhead to make the calls. The base cases of the recurrence:

$$\begin{aligned} T(0, k) &\leq Q(k+1)kn && \text{to do } k+1 \text{ DFSs. We find } k \text{ paths from } X \text{ to } Y, \\ & && \text{and search for augmenting paths.} \\ T(t, 0) &\leq Qkn && \text{We have to test whether } S \text{ is indeed a separator.} \end{aligned}$$

The recurrence in 2.4 might seem hopeless, so we simplify it (generous bound).

$$T(t, k) \leq T(t, k-1) + 2T(t-1, k) + Qkn + \mathcal{O}(1), \quad t, k > 0$$

(2.5)

Now, we have to solve this recurrence. Our recursion tree starts from the root $T(t, k)$ and has two children $T(t, k-1)$ and $T(t-1, k)$, left and right, respectively. This is an unbalanced binary tree. Each strand terminates when one of the arguments of $T(\cdot, \cdot)$ becomes zero. Each time we choose the left branch (putting one representative into the separator), we decrease k by 1. Otherwise (putting the representative and its subtree to the right or left set of the separator), we decrease t by 1 and multiply the value by 2. Let $\#(t-i, 0)$ be the (worst case) number of leaves with the first argument $t-i$ and the second argument $k=0$, for $0 \leq i < t$ (analogous notation for $\#(0, k-j)$ for $0 \leq j < k$). Observe that $\#(t-i, 0) = \binom{k+i}{i}$, and $\#(0, k-j) = \binom{t+j}{j}$. The first two terms of Equation 2.5 can be computed at the leaves and the other two terms are spent in every vertex of the recursion tree. The reason is that for the first two terms, we need the results of the children and recursively everything relies on the results of the leaves. Each time we decide to consider a subproblem (any internal node corresponds to a subproblem), we have to find the actual separator. This step takes Qkn time. Also, in order to make the recursive calls, we spend $\mathcal{O}(1)$ time.

Now, we compute the first part (the first two terms).

$$\begin{aligned}
& \sum_{i=0}^{t-1} \left(\#(t-i, 0) \underbrace{2^i}_{i \text{ right branches}} T(t-i, 0) \right) + \sum_{j=0}^{k-1} \left(\#(0, k-j) \underbrace{2^t}_{t \text{ right branches}} T(0, k-j) \right) \\
& \leq \sum_{i=0}^t \left(\binom{k+i}{i} 2^i \underbrace{Qk^2n}_{\text{Eq. 2.5}} \right) + \sum_{j=0}^{k-1} \left(\binom{t+j}{j} 2^t \underbrace{Qkn}_{\text{Eq. 2.5}} \right) \\
& \leq Qk^2n \sum_{i=0}^t \binom{k+i}{k} 2^i + Qkn2^t \sum_{j=0}^{k-1} \binom{t+j}{j} \\
& < Qk^2n \sum_{i=0}^t \binom{k+t}{k} 2^i + Qkn2^t \binom{k+t}{k-1} \\
& \leq Qk^2n \binom{k+t}{k} (2^{t+1} - 1) + Qk^2n2^t \binom{k+t}{k}
\end{aligned}$$

Now, we have to compute the second part of Equation 2.5 where we should look at every internal vertex of the tree. We have $\binom{t+k}{k}$ internal vertices and in each vertex with value k' , we spend at most $Qk'kn + \mathcal{O}(1) \leq Qk^2n(1 + \mathcal{O}(\frac{1}{n}))$. Hence,

$$\begin{aligned}
T(t, k) & \leq (2^{t+1} - 1) Qk^2n \binom{k+t}{k} + Qk^2n(1 + \mathcal{O}(\frac{1}{n})) \binom{t+k}{k} \\
& = Qk^2n \left(2^{t+1} + \mathcal{O}\left(\frac{1}{n}\right) \right) \binom{k+t}{k}
\end{aligned} \tag{2.6}$$

Notice that $T(\cdot, k)$ is monotonic (due to the definition of T) and use the fact that $t \leq C_\epsilon k$. Now, we simplify Equation 2.6 by bounding $T(t, k)$ with $T(C_\epsilon k, k)$ and using Striling's approximation.

$$\begin{aligned}
T(t, k) & \leq T(C_\epsilon k, k) \leq Qk^2n \left(2^{C_\epsilon k+1} + \mathcal{O}\left(\frac{1}{n}\right) \right) \binom{(C_\epsilon + 1)k}{k} \\
& \leq Qk^{\frac{3}{2}}n \left(2^{C_\epsilon k+1} + \mathcal{O}\left(\frac{1}{n}\right) \right) \sqrt{\frac{C_\epsilon + 1}{2\pi C_\epsilon}} \left(1 + \frac{1}{C_\epsilon} \right)^{C_\epsilon k} (C_\epsilon + 1)^k
\end{aligned}$$

Now, we compute the running time (T_V), when we split based on V . We assume a worst case split of ϵ to $1 - \epsilon$ and worst case separator of size k .

$$\begin{aligned}
T_V(n, k) & = T_V(\epsilon n + k, k) + T_V((1 - \epsilon)n + k, k) + T(t, k) \\
& \leq T_V(\epsilon n + k, k) + T_V((1 - \epsilon)n + k, k)
\end{aligned}$$

$$\begin{aligned}
& + Qk^{\frac{3}{2}}n \left(2^{C_\epsilon k+1} + \mathcal{O}\left(\frac{1}{n}\right) \right) \sqrt{\frac{C_\epsilon+1}{2\pi C_\epsilon}} \left(1 + \frac{1}{C_\epsilon} \right)^{C_\epsilon k} (C_\epsilon + 1)^k \\
& \leq Qk^{\frac{3}{2}} \left(2^{C_\epsilon k+1} + \mathcal{O}\left(\frac{1}{n}\right) \right) \sqrt{\frac{C_\epsilon+1}{2\pi C_\epsilon}} \left(1 + \frac{1}{C_\epsilon} \right)^{C_\epsilon k} (C_\epsilon + 1)^k \frac{1}{\epsilon} n \ln n
\end{aligned}$$

In the above equation, we use the corollary to the following lemma (Lemma 2.5.1). Note that the reason we add k to both recursive calls is that we add the separator to both subproblems.

Lemma 2.5.1. Assume $0 < \epsilon \leq \frac{1}{2}$, $0 < c' \leq c$, $2 \leq k$, and $n_1 + n_2 = n$. Then the recurrence

$$f(n+k) \leq \begin{cases} c'(n+k) & \text{if } n-k \leq 4k \\ f(n_1+k) + f(n_2+k) + c(n+k) & \text{otherwise,} \end{cases}$$

where $\frac{1}{2}n \leq n_1 \leq (1-\epsilon)n$ has a solution with $f(n+k) \leq \frac{c}{\epsilon}n \ln n - ck$, for $n \geq 2k$.

Proof. Case 1: $2k \leq n \leq 4k$. Then $n \geq 4$ implying $\ln n > 1$ and

$$f(n+k) \leq c(n+k) < 2cn - ck < \frac{c}{\epsilon}n \ln n - ck.$$

Case 2: $n \geq 4k$ and $n_2 \geq 2k$.

$$\begin{aligned}
f(n+k) & \leq f(n_1+k) + f(n_2+k) + c(n+k) \\
& \leq \frac{c}{\epsilon} (n_1 \ln n_1 + (n-n_1) \ln(n-n_1)) - 2ck + c(n+k) \\
& < \frac{c}{\epsilon} \left((1-\epsilon)n \underbrace{(\ln(1-\epsilon) + \ln n)}_{< -\epsilon} + \epsilon n (\ln \epsilon + \ln n) \right) + c(n-k) \\
& < \frac{c}{\epsilon} n \ln n - c(1-\epsilon)n + cn \ln \epsilon + c(n-k) \\
& \leq \frac{c}{\epsilon} n \ln n + \underbrace{(\epsilon + \ln \epsilon)}_{< 0 \text{ for } \epsilon \leq \frac{1}{2}} cn - ck \\
& \leq \frac{c}{\epsilon} n \ln n - ck
\end{aligned}$$

Case 3: $n \geq 4k$ and $n_2 = n - n_1 < 2k$.

$$\begin{aligned}
f(n+k) & \leq f(n_1+k) + f(n_2+k) + c(n+k) \\
& \leq \frac{c}{\epsilon} n_1 \ln n_1 - ck + c'(n_2+k) + c(n+k)
\end{aligned}$$

$$\begin{aligned}
&\leq \frac{c}{\epsilon}(1-\epsilon)n(\ln(1-\epsilon) + \ln n) - ck + c'(n_2 + k) + c(n+k) \\
&< \frac{c}{\epsilon}(1-\epsilon)n(-\epsilon + \ln n) - ck + c'(n_2 + k) + c(n+k) \\
&\leq \frac{c}{\epsilon}n \ln n - ck - cn \ln n - c(1-\epsilon)n + c'(n_2 + k) + c(n+k) \\
&< \frac{c}{\epsilon}n \ln n - ck
\end{aligned}$$

The last inequality is true, because $n \geq 4k \geq 8$ implying $\ln n > 2$, and $n+k < 3k \leq \frac{3}{4}n$. $\square$

Corollary. Under the conditions of the Lemma 2.5.1,

$$f(n) \leq \frac{c}{\epsilon}n \ln n.$$

Now, the total running time of the algorithm (T_t) is:

$$T_t(n, k) \leq Qk^{\frac{3}{2}} \left(2^{C_\epsilon k+1} + \mathcal{O}\left(\frac{1}{n}\right) \right) \sqrt{\frac{C_\epsilon + 1}{2\pi C_\epsilon}} \left(1 + \frac{1}{C_\epsilon} \right)^{C_\epsilon k} (C_\epsilon + 1)^k \frac{(1 + \log_{\frac{3}{2}} k)}{\epsilon} n \ln n, \quad (2.7)$$

where it takes $\log_{\frac{3}{2}} k$ steps so that the k_{excess} drops to zero (that is when we need to split by V once more).

As we mentioned in Section 2.4.1, $C_\epsilon = \frac{4}{1-2\epsilon}$. We plug that into Equation 2.7.

$$T_t(n, k) \leq Qk^{\frac{3}{2}} \left(2^{\frac{4k}{1-2\epsilon}+1} + \mathcal{O}\left(\frac{1}{n}\right) \right) \sqrt{\frac{5-2\epsilon}{8\pi}} \left(1.25 - \frac{\epsilon}{2} \right)^{\frac{4k}{1-2\epsilon}} \left(\frac{4}{1-2\epsilon} + 1 \right)^k \frac{(1 + \log_{\frac{3}{2}} k)}{\epsilon} n \ln n, \quad (2.8)$$

Theorem 2.5.1. Let $C_0 = 4 + 4 \log 1.25 + \log 5 < 7.60965$. For any $C > C_0$, there exist a 5-approximation algorithm that solves treewidth in time $\mathcal{O}(2^{Ck} n \log n)$, where k is treewidth+1.

Proof. In Equation 2.8, let $\epsilon \rightarrow 0$.

$$\lim_{\epsilon \rightarrow 0} T_t(n, k) \leq \frac{1}{\epsilon} \sqrt{\frac{5}{2\pi}} Qk^{\frac{3}{2}} (\log_{\frac{3}{2}} k) 2^{(4+4 \log 1.25 + \log 5)k} n \ln n \quad (2.9)$$

$\square$

For instance, by setting $\epsilon = 10^{-2}$, we have

$$T_t(n, k) \leq 90Qk^{1.5} \log_{3/2} k 2^{7.718k} n \ln n,$$

and by setting $\epsilon = 10^{-3}$, we get

$$T_t(n, k) \leq 893Qk^{1.5} \log_{3/2} k 2^{7.621k} n \ln n$$

Corollary 2.5.1. A 5-approximation of treewidth can be solved in time $\mathcal{O}(2^{7.61k} n \log n)$.

2.6 Conclusion

In this paper, we have given a detailed analysis of Reed's treewidth approximation algorithm [13]. We have shown that it runs in time $\mathcal{O}(2^{24k} k! n \log n)$, where k is treewidth+1. Furthermore, we have shown that it is a 5 or 7 approximation, depending on how frequently we split based on V .

Then, we have given our improved algorithm which runs in time $\mathcal{O}(2^{7.61k} n \log n)$, with the same approximation ratio.

Our main goal was to further improving the coefficient of k in the exponent to make it more applicable. We think it is still possible to further improve it. Also, trying to come up with a better approximation ratio is worthwhile while maintaining the same dependence on k (a 2-approximation [1] is known but the coefficient of k in the exponent is larger).

Finally, another direction for future work could be giving hardness results on the approximation ratio.

Chapter 3 |

Finding All Leftmost Separators of Size $\leq k$

This chapter is based on a joint work with Martin Fürer [22].

3.1 Introduction

Finding vertex separators that partition a graph in a “balanced” way is a crucial problem in computer science, both in theory and applications. For instance, in a divide and conquer algorithm, most of the time it is vital to have balanced subproblems. If we want to separate two subsets of vertices in a graph, we prefer the separator to be closer to the bigger side. In this work, we place the bigger set on the left side and the smaller one on the right. Before going into depth, we review and introduce some notations.

3.1.1 Notation

W.l.o.g., assume that G is a connected graph. $S \subseteq V$ is a separator that separates two subsets of vertices $X, Y \subseteq V$ in G , if there is no $X - Y$ path in $G[V \setminus S]$, where $G[V \setminus S]$ is the induced graph on $V \setminus S$. In the following, we use $G - S$ instead of $G[V \setminus S]$ for the sake of simplicity. We call S an $(X, Y)^G$ -separator. Later on, we drop the superscripts if it is obvious from the context.

Definition 3.1.1. $S_{X,Y}^G$ is the set of all $(X, Y)^G$ -separators.

Definition 3.1.2. The separator $S \in S_{X,Y}^G$ partitions $G - S$ into three sets $V_{X,S}$, $V_{S,Y}$, and V_Z , where $V_{X,S}$ is the set of vertices with a path from $X \setminus S$, $V_{S,Y}$ is the set of vertices with a path from $Y \setminus S$, and V_Z is the set of all vertices reachable from neither $X \setminus S$ nor $Y \setminus S$ in $G - S$.

Having a non-empty V_Z set is only to our advantage. We think of X being on the left side and Y on the right side of S . Any of the three sets (X , Y , and S) might intersect.

Definition 3.1.3. Partial Ordering. We say separator $S \in S_{X,Y}^G$ is at least as much to the left as separator $S' \in S_{X,Y}^G$ if $V_{X,S} \subseteq V_{X,S'}$. In this case, we use the notation $S \preceq S'$.

Definition 3.1.4. Separator $S \in S_{X,Y}^G$ is called an $(X, Y, \leq k)^G$ -separator if $|S| \leq k$.

Definition 3.1.5. Separator $S \in S_{X,Y}^G$ is called a *leftmost* $(X, Y, \leq k)^G$ -separator if it is minimal and there exists no other minimal $(X, Y, \leq k)^G$ -separator S' such that $S' \preceq S$.

Notice that the minimality is important here, otherwise according to the partial ordering definition, one can keep adding extra vertices to the left of S' (towards X) and artificially make it more to the left. In order to avoid this, we require all separators we work with to be minimal unless specified otherwise.

The notion of leftmost separator is closely related to the notion of *important separator*. Important separator has been defined in [24], and then used in [25] and [26].

The difference between a leftmost separator and an important separator comes from their corresponding partial orders. The partial order defined for important separators is as follows:

Definition 3.1.6. Partial Ordering used for Important Separators. Separator $S \in S_{X,Y}^G$ dominates (or “is more important than”) separator $S' \in S_{X,Y}^G$ if $|S| \leq |S'|$ and $V_{S,Y} \subset V_{S',Y}$.

Definition 3.1.7. Separator $S \in S_{X,Y}^G$ is an *important* $(X, Y, \leq k)^G$ -separator if there exists no other minimal $(X, Y, \leq k)^G$ -separator S' dominating S .

As you see, when ordering important separators we also look at the relation between the sizes but in a leftmost separator, its size just has to be $\leq k$.

Lemma 3.1.1. Every leftmost $(X, Y, \leq k)^G$ -separator is an important $(X, Y, \leq k)^G$ -separator, but the converse does not hold.

Proof. Let S be a leftmost $(X, Y, \leq k)^G$ -separator. Assume for the sake of contradiction that S is not important. Hence, there exists important $(X, Y, \leq k)^G$ -separator S' that dominates S . Therefore $V_{S',Y} \supset V_{S,Y}$. Notice that $S \setminus S'$ is nonempty. Let $v \in S \setminus S'$. Then, $v \in V_{S',Y}$. Therefore, $V_{X,S'} \subset V_{X,S}$. This implies that $S' \preceq S$, which is a contradiction.

To show that the converse of the proposition does not hold, consider the following counter-example. □

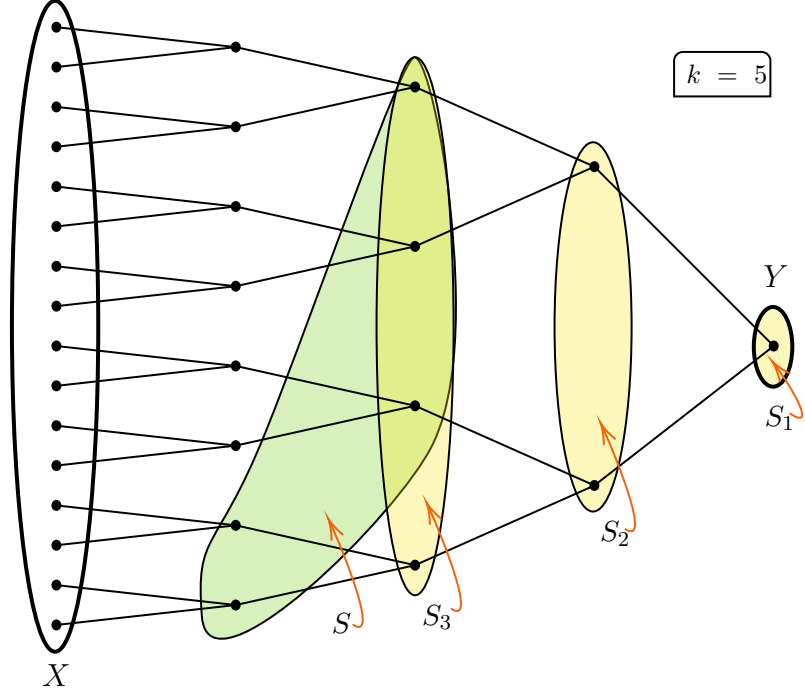


Figure 3.1. Here $k = 5$. Separators S_1 , S_2 , S_3 , and S are all important but only S is leftmost

As shown in Figure 3.1, not all the important separators are leftmost. Our purpose is to find a separator more towards the bigger side in order to have more balanced separators. For that reason, not all the important separators are good. For instance we do not need to consider S_1 because that is the most unbalanced separator one can find. This is the main reason that we defined the new notion of leftmost separators. The reason that S_1 in Figure 3.1 is important is that $|S_1| \leq |S_2|$ (even though $V_{S_2, Y}$ contains $V_{S_1, Y}$). As argued, leftmost separators are better candidates for our application. However, as the reader can see, there is a strong similarity between these two notions. We give tight upper bounds for the number of leftmost separators and a tight upper bound for the number of important separators.

Lemma 3.1.2. Let $\mathcal{A}_{X, Y, \leq k}^G$ and $\mathcal{B}_{X, Y, \leq k}^G$ be the set of all leftmost $(X, Y, \leq k)^G$ -separators and the set of all important $(X, Y, \leq k)^G$ -separators, respectively. Then,

$$\mathcal{B}_{X, Y, \leq k}^G = \bigcup_{i=1}^k \mathcal{A}_{X, Y, \leq i}^G$$

Proof. Based on Lemma 3.1.1, for every $1 \leq i \leq k$, every leftmost $(X, Y, \leq i)^G$ -separator is an important $(X, Y, \leq i)^G$ -separator. Furthermore, every important $(X, Y, \leq i)^G$ -

separator is an important $(X, Y, \leq k)^G$ -separator. Hence,

$$\mathcal{B}_{X,Y,\leq k}^G \supseteq \bigcup_{i=1}^k \mathcal{A}_{X,Y,\leq i}^G$$

For the other side, we show that every important $(X, Y, \leq k)^G$ -separator S is a leftmost $(X, Y, \leq |S|)^G$ -separator. For the sake of contradiction, assume that there exist important $(X, Y, \leq k)^G$ -separator S that is not a leftmost $(X, Y, \leq |S|)^G$ -separator (Notice that $\mathcal{A}_{X,Y,\leq |S|}^G$ is nonempty since there is at least one separator of $\leq |S|$ between X and Y). This means there exist leftmost $(X, Y, \leq |S|)^G$ -separator S' that $S' \preceq S$. Notice that $|S'| \leq |S|$ and also $V_{S',Y} \supset V_{S,Y}$. This means S' dominates S and hence S is not important, which is a contradiction. $\square$

In this paper, we show that the number of leftmost $(X, Y, \leq k)$ -separators is $\leq C_{k-1}$, where C_n is the n -th Catalan number. Furthermore, we close the gap and show that this upper bound is tight. Then, we give an $\mathcal{O}(4^k kn)$ -time algorithm finding all minimal leftmost $(X, Y, \leq k)$ -separators. Notice that $C_{k-1} \sim \frac{4^{k-1}}{\sqrt{\pi(k-1)}^{\frac{3}{2}}}$.

Based on Lemma 3.1.2, this implies that the number of important $(X, Y, \leq k)$ -separators is $\leq \sum_{i=1}^{k-1} C_i$ and the bound is tight.

One of the important applications of the algorithm that finds all the leftmost separators is treewidth approximation. Treewidth approximation is a crucial problem in computer science. Courcelle's methatheorem [4] states that every problem which can be described in monadic second order logic has an FPT algorithm with the treewidth k as its parameter. An FPT algorithm is an algorithm that runs in time $\mathcal{O}(f(k) \text{poly}(n))$, where n is the input size, k is the parameter (here, treewidth), and $f(\cdot)$ is a computable function.

So, based on Courcelle's methatheorem, many NP-complete graph problems obtain polynomial algorithms (in terms of the input size), and hence they can be solved fast if the treewidth is small. These algorithms require access to tree decompositions of small width. However, finding the exact treewidth itself is another NP-complete problem [6]. Here, we look for an approximation algorithm to solve the treewidth problem.

Problem 0. Given a graph $G = (V, E)$, and an integer $k \in \mathbb{N}$, is the treewidth of G at most k ? If yes, output a tree decomposition with width $\leq \alpha k$, where $\alpha \geq 1$ is a constant. Otherwise, output a subgraph which is the bottleneck.

There are various algorithms solving this problem for different α 's (the approximation

Reference	Approximation Ratio α	Dependence on k	Dependence on n	Comments
Robertson & Seymour [14]	4	3^{3k}	n^2	
Lagergren [15]	8	$2^{\mathcal{O}(k \log k)}$	$n \log^2 n$	
Reed [13]	7 or 5	$2^{24k} k!$	$n \log n$	Large coefficient of k in the exponent
Amir [17]	4.5	2^{3k}	n^2	
Amir [17]	$\frac{11}{3}$	$2^{3.6982k}$	n^2	
Bodlaender et al. [19]	3	$2^{\mathcal{O}(k)}$	$n \log n$	The coefficient of k is not stated
Bodlaender et al. [19]	5	$2^{\mathcal{O}(k)}$	n	The coefficient of k is not stated
Belbasi & Fürer [8]	5	$2^{8.766k}$	$n \log n$	Small coefficient of k
Korhonen [1]	2	$2^{\mathcal{O}(k)}$	n	better coefficient of k compared to [19] but still not very applicable.
This chapter	5	$2^{6.755k}$	$n \log n$	Practical for small k

ratio). As mentioned above, we are interested in constant-factor approximation FPT algorithms. Table 4.1 shows a brief history of the previous work.

Algorithms [19] and [1] both run in $2^{\mathcal{O}(k)}n$ time, which is linear in n . However, the coefficients of k in the exponent are large. The former one does not mention the exact coefficient and seems to have a very large coefficient. The latter one, which is a very recent paper, mentions that the coefficient of k in the exponent is some number between 10 and 11. Our goal is to make treewidth approximation more applicable by decreasing the coefficient of k in the exponent. We can afford an extra $\log n$ factor in the running time in order to reduce the huge dependence on k . We sacrifice the linear dependence on n , and give an algorithm which runs way faster in various cases. So, let us look at $n \log n$ -time algorithms. Reed [13] gave the first $n \log n$ -time algorithm. He did not mention the dependence on k precisely but a detailed analysis in [8] shows that it is $\mathcal{O}(2^{24k} k!)$. Here, even though $2^{24k} = o(k!)$, actually $k!$ is reasonable for small k 's while 2^{24k} is not. Later on, the authors of this paper introduced an $\mathcal{O}(2^{8.766k} n \log n)$ -algorithm [8]. The algorithm presented in this paper is based on [13] and [8]. In these papers, when it is known that a good separator S exists between X and Y , an efficient algorithm finds an arbitrary separator between X and Y . The ability to find leftmost separators allows for

an improvement. If S is a good separator between X and Y , and $V_{X,S}$ is estimated to be at least as big as $V_{S,Y}$, then the best leftmost separator between X and Y has a definite advantage.

Instead of a balanced separator with minimum size, we consider all leftmost separators (closest possible to the bigger side). This helps us to obtain an $\mathcal{O}(2^{6.755k} n \log n)$ -time algorithm with the same approximation ratio of 5 as in [13] and [8].

Before moving onto the next section, we have to mention that the algorithm to find all leftmost separators works for both directed and undirected graphs.

Below, we summarize our contributions.

3.1.2 Our Contributions

First, we give a tight upper bound on the number of the leftmost separators.

Theorem 3.1.1. Let $G = (V, E)$ be a graph, $X, Y \subseteq V$, and $k \in \mathbb{N}$. The number of leftmost $(X, Y, \leq k)^G$ -separators¹ is at most C_{k-1} ² $= \frac{1}{k} \binom{2(k-1)}{k-1} \sim \frac{4^{k-1}}{\sqrt{\pi(k-1)^{3/2}}}$. Furthermore, the number of important $(X, Y, \leq k)^G$ -separators is at most $\sum_{i=0}^{k-1} C_i$. Both bounds are tight.

Then, we give an algorithm finding all leftmost separators.

Theorem 3.1.2. Let $G = (V, E)$ be a graph, $X, Y \subseteq V$, and $k \in \mathbb{N}$. There is an $\mathcal{O}(2^{2k} \sqrt{kn})$ -time algorithm which outputs all the leftmost $(X, Y, \leq k)^G$ -separators.

Now, we use the algorithm finding all the leftmost separators to solve treewidth approximation much faster.

Theorem 3.1.3. Let $G = (V, E)$ be a graph, and $k \in \mathbb{N}$. There is an algorithm that either outputs a tree decomposition of G with width $\leq 5(k-1)$, or determines that $tw(G) > k-1$ in time $\mathcal{O}(2^{6.755k} n \log n)$.

3.2 Finding the Leftmost Minimum Size $(X, Y, \leq k)^G$ -Separator

In this section, we review an algorithm for the following problem which has been fully described in [23] in details. It is based on [27], [14], and [13].

¹Notice that all leftmost separators are minimal per definition.

² C_n is the n th Catalan number

Problem 1. Given a graph $G = (V, E)$, sets $X, Y \subseteq V$, and $k \in \mathbb{N}$, is there an $(X, Y, \leq k)$ -separator?

Lemma 3.2.1. (Lemma 11.20 of [23]) There is an algorithm which solves Problem 1 in time $\mathcal{O}(k|V|)$.

The proof has been given in [23]. Here, we just briefly mention the crux of the idea and apply a tiny modification. Later on, we write the pseudocode of Algorithm 6 since we use it in Algorithm 3.3. The main idea is based on the following theorem of Menger [27].

Theorem 3.2.1. (A version of Menger's theorem [27]) Let $G = (V, E)$ be a graph, and $X, Y \subseteq V$. Then, the size of the minimum $(X, Y)^G$ -separator is equal to the maximal number of disjoint paths from vertices of X to vertices of Y .

So, the problem reduces to finding the number of disjoint paths from X to Y , and this itself reduces to a network flow problem (with the capacities on the vertices and not the edges).

Definition 3.2.1. Let $\mathcal{P}$ be a family of disjoint paths from a set $X \subseteq V$ to a set $Y \subseteq V$ in a graph $G = (V, E)$. We call Q a $\mathcal{P}$ -augmenting walk if $Q = q_1 \cdots q_s$ such that $q_i \in V$ for all i with $1 \leq i \leq s$, and $\{q_i, q_{i+1}\} \in E$ for all i with $1 \leq i \leq s - 1$, and also

1. No edge shows up twice on Q ,
2. If Q intersects $P = p_1 \cdots p_l \in \mathcal{P}$ at $q_i = p_j$, then $i \neq s$ and $q_{i+1} = p_{j-1}$ (i.e., if Q intersects a path $P \in \mathcal{P}$, they share at least one edge and that edge appears in opposite directions on P and Q).

Claim 3.2.1. (Claim 1 of Lemma 11.20 in [23]) Let $\mathcal{P}$ be a family of pairwise disjoint paths from X to Y and let Q be a $\mathcal{P}$ -augmenting walk from X to Y . Then, there exists a family of pairwise disjoint paths from X to Y of size $|\mathcal{P}| + 1$.

The idea is to think of paths in $\mathcal{P}$ and also Q as sending a unit flow from X to Y ³ (the flows in opposite directions cancel each other when Q intersects with a path in $\mathcal{P}$). This gives a new family of pairwise disjoint paths with one more disjoint path. Then, we keep trying to find another $\mathcal{P}$ -augmenting walk and update $\mathcal{P}$ until it is impossible to proceed (no $\mathcal{P}$ -augmenting walk is found). Check Figure 3.2; initially, $\mathcal{P} = \{P_1, P_2, P_3, P_4\}$. We find an $\mathcal{P}$ -augmenting path Q . After sending a unit flow through all the paths, we get a new set of paths $\mathcal{P}' = \{P'_0, P'_1, P'_2, P'_3, P'_4\}$ of size one more.

³Again, notice that the capacities are on the vertices and not the edges.

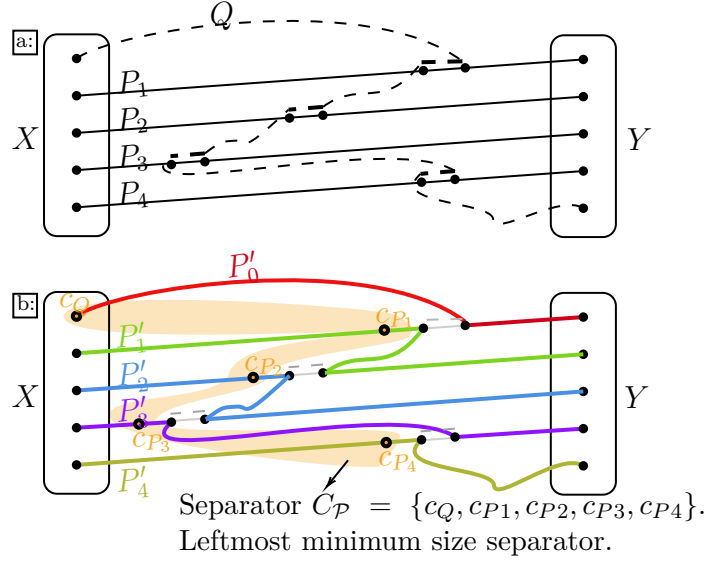


Figure 3.2. Augmenting Paths

Now that we have the maximum size family of pairwise disjoint paths from X to Y (namely $\mathcal{P}$), we show how to find the leftmost minimum sized $(X, Y, \leq k)^G$ -separator⁴

Let $R(\mathcal{P})$ ⁵ be the set of all vertices $v \in V$ such that there is a $\mathcal{P}$ -augmenting path from X to v . For every path $P \in \mathcal{P}$, let c_P be the first vertex on P not contained in $R(\mathcal{P})$. Also, let $C(\mathcal{P}) = \bigcup_{P \in \mathcal{P}} \{c_P\}$.

Note that if $c_P \notin X$, then c_P lies on an augmenting path that continues to the predecessor of c_P on P .

Claim 3.2.2. (Claim 2 of Lemma 11.20 in [23]) $C(\mathcal{P})$ separates X from Y .

So, $C(\mathcal{P})$ is an $(X, Y, \leq k)$ -separator and also based on Menger's result, this is a separator of the minimum size. In fact, $C(\mathcal{P})$ is the leftmost minimum size separator due to the construction of the c_P 's (check $C_{\mathcal{P}}$ in Figure 3.2 and S in Figure 3.3).

Now, we present pseudocode for the algorithm described above with a tiny modification. The modification is that we do not start from scratch necessarily and we pass a subset $\mathcal{P}$ of disjoint paths to the algorithm in the beginning. Then, the algorithm tries to find a $\mathcal{P}$ -augmenting walk Q , updates $\mathcal{P}$, and recurses until no augmenting walk is found. The original algorithm can be implemented by initializing $\mathcal{P} = \emptyset$. The reason why we need this modification is described in Section 3.3.

The following are the global variables used in Algorithms 6, 7, and 8.

⁴Again, for more details refer to [23].

⁵We have borrowed the notation for this part from [23].

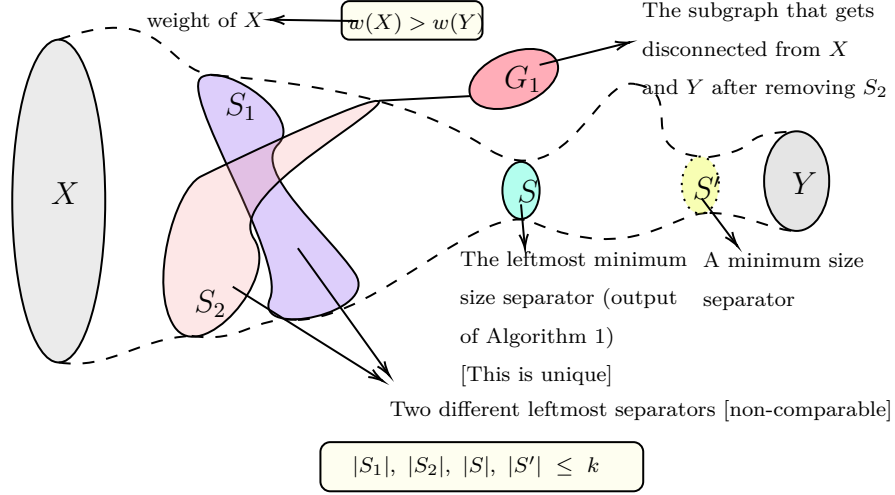


Figure 3.3. Leftmost separators vs leftmost minimum size separator

- $X \subseteq V(G)$: the left set
- $\mathcal{A} \subseteq 2^{V(G)}$: the set of all minimal leftmost $(X, Y, \leq k)^G$ -separators
- $k \in \mathbb{N}$: a given upper bound on the treewidth

Observation. Note that the output of Algorithm 6 is unique and well-defined.

Algorithm 6 *Left_Minimum_Sep*($G, Y, \mathcal{P}$): FIND THE LEFTMOST MINIMUM SEPARATOR

Require: Graph $G = (V, E)$, a subset of vertices Y , and a set $\mathcal{P}$ of pairwise disjoint paths from X to Y

Ensure: The leftmost minimum $(X, Y, \leq k)^G$ -separator, and an updated set of pairwise disjoint paths $\mathcal{P}$

- 1: **while** $\exists$ a $\mathcal{P}$ -augmenting walk Q **do**
 - 2: Update $\mathcal{P}$. $\triangleright$ by sending a unit flow through the edges of $\mathcal{P}$ and Q . Also, $|\mathcal{P}|$ is increased by 1
 - 3: **end while**
 - 4: Construct $R(\mathcal{P}) = \{v \in V \mid \exists \text{ a } \mathcal{P}\text{-augmenting walk from } X \text{ to } v\}$ using DFS.
 - 5: Initialize $C(\mathcal{P}) \leftarrow \emptyset$
 - 6: **for** $P \in \mathcal{P}$ **do**
 - 7: $c_P \leftarrow$ the first vertex $\in P$ and $\notin R(\mathcal{P})$.
 - 8: $C(\mathcal{P}) \leftarrow C(\mathcal{P}) \cup \{c_P\}$.
 - 9: **end for** **return** $(C(\mathcal{P}), \mathcal{P})$.
-

Algorithm 7 *Init*(G, Y): INITIALIZATION

Require: Graph $G = (V, E)$

- 1: $\{S, \mathcal{P}\} \leftarrow \text{Left_Minimum_Sep}(G, Y, \emptyset)$. $\triangleright$ S is the minimum size leftmost $(X, Y, \leq k)$ -separator and $\mathcal{P}$ is a set of pairwise disjoint paths
 - 2: push all the vertices in S onto empty stack R .
 - 3: $\text{Branch}(G[V_{X,S} \cup S], S, S, \emptyset, \mathcal{P}, R, \text{True})$.
-

Algorithm 8 *Branch*($G, S, Y, I, \mathcal{P}, R, \text{LEFTMOST}$): THE MAIN PROCEDURE FOR FINDING ALL LEFTMOST $(X, Y, \leq k)$ -SEPARATORS

Require: Graph G , a separator $S \in S_{X,Y}$, a subset of vertices Y , I : the included vertices, $\mathcal{P}$: a set of pairwise disjoint paths between X and Y , R : the stack to hold the order of handling vertices, and *leftmost*: a boolean indicating whether we have a leftmost separator.

- 1: **if** $I == S$ **then**
 - 2: $A \leftarrow A \cup \{S\}$
 - 3: **else**
 - 4: pop v from R
 - 5: $Y' \leftarrow (S \setminus \{v\}) \cup (N(v) \cap V_{X,S})$
 - 6: $\{S', \mathcal{P}'\} \leftarrow \text{Left_Minimum_Sep}(G, Y', \mathcal{P})$
 - 7: **if** $|S'| \leq k \wedge I \subseteq S'$ **then**
 - 8: $\text{leftmost} \leftarrow \text{False}$
 - 9: **if** $|S'| < k$ **then**
 - 10: let R' be a copy of R . Push all vertices of $(N(v) \cap V_{X,S})$ onto R'
 - 11: $\text{Branch}(G[V_{X,S'}], S', Y', I, \mathcal{P}', R', \text{True})$
 - 12: **end if**
 - 13: **end if**
 - 14: **if** $(|S \setminus I| \geq 2 \vee \text{leftmost}) \wedge (|S| \leq k)$ **then**
 - 15: $\text{Branch}(G[V_{X,S}], S, Y, I \cup \{v\}, \mathcal{P}, R, \text{leftmost})$
 - 16: **end if**
 - 17: **end if**
-

3.3 Finding All Minimal Leftmost $(X, Y, \leq k)$ -Separators

In this section, we present our main algorithm. In the introduction, we mentioned why it is important to find the leftmost⁶ separators. Also, in Section 3.2 we reviewed an algorithm (Algorithm 6) to find the leftmost minimum separator. We use this algorithm in ours.

In our problem, we have two subsets of vertices X and Y such that $|X| \geq |Y|$. W.l.o.g., assume X is the set on the left and Y is the set on the right.

⁶We drop the term “minimal” because it is the default for the separators throughout this paper unless mentioned otherwise.

Problem 2. Given a graph $G = (V, E)$, sets $X, Y \subseteq V$, and $k \in \mathbb{N}$, what are the minimal leftmost $(X, Y, \leq k)^G$ -separators?

Theorem 3.3.1. Given a graph $G = (V, E)$, sets $X, Y \subseteq V$, and $k \in \mathbb{N}$, there exists an algorithm which solves Problem 2 in time $\mathcal{O}(2^{2k}\sqrt{kn})$

Proof. We present a recursive branching algorithm (Algorithm 8). Initially, it calls Algorithm 6 to find the leftmost minimum size $(X, Y, \leq k)$ -separator (using a simple flow algorithm), namely S by feeding X as the left set, Y as the right set, and an empty set of pairwise disjoint paths (namely $\mathcal{P}$) from X to Y (inherited from the parent branch). Notice that the leftmost minimum size separator is unique. This is the root (namely r) of the computation tree (namely $\mathcal{T}$). Let us refer to the computation subtree rooted at node x , as $\mathcal{T}(x)$.

In each node x of $\mathcal{T}$ with the corresponding graph G_x , let Y_x , I_x , and $\mathcal{P}_x$ be the right set, the set of vertices that we require to be in the separator, and the set of disjoint paths inherited from the parent's node, respectively. Notice that we do not pass X_x (the left set) as an argument since it does not change throughout the algorithm and hence we have defined it as a global variable ($\forall x, X_x = X$).

Claim 3.3.1. Let S be a minimal leftmost $(X, Y, \leq k)^G$ -separator, and S' be the separator generated by Algorithm 6. Then, $S \subseteq V_{X,S'} \cup S'$.

Proof. For the sake of contradiction, assume that $\exists v \in S$ such that $v \notin V_{X,S'} \cup S'$. This means that $v \in V_{S',Y} \cup V_Z$.

If $v \in V_Z$, then $S \setminus \{v\}$ is still a leftmost $(X, Y, \leq k)^G$ -separator which contradicts the minimality of S .

The other possibility is that $v \in V_{S',Y}$. Let S_{out} be the set of all such vertices; i.e., $S_{out} = \{v \in S \mid v \in V_{S',Y}\}$. Any $X - S_{out}$ -path⁷ goes through S' , otherwise S' would not be a separator. Let S'_{in} be the set of all vertices of S' that are on a path from vertex of X to a vertex of S_{out} . Now, let $S'' = (S \setminus S_{out}) \cup S'_{in}$. Hence, S'' is an $(X, Y, \leq k)^G$ -separator⁸ such that $S'' \preceq S$ with $|S''| \leq |S| \leq k$, which contradicts that S is a leftmost $(X, Y, \leq k)$ -separator. $\square$

Claim 3.3.1 allows us to ignore the subgraph $G[V_{S,Y} \cup V_Z]$ and focus only on the graph to the left of the current separator and keep moving towards left until it is impossible.

⁷any path from a vertex in X to a vertex in S_{out}

⁸TODO, why is at most $\leq k$

Let S_x be the separator found by Algorithm 6 while processing node x from the computation tree. If S_x is a leftmost $(X, Y, \leq k)$ -separator⁹, this branch terminates and we add S_x to the set of all the leftmost $(X, Y, \leq k)^G$ -separators, namely $\mathcal{A}$ ($\mathcal{A}$ is a global variable). Otherwise, we keep pushing the separator to the left by branching 2-fold. Let us call the children of x by c_1 and c_2 . If S_x was not a leftmost $(X, Y, \leq k)^{G_x}$ -separator¹⁰, this means that there exists at least one leftmost $(X, Y, \leq k)^{G_x}$ -separator, namely S' such that $S' \neq S$ and $S' \preceq S$. Also, as a result of Claim 3.3.1, S' is a leftmost $(X, Y, \leq k)^G$ -separator, too.

In each node x , we call Algorithm 6 to find the minimum separator S_x of size $\leq k$ between X and Y_x . Now, we push all the vertices of S_x onto stack R . Then, we pop vertex v which is on top of the stack R and consider the following two scenarios (corresponding to c_1 and c_2).

1. If we want v to belong to the leftmost separators. In this case, we add v to set I ¹¹, which is the set of vertices that we require to be in all the leftmost separators in $\mathcal{T}_{c_1}$.
2. If v does not belong to the leftmost separators. Here, we pop v and push the left neighbors of v (i.e., $N(v) \cap V_{X, S_x}$) onto R (we just move to the left due to Claim 3.3.1).

Notice that the order of handling vertices is not important but we use stack because it simplifies the proof later on. Every produced separator is a leftmost $(X, Y, \leq k)^G$ -separator because the only time that one branch terminates is when it finds a leftmost separator. Now, we show that all the leftmost $(X, Y, \leq k)^G$ -separators are generated by the algorithm given.

Let S_0 be an arbitrary leftmost $(X, Y, \leq k)^G$ -separator, and as before let S be the separator generated by Algorithm 6. At this point R is filled with the vertices of S . Pop v from R .

- If $v \in S_0$, then put v in I , and recurs.
- If $v \notin S_0$, push $N(v) \cap V_{X, S}$ into R , recurs.

This determines an exact computation branch. All branches halt with a minimal leftmost separator since each time we go at least one more to the left. So, this branch terminates

⁹Later, we explain how this is done

¹⁰note that by Claim 3.3.1 $G_x = G[V_{X, S} \cup S]$

¹¹Include

with minimal leftmost separator S'_0 as well. All the vertices of S_0 are pushed into R at some point because otherwise this branch terminates with a separator which is not leftmost and we can push it more to the left. At the end of this branch, $I \subseteq S'_0$. On the other hand, $I = S_0 \subseteq S'_0$, which implies that $S_0 = S'_0$ because both of them should be minimal. $\square$

In [28], the authors mention that the number of important $(X, Y, \leq k)^G$ -separators is $\leq 4^k$, and they even mention that this upper bound can be tight by a polynomial factor. Here, we give a precise upper bound for the number of leftmost $(X, Y, \leq k)^G$ -separators and the number of important $(X, Y, \leq k)^G$ -separators and we show that both bounds are tight.

Before that, let us review the definition of the Catalan numbers.

Definition 3.3.1. *Catalan numbers* is a sequence of numbers where the n -th Catalan number is:

$$C_n = \binom{2n}{n} - \binom{2n}{n+1} = \frac{1}{n+1} \binom{2n}{n} \sim \frac{4^n}{n^{3/2} \sqrt{\pi}}.$$

Theorem 3.3.2. Let $G = (V, E)$ be a graph and let $X, Y \subseteq V$ and let $k \in \mathbb{N}$. The number of leftmost $(X, Y, \leq k)^G$ -separators is at most C_{k-1} and the number of important $(X, Y, \leq k)^G$ -separators is at most $\sum_{i=0}^{k-1} C_i$. Furthermore, both upper bounds are tight.

Before giving a proof, we mention some definitions.

Definition 3.3.2. A *full k -paranthesization* is a string over the alphabet $\{[,]\}$ consisting of k “[” and k “]” having more “[” than “]” in every non-trivial prefix. This forms a language slightly different from the Dyck language (a well-known language). We call this language *restricted Dyck language* and any string in this language is called a *restricted Dyck word*.

Notice that $[x]$ is a restricted Dyck word iff x is a Dyck word¹².

Full paranthesizations (the restricted Dyck language over the alphabet $\{[,]\}$) are generated by the following context-free grammar $G = (\{S, A\}, \{[,]\}, R, S)$, where the set of the rules R is:

$$\begin{aligned} S &\rightarrow [A] \mid \epsilon \\ A &\rightarrow AA \mid [A] \mid \epsilon \end{aligned}$$

¹²Notice the difference between the restricted Dyck Language and the Dyck language itself. In every non-trivial prefix of a Dyck word, the number of “[” is $\geq$ the number of “]”, whereas for the restricted version it should be strictly greater.

Definition 3.3.3. The k -parentheses tree is the binary tree (V, E) with the following properties.

- V is the set of prefixes of full k' -parenthesizations for $0 < k' \leq k$.
- If “ x ” is in V , then “ $x[$ ” is the left child of x , and if “ $x]$ ” is in V , then “ $x]$ ” is the right child of x .
- If “ $x[$ ” or “ $x]$ ” are not in V , then x has no left or right child respectively.

Definition 3.3.4. The *compact* $\leq k$ -parentheses tree is obtained from the $\leq k$ -parentheses tree by removing all nodes containing k “[” and ending in “]”.

Note that the nodes with k “[” ending in “]” have been removed, because no branching happens at their parents, as there are no left children.

The compact $\leq k$ -parentheses tree is a full binary tree, every node has 0 or 2 children. The number of leaves in this tree is equal to the number of full $\leq k$ -parenthesizations, which is equal to $\sum_{k'=1}^{k-1} C_{k'} = \mathcal{O}(C_k)$ where C_k is the k -th Catalan number. An immediate consequence is that the number of nodes in the compact $\leq k$ -parentheses tree is $2 \sum_{k'=1}^{k-1} C_{k'} - 1 = \mathcal{O}(C_k)$.

For the algorithm to find all leftmost $(X, Y, \leq k)$ -separators, the worst case computation tree is the compact $\leq k$ -parentheses tree with the nodes representing full k' -parenthesizations for $1 \leq k' \leq k-1$ removed.

An arbitrary computation tree for the algorithm to find all leftmost $\leq k$ -separators is obtained from the worst case computation tree by the following two operations.

- For any node x , the subtree rooted at the left child $c_1(x)$ may be just removed or replaced by the subtree rooted at $c_1(c_1(x))$. This splicing operation can be repeated to jump down arbitrarily far.
- For any node x with less than or equal to k “[”s, let x_i be x concatenated with i “]”s. If x_{i^*} is a full k' -parenthesization for some k' with $0 < k' < k$, and the nodes $x_{i'}$ for $0 \leq i' < i^*$ have no left children, then the leaf x_{i^*} is added. (In the added node x_{i^*} , a minimal separator S of size $< k$ is picked, because there is no larger minimal $\leq k$ -separator to the left of S .)

Note that the number of leaves in an arbitrary computation tree for the algorithm to find all $(X, Y, \leq k)$ -separators is $\leq C_{k-1}$, because for every leaf “ $x]$ ” added in (b), at least one leaf, namely “ $x[\cdots [] \cdots]]$ ” (full parenthesization of length k) has been removed in (a).

For the recursive procedure, we introduce a boolean parameter *leftmost*. The parameter is originally true in the root and in every node that is a left child. If a node has a left child, i.e., excluding v , a new larger separator S' with $|S'| \leq k$ has been found, then *leftmost* is set to false. The current truth value of *leftmost* is passed to the right child (include v call). If x contains $k' < k$ “[” and $k' - 1$ “]”, then a right call (including v) is only made if *leftmost* is true, i.e., there is no larger $(X, Y, \leq k)$ -separator to the left of the current small (size $< k$) separator.

In every node, S is the minimum *leftmost* separator between and including X and the current Y .

- In the root, Y is the original Y . $I = \emptyset$, and *leftmost* = *true*. The node is represented by a sequence of “[”s of length $|S|$.
- A node x with $|S| = k' < k$, is represented by a prefix x of a full parenthesization with k' “[” and $|I|$ “]”.

[Alternatively, one could include a root in the computation tree corresponding to the empty 0-parenthesization. In such a root, no S is defined. The minimum *leftmost* separator S' is computed. A left call is made if $|S'| \leq k$. No right call is made. In this view, the root node has only one child. Not including such a root node means that this node is not handled by the recursive procedure, but by a different calling procedure.]

Now, here is the proof of Theorem 3.3.2.

Proof. First, we prove the result corresponding to the *leftmost* separators. As explained above, we show excluding a vertex by a “[”, which corresponds to a left branch. Analogously, “]” denotes including a vertex in I (requiring the *leftmost* separator to have v in it), which corresponds to the right branch.

Now, we see why we used a stack to handle branching on the vertices. Even though the order does not matter but we need a stack to have a nice correspondence between the algorithm behavior and the restricted Dyck words. It is well known that the number of the Dyck words of length $2k$ is C_k . So, the upper bound on the number of *leftmost* $(X, Y, \leq k)$ -separators is $C_{k-1} = \frac{\binom{2(k-1)}{k-1}}{k} \sim \frac{4^{k-1}}{(k-1)^{3/2}\sqrt{\pi}}$. We also close the gap and show that this bound is tight.

Let $\hat{G}$ be a complete binary tree with depth at least $k + 1$. Let X be the set of all the leaves of $\hat{G}$, and Y be the root. (see Figure 3.4). The number of full subtrees with exactly k leaves is C_{k-1} . On the other hand in $\hat{G}$, no $(X, Y, \leq k - 1)^{\hat{G}}$ -separator is a *leftmost* $(X, Y, \leq k)^{\hat{G}}$ -separator since we can replace a node with its both children and

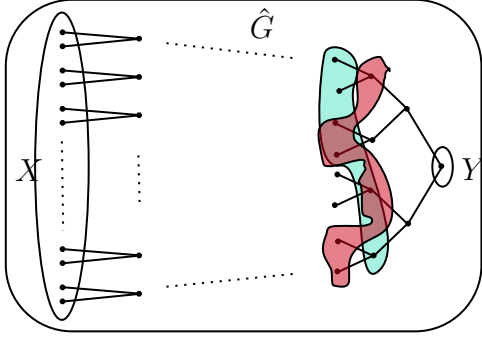


Figure 3.4. Here, $k = 6$. We illustrate two out of 42 minimal leftmost $(X, Y, \leq 6)^{\hat{G}}$ -separators. Also, there are $1 + 2 + 5 + 14 + 42 = 64$ important $(X, Y, \leq 6)^{\hat{G}}$ -separators.

move more to the left. So, the number of minimal leftmost $(X, Y, \leq k)^{\hat{G}}$ -separators is exactly C_{k-1} .

For the upper bound on the number of important separators, we use Lemma 3.1.2 and this immediately implies that the number of important $(X, Y, \leq k)^{\hat{G}}$ -separators $\leq \sum_{i=1}^{k-1} C_i$.

For the tightness part, assume the same $\hat{G}$ with the same X and Y . The number of important $(X, Y, \leq k)^{\hat{G}}$ -separators is $\sum_{i=1}^{k-1} C_i$ since based on Lemma 3.1.2, the number of important $(X, Y, \leq k)^{\hat{G}}$ -separators is $\leq \sum_{i=1}^{k-1} (\text{the number of leftmost } (X, Y, \leq k)^{\hat{G}}\text{-separators})$. On the other hand, as we mentioned earlier, in $\hat{G}$, no leftmost $(X, Y, \leq k-1)^{\hat{G}}$ -separator is a leftmost $(X, Y, \leq k)^{\hat{G}}$ -separator. Hence, the number of important $(X, Y, \leq k)^{\hat{G}}$ -separators is exactly $\sum_{i=1}^{k-1} (\text{the number of leftmost } (X, Y, \leq k)^{\hat{G}}\text{-separators})$, which is equal to $\sum_{i=1}^{k-1} C_i$. □

Notice that $\sum_{i=1}^{k-1} C_i = \Theta(\frac{4^{k-1}}{(k-1)^{3/2}})$.

Remark 3.3.1. Let $\mathcal{A}_{X,Y,\leq}^G$ and $\mathcal{B}_{X,Y,\leq}^G$ be the set of all leftmost $(X, Y, \leq k)^G$ -separators and the set of Based on Lemma 3.1.2, the set of all important $(X, Y, \leq k)^G$ -separators that are not leftmost $(X, Y, \leq k)^G$ -separator is the union of the set of all leftmost $(X, Y, \leq i)^G$ -separator, for $i = 1, \dots, k-1$.

Remark 3.3.2. Let $\hat{G}$ be a complete binary tree with depth at least $k+1$, $\hat{X}$ be the set of all leaves of $\hat{G}$, and $\hat{Y}$ be the root. For a fixed $k \in \mathbb{N}$,

1. $\hat{G}$ has the highest number of leftmost $(X, Y, \leq k)^G$ -separators among all graphs like G and for all $X, Y \subseteq V(G)$ and it happens by setting $X = \hat{X}$ and $Y = \hat{Y}$.

2. $\hat{G}$ has the highest number of important $(X, Y, \leq k)^G$ -separators among all graphs like G and for all $X, Y \subseteq V(G)$ and it happens by setting $X = \hat{X}$ and $Y = \hat{Y}$.
3. $\hat{G}$ has the highest number of important but not leftmost $(X, Y, \leq k)^G$ -separators among all graphs like G and for all $X, Y \subseteq V(G)$ and it happens by setting $X = \hat{X}$ and $Y = \hat{Y}$.

Theorem 3.3.3. Let $G = (V, E)$ be a graph, $X, Y \subseteq V$, and $k \in \mathbb{N}$. There is an algorithm which finds all minimal leftmost $(X, Y, \leq k)^G$ -separators in time $\mathcal{O}(\frac{4^k}{\sqrt{k}}n)$.

Proof. Our algorithm searches all the possibilities and enumerates all the possible leftmost separators. We showed that the number of leaves, which is the number of leftmost separators is $\mathcal{O}(C_{k-1})$. So, we have $\mathcal{O}(C_{k-1})$ nodes in our computation tree and work done in every node is $\mathcal{O}(kn)$ (the running time of a simple flow algorithm). Hence, the total running time is $\mathcal{O}(\frac{2^{2k}}{\sqrt{k}}n)$. $\square$

3.4 Application to Treewidth Approximation

As mentioned in the introduction, Reed's 5-approximation algorithm [13] for treewidth runs in time $\mathcal{O}(2^{24k}(k+1)!n \log n)$. This was improved to a 5-approximation algorithm running in $\mathcal{O}(k^2 2^{8.766k}n \log n)$ [8]. Here, we further improve the treewidth approximation algorithm. In order to do so, we briefly describe the algorithms given in [13], and [8]. Hence, we review some notations and for the others we give references.

For a graph $G = (V, E)$ and a subset W of the vertices, $G[W]$ is the subgraph induced by W . For the sake of simplicity throughout this paper, let $G - W$ be $G[V \setminus W]$ and $G - v$ be $G - \{v\}$ for any $W \subseteq V(G)$ and any $v \in V(G)$.

Also, in a weighted graph, a non-negative integer weight $w(v)$ is defined for each vertex v . For a subset W of the vertices, the weight $w(W)$ is simply the sum of the weights of all vertices in W . Furthermore, the total weight or the weight of G is the weight of V .

Definition 3.4.1. A *tree decomposition* of a graph $G = (V, E)$ is a tree $\mathcal{T} = (V_{\mathcal{T}}, E_{\mathcal{T}})$ such that each node x in $V_{\mathcal{T}}$ is associated with a set B_x (called the bag of x) of vertices in G , and $\mathcal{T}$ has the following properties:

- $\bigcup_{x \in V_{\mathcal{T}}} B_x = V(G)$
- $\forall \{u, v\} \in E, \exists x \in V_{\mathcal{T}} : u, v \in B_x$
- $\forall x, y \in V_{\mathcal{T}}, \forall z \in V_{\mathcal{T}}$ on the path connecting $x, y \in V_{\mathcal{T}}: B_x \cap B_y \subseteq B_z$.

Even though historically the *width of a tree decomposition* is defined to be the size of its largest bag minus one, in this paper we follow Reed's definition [13] and define the width of a tree decomposition to be the size of its largest bag¹³.

The *treewidth* of a graph G is the minimum width over all tree decompositions of G called $tw(G)$. In the following, we use the letter k for the treewidth.

Definition 3.4.2. A *nice tree decomposition* is a tree decomposition which is rooted and its every node has at most two children. Any node x in a nice tree decomposition $\mathcal{T}$ is of one of the following types (let c be the only child of x or let c_1 and c_2 be the two children of x):

- *Leaf* node, a leaf of $\mathcal{T}$ without any children.
- *Forget* node (forgetting vertex v), where $v \in B_c$ and $B_x = B_c \setminus \{v\}$,
- *Introduce vertex* node (introducing vertex v), where $v \notin B_c$ and $B_x = B_c \cup \{v\}$,
- *Join* node, where x has two children with the same bag as x , i.e. $B_x = B_{c_1} = B_{c_2}$.

It has been shown that any given tree decomposition can be converted to a nice tree decomposition with the same width in polynomial time [?].

Definition 3.4.3. A centroid of a weighted tree T is a node x such that none of the trees in the forest $T - x$ has more than half the total weight.

For nice tree decompositions¹⁴, we choose a stronger version of centroid for this paper.

Definition 3.4.4. A strong centroid of a nice tree decomposition $\mathcal{T}$ of a graph $G = (V, E)$ with respect to $W \subseteq V$ is a node x of $\mathcal{T}$ such that none of the connected components of $G - B_x$ contains more than $\frac{1}{2}|W \setminus B_x|$ vertices of W .

The following lemma shows there existence of a strong centroid for any given $W \subseteq V$.

Lemma 3.4.1. For every nice tree decomposition $(\mathcal{T}, \{B_x : x \in V_{\mathcal{T}}\})$ of a graph $G = (V, E)$ and every subset $W \subseteq V$, there exist a strong centroid with respect to W .

The proof of Lemma 3.4.1 can be found in [8].

¹³For the sake of simplicity in the computation

¹⁴The property that we use is that the bags of the adjacent nodes in a nice tree decomposition differ by at most one vertex. In fact, any tree decomposition with adjacent bags differing in at most one vertex works just fine.

Definition 3.4.5. Let $G = (V, E)$ be a graph and $W \subseteq V$. A *balanced W -separator* is a set $S \subseteq V$ such that every connected component of $G - S$ has at most $\frac{1}{2}|W|$ vertices.

Lemma 3.4.2. [23, Lemma 11.16] Let $G = (V, E)$ be a graph of treewidth at most $k - 1$ and $W \subseteq V$. Then there exists a balanced W -separator of G of size at most k .

Definition 3.4.6. Let $G = (V, E)$ be a graph and $W \subseteq V$. A *weakly balanced separation* of W is a triple (X, S, Y) , where $X, Y \subseteq W$, $S \subseteq V$ are pairwise disjoint sets such that:

- $W = X \cup (S \cap W) \cup Y$.
- S separates X from Y .
- $0 < |X|, |Y| \leq \frac{2}{3}|W|$.

Lemma 3.4.3. [23, Lemma 11.19] For $k \geq 3$, let $G = (V, E)$ be a graph of treewidth at most $k - 1$ and $W \subseteq V$ with $|W| \geq 2k + 1$. Then there exists a weakly balanced separation of W of size at most k .

Theorem 3.4.1. [23, Corollary 11.22] For a graph of treewidth at most $k - 1$ with a given set $W \subseteq V$ of size $|W| = 3k - 2$, a weakly balanced separation of W can be found in time $O(2^{3k}k^2n)$.

3.4.1 Summary of Reed's algorithm

Reed finds a weakly balanced separator for the set of “representatives” as follows (it has been described in [13] and the details have been filled in [8]):

- Do a DFS on G and find the deepest vertex $v \in V(G)$ whose subtree has at least $\frac{n}{24k}$ vertices. Let us call v a representative. Also, define the weight of v to be the size of this subtree, denoted as $w(v)$.
- Let W be the set of all representatives.
- Form a balanced separator S of size $\leq k$, partitioning $G - S$ into X and Y as follows:
 - Decide if any representative is going to go into S . Branch 2-fold and consider both scenarios.
 - * Scenario I: At least a representative like v is going into S . Place v into S , decrease k by one, do a DFS from scratch and form a new set of representatives and recurs.

* *Scenario II: Branch on every representative like v going into X or Y . The crux of the idea is that if a representative goes into one side, most probably its corresponding subtree will also go into the same side. It is because if v goes into let us say X and one of the proper descendants of one of its children in its corresponding subtree like u goes into Y , then the path from v to u should go through S , which means it requires a vertex from its subtree to be in S . However, the size of S is bounded by k . So, not more than k such incidents can happen. This means at most $k \cdot \frac{n}{24k}$ vertices might go to the opposite side of their corresponding representative (notice that no children of v has $\geq \frac{n}{24k}$ vertices in its subtree). This means the error is $\leq \frac{1}{24}n$. This property allows Reed to work with the representatives (there are $\leq 24k$ of them), rather than all the vertices. Branching on every vertex results in an exponential-time algorithm in term of n but it does not happen when we work with the representatives.*

Notice that we separated the branching on X and Y from the branching on S . It is because if a representative goes into S , we have no control over its corresponding subtree.

Because of the error mentioned above, $\frac{1}{3}n - \frac{1}{24}n \leq w(X), w(Y) \leq \frac{2}{3}n + \frac{1}{24}n$. These bounds are in terms of weight while we want a balanced separator in terms of the exact size. So, in order to go from a separator in terms of weight to a separator in terms of the volume (actual size), we might have to pay up to the error, once more. So, the separator that we find for W , separates the entire graph into L , and R such that $\frac{1}{3}n - \frac{1}{24}n - \frac{1}{24}n \leq |L|, |R| \leq \frac{2}{3}n + \frac{1}{24}n + \frac{1}{24}n$, which means $\frac{1}{4}n \leq |L|, |R| \leq \frac{3}{4}n$.

Once Reed obtains X and Y , uses the flow algorithm to find the leftmost minimum size $(X, Y, \leq k)$ -separator, which is unique.

Now, $G - S$ is separated into two sides L and R , such that $\frac{1}{4}n \leq |L|, |R| \leq \frac{3}{4}n$. We recurs on subproblems with the inputs $G[X \cup S]$ and $G[Y \cup S]$. Then, the algorithm finds a tree decomposition for each subproblem and finally merges them together to obtain a tree decomposition for G .

Based on Lemma 3.4.1 there exists a strong centroid with respect to W . We do not necessarily know what it is but we sure know that it exists. Reed's algorithm checks all the partitions and one of them is in fact the centroid. So, this is why the algorithm definitely finds a balanced separator if it exists.

3.4.2 Further Previous Improvement

Recently, the authors of this paper, improved Reed's algorithm so that it runs in time $\mathcal{O}(k^2 2^{8.766k} n \log n)$. Here, we briefly describe how this improvement has been achieved.

As mentioned earlier, Reed picks 24 but any constant $C_\epsilon > 6$ gives us two subproblems with sizes ϵn and $(1 - \epsilon)n$ (or better), for any small and nonzero ϵ . In Reed's algorithm, $\epsilon = \frac{1}{4}$, and $C_\epsilon = 24$. We can fix ϵ and C_ϵ , later.

There are two major improvements in [8].

- First improvement is that we do not separate scenarios *I* and *II* as in Reed's algorithm. We consider one representative at a time and once we are working on representative v , we branch on v going into the separator or not. If it goes into the separator, we put v there but this time we do not do the DFS from scratch to form a new group of representatives (which is costly). We undo the DFS for the subtree rooted at v (the subtree that v represents) and continue the DFS.
- Second, we do not construct a bipartition at the very beginning after removing S . Reed starts with looking for a weakly balanced separator ($\frac{1}{3}n$ to $\frac{2}{3}n$), which is known to exist. However, he works with the weights and as argued before, he makes a bipartition based on the weights but the volumes can be $\frac{1}{4}n$ to $\frac{3}{4}n$ (or better). In [8], our argument starts with a restricted balanced separator by volume. We allow more than two partitions (multi partition by volume for the analysis). Each part has volume of at most $\frac{1}{2}$. This means each part has weight up to $3/4 - \epsilon/2$. Only now, we form a two-partition (after applying the weights) by weight. Each part has weight $\leq \max\{3/4 - \epsilon/2, 2/3\}$ (for $\epsilon \leq 1/6$). Then, we find a separator for this weight partition. The larger part has volume between $1/4 + \epsilon$ and $1 - \epsilon$. So, in worst-case scenario, we end up with an ϵ to $1 - \epsilon$ partition, which is still good. Hence, we set $C_\epsilon = \frac{1}{((1-\epsilon)-1/2)/2} = \frac{4}{1-2\epsilon} \leq 4 + 12\epsilon$, for $\epsilon \leq 1/6$.

Our time analysis in [8] shows that the running time complexity to split based on W is as given below, where t is the number of subtrees (or representatives), and k is the upper bound on treewidth.

$$T_W(t, k) \leq T_W(t, k-1) + 2T_W(t-1, k) + Qkn + \mathcal{O}(1), \quad (3.1)$$

where Q is the constant which shows up in the time complexity of the flow algorithm.

We have shown that finally,

$$T_W(t, k) \leq Qk^2ne^k(C_\epsilon + 1)^k(2^{(C_\epsilon+1)k}(C_\epsilon + 1) + 2). \quad (3.2)$$

The following theorem can be found in [8].

Theorem 3.4.2. Let $C_0 = \log_2 e + \log_2 5 + 5 < 8.765$. For every $C > C_0$, a 5-approximation algorithm of the treewidth can be computed in time $\mathcal{O}(2^{Ck}n \log n)$.

3.4.3 Our Improvement

In this subsection, our goal is to use the algorithm for finding the leftmost separators to further improve the coefficient of k in the exponent of the tree decomposition algorithm to make it more applicable.

- For the analysis, we consider a centroid by volume, namely C . It has size $k' \leq k$.
- Each connected component of $G - C$ has volume at most $\frac{1}{2}(n - k')$.
- These connected components can be grouped into 3 parts, each with volume at most $\frac{1}{2}(n - k')$. (Just place the components by decreasing volume into the part with currently smallest volume.)
- Let the proper volume be the part of the volume that has its corresponding weight in the same part. In other words, the proper volume is the number of vertices whose representative is in the same part.
- Let t be the threshold for the size of the small trees. At most $k'(t - 2) = k'((\frac{1}{2} - \epsilon)n/k - 2) \leq (\frac{1}{2} - \epsilon)n - 2k'$ vertices can be in a different part than their representative. Therefore, the total proper volume is at least $n - k' - (\frac{1}{2} - \epsilon)n + 2k' \geq (\frac{1}{2} + \epsilon)n + k'$.
- Of the proper volume, at least $(\frac{1}{2} + \epsilon)n + k' - \frac{1}{2}(n - k') > \epsilon n$ is not in the part with largest proper volume.
- Therefore, there are at least 2 parts with proper volume at least $\frac{\epsilon n}{2}$.
- Of these 2 parts, we put the part with larger weight on the left side, the other one on the right side.
- We also put the third part on the left side.

- The left part has weight at least half the total weight, which is $\frac{1}{2}(n - k')$.
- The right part has weight at most $\frac{1}{2}(n - k')$ and (proper) volume at least $\frac{\epsilon n}{2}$

The algorithm tries all possible 2-partitions of the representatives. This includes the left-right partition that we are currently investigating. While searching for a leftmost separator, the centroid is a competitor. Thus the algorithm finds a separator that is equal to the centroid or is located strictly to the left of it. From now on, left (call it X) and right (namely Y) are defined by the leftmost $(X, Y, \leq k)^G$ -separator found by Algorithm 7. This separator has size k'' with $k' \leq k'' \leq k$. It produces the same weight partition as the centroid, but part of the volume might shift to the right.

- Thus the left part has still weight at least $\frac{1}{2}(n - k')$ and therefore volume at least $\frac{1}{2}(n - k') - (\frac{1}{2} - \epsilon)n + 2k \geq \epsilon n + \frac{3}{2}k$.
- The right part has still volume at least $\frac{\epsilon n}{2}$
- The recursive calls are done with the subgraphs induced by the union of the vertices of a connected component with the vertices of the separator. Their number of vertices is upper bounded by n minus the volume of the smaller side. It is less than $(1 - \frac{\epsilon}{2})n$.
- After $\frac{2 \ln 2}{\epsilon}(\log n - \log b) = O(\log n)$ rounds for $b \geq k$, the largest volume of a recursive call is at most b .

In the worst case, the algorithm alternates between a split by volume and $\log k$ splits of W steps. Let the time spent between two splits by volume be at most $f(k)n$. Note that $f(k) \leq g(k) + 3^{3k}k \log k = O(g(k))$, where $g(k)$ is the time of one split by volume step. Then we get the following recurrence for an upper bound on the running time of the whole algorithm.

$$T(n) = \begin{cases} O(k) & \text{if } n \leq 3k \\ \max_{p, n_1, \dots, n_p} \{ \sum_{i \in [p]} T(n_i) \} + f(k)n & \text{otherwise,} \end{cases}$$

where the maximum is taken over $p \geq 2$ and $n_1, \dots, n_p \in [n-1]$ such that $\sum_{i \in [p]} (n_i - k) = n - k$. Note that $\sum_{i \in [p]} (n_i - k'') = n - k''$ reflects that every recursive call includes a connected component of $G - S$ together with the separator S of size k'' . We can round up k'' to k , because $T(n)$ is an increasing function. Because, the sum of the n_i 's is more

than n , it is beneficial to consider the following modified function $T'(n') = T(n + k)$. Then we get the simpler recursion

$$T'(n') = \begin{cases} O(k) & \text{if } n' \leq 2k \\ \max_{p, n'_1, \dots, n'_p} \{ \sum_{i \in [p]} T'(n'_i) \} + f(k)(n' + k) & \text{otherwise,} \end{cases}$$

where the maximum is taken over $p \geq 2$ and $n'_1, \dots, n'_p \in [n' - 1]$ such that $\sum_{i \in [p]} (n'_i) = n'$.

Now we prove

$$T'(n') \leq \frac{c}{\epsilon} f(k) n' \log n'$$

by induction, where c is minimal such that $c \geq 3$ and the base case ($n' \leq 2k$) is satisfied. Assume that the i th component of size n_i is on the side of the separator with smaller volume if and only if $1 \leq i \leq p'$. Let $n_S = \sum_{i \in [p']} n'_i$, and let $n_L = n' - n_S$. Furthermore, let

$$h_S = \sum_{i \in [p']} n'_i \log n'_i \leq \sum_{i \in [p']} n'_i \log \frac{n'}{2} = n_S (\log n' - 1),$$

and

$$h_L = \sum_{i \in \{p'+1, \dots, p\}} n'_i \log n'_i \leq \sum_{i \in \{p'+1, \dots, p\}} n'_i \log n' = n_L \log n'.$$

Recall that $\epsilon n'/2 \leq n_S \leq n'/2$. By the inductive hypothesis, for $n' > 2k$ we have

$$T'(n') \leq \frac{c}{\epsilon} f(k) (h_S + h_L) + f(k)(n' + k)$$

This implies

$$T'(n') \leq \frac{c}{\epsilon} f(k) (n' \log n' - n_S) + f(k)(n' + k)$$

Thus $T'(n') \leq \frac{c}{\epsilon} f(k) n' \log n'$ if $\frac{c}{\epsilon} n_S \geq n' + k$. As $n_S \geq \epsilon n'/2$ and $n' > 2k$, this is the case when $c \geq 3$.

Each split by volume can be done by finding at most $C_{k-1} = \Theta(4^k/k^{3/2})$ separators in time $\mathcal{O}(C_{k-1}kn) = \mathcal{O}(4^k/\sqrt{k})$ for each placement of at most $n/t = k/(1/2 - \epsilon)$ representatives to the left or right side and the placement of at most k vertices into the centroid. These are at most $(2 + 8\epsilon)k$ representatives for $\epsilon \leq 1/4$. Choosing $\epsilon = \Theta(1/k)$ this results in a running time of $f(k)n = \mathcal{O}(\binom{(3+8\epsilon)k}{k} 2^{4k} k^{-1/2} n) = \mathcal{O}(\frac{3^{3k}}{2^{2k}} 2^{4k} n)$ for one split by volume in a graph of size n . Together with the solution of the previous recurrence, we obtain.

Theorem 3.4.3. If a graph has treewidth at most k , then a tree decomposition of width at most $4k - 1$ can be found in time $\mathcal{O}(2^{6.755k} n \log n)$.

Proof. Here, we analyze the running time of the treewidth algorithm that we just described.

First, we check the running time of split by W , denoted by $T_W(k, n)$. Since $|W| \leq 3k$, and $|S| \leq k$, then we have at most k placements into the centroid and $2k$ outside the centroid. Thus,

$$\begin{aligned}
T_W(k, n) &\approx \binom{3k}{k} 2^{2k} \frac{4^k}{\sqrt{k}} n \\
&\approx \frac{3^3 k}{2^2 k} 2^{2k} \frac{4^k}{\sqrt{k}} n \\
&= 2^{(3 \log_2 3 + 2)k} \frac{1}{\sqrt{k}} n \\
&= \frac{2^{2k} 3^{3k}}{\sqrt{k}} n \\
&< \frac{2^{6.755k}}{\sqrt{k}} n
\end{aligned}$$

Therefore, $T_W(k, n) = \mathcal{O}(2^{6.755k} k^{-1/2} n)$.

Now, we do a split by V every $\log_2 k$ step. So, the total running time of our algorithm is:

$$\mathcal{O}\left(\frac{2^{6.755k}}{\sqrt{k}} k \log k n \log n\right) = \mathcal{O}(2^{6.755k} (\log k) \sqrt{k} n (\log n)).$$

Notice that we rounded the exponent up, hence the polynomial part is dominated by this roundup and the total running time is $\mathcal{O}(2^{6.755k} n \log n)$. $\square$

Chapter 4 |

Faster 2-Approximation of the Treewidth

This chapter is based on a joint work with Martin Fürer and Medha Kumar.

Solving many Fixed Parameter Tractable (FPT) graph problems requires computing treewidth and finding a corresponding tree decomposition. It is known that the treewidth can be 2-approximated in time $\mathcal{O}^*(2^{10.755k})n$, where $k \in \mathbb{N}$ is the upper bound on the treewidth and n is the size of the input graph. In this paper, we present a 2-approximation of treewidth that runs in time $\mathcal{O}^*(2^{8k})n$.

Since constructing a tree decomposition is computationally expensive, we usually assume a near-optimal tree decomposition is available, while solving FPT graph problems. However, finding a tree decomposition remains an important problem. In various instances of FPT graph problems the best algorithm for solving a graph problem with a near-optimal tree decomposition in the input, runs faster than the fastest known treewidth algorithm. That is why focusing on improving the treewidth algorithms is vital. In this paper, we address how our new algorithm changes the bottle-neck for some graph problems. In fact, using our algorithm, any graph problem with a best known running time not better than $\mathcal{O}^*(2^{(3+\delta)k})n$ for some $\delta > 0$ with a near-optimal tree decomposition as input, the bottle-neck shifts to solving the problem itself rather than solving the treewidth. This argument can also be used to validate the importance of further improving the dependence on k in the time complexity of the treewidth solver, or the approximation ratio of the treewidth.

4.1 Introduction

Since the notion of treewidth has been introduced [10, 11] in the 1980s, it has played an important role in computer science. Treewidth measures how “close” a given graph is to a tree structure. The more “tree-like” a graph is (i.e. smaller treewidth), the more efficient it is to solve hard problems on the resulting tree decomposition. There are many exact or approximate algorithms designed to decompose graphs into their tree decompositions and compute treewidths. Courcelle’s metatheorem [4] states that every graph property describable in the monadic second-order logic of graphs, can be decided in linear time for the graphs with bounded treewidth. A problem is said to be fixed parameter tractable (FPT) if there exists an algorithm solving the problem in time $f(k)n^{\mathcal{O}(1)}$, where n is the input size, k is the parameter, and $f(\cdot)$ is a computable function. For graphs with bounded treewidth, many NP-complete graph problems obtain FPT algorithms using dynamic programming, i.e. the running time is polynomial in the size of the graph, and the treewidth is the parameter. Luckily, many practical instances have bounded treewidth.

In order to employ a polynomial time dynamic programming algorithm, we need to have access to a “reasonable” tree decomposition of the input graph. However, computing the treewidth and finding an optimal tree decomposition are NP-complete problems themselves [6]. Even though the exact solution is very difficult to find, it turns out that for graphs with bounded treewidth, a constant factor approximation can be found relatively fast. Note that the exact solution is proven to be in FPT, but its running time is very slow. Bodlaender gave the first FPT exact algorithm running in time $2^{\mathcal{O}(k^3)}n$ [16], where k is the treewidth and n is the size of the input graph. The dependence on n is only linear and thus optimal, but the dependence on k is very large. That is why current research focuses on approximate solutions, that will hopefully decrease the time complexity. Historically, the dependence on n was the main concern. Robertson-Seymour gave their 4-approximation $\mathcal{O}(3^{3k}n^2)$ algorithm in 1995 [14]. Lagergren [15] improved the dependence on n and gave an 8-approximation $2^{\mathcal{O}(k \log k)}n \log^2 n$ -time algorithm. Reed [13] improved this dependence to only $\mathcal{O}(n \log n)$ and Belbasi-Fürer [21] showed that the dependence on k in Reed’s 5-approximation algorithm is $\mathcal{O}(2^{24k}k!)$, which is $2^{\mathcal{O}(k \log k)}$. In 2016, Bodlaender et al. [19] presented a linear-time algorithm which is only single-exponential in k . This was a huge breakthrough from a theoretical point of view. However, the dependence on k was still not low enough for practical applications. Recently, Korhonen [29] introduced a linear-time 2-approximation algorithm where the

dependence on k is $\mathcal{O}^*(2^{10.755k})$. This elegant algorithm - inspired by an idea introduced in [30] - is the first 2-approximation linear-time algorithm which is single-exponential in the treewidth.

Initially, Korhonen [29] presented an algorithm with a time bound of $\mathcal{O}^*(7^{4k}4^{4k})n$. The notation $\mathcal{O}^*$ is similar to the $\mathcal{O}$ -notation except that it ignores polynomial factors. Then, Korhonen observed that the exponents of $4k$ can be reduced to $3k$ because for large root bags a 2-partition instead of a 3-partition is sufficient. Furthermore, he has reduced the potential function from $\mathcal{O}^*(7^{3k})n$ to $\mathcal{O}^*(3^{3k})n$, resulting in a running time of $\mathcal{O}^*(3^{3k}4^{3k})n = \mathcal{O}^*(1728^k)n$. The base 3 is inevitable due to the repeated replacements of each splittable bag (defined below) by three bags in a tree decomposition.

There are three criteria in the time complexity as described below. Here, we restrict our focus to only constant-factor FPT approximations. The time complexity criteria are:

- The dependence on n . The best achieved is linear [16, 19, 29], which is optimal.
- The approximation ratio. The best achieved is 2 by Korhonen [29].
- The dependence on k .

We focus on improving the third criterion. The first criterion is already optimal, and we might expect that finding an approximation ratio better than 2 combined with a linear exponent depending on k in the time complexity, will be an open problem for a while. From a practical point of view, the dependence on k is one of the main decisive properties. The essential part of this practical problem can be stated as a theoretical question by asking for the smallest coefficient c in a running time $\mathcal{O}^*(2^{ck})n$ of an α -approximation. In particular, one wants to optimize c for $\alpha = 2$, the best known such approximation ratio.

Amir [17] has a 4.5-factor $\mathcal{O}(n^2)$ -time algorithm with 2^{3k} dependence on k . Even though the dependence on k is very good, the quadratic dependence on n makes it slow for larger inputs - which is why we do not consider it in the rest of this section. Belbasi-Fürer [21] have a 5-approximation $n \log n$ -time algorithm, where the dependence on k is $2^{8.766k}$. Later, they improved the dependence on k to $2^{6.755k}$ by using the notion of “leftmost separators” [?]. However, these two algorithms are still $n \log n$ -time algorithms. Korhonen’s most recent algorithm presents a linear-time algorithm in n , but worsens the dependence on k . However, his algorithm has an approximation ratio of 2 - which is remarkably low and makes the algorithm useful for solving hard graph problems on tree decompositions of small widths.

In this paper, we build on Korhonen’s idea and further improve the dependence on k of the running time to $\mathcal{O}^*(2^{8k})$. We present an algorithm that has the advantages of Korhonen’s algorithm [29] (i.e. a small approximation ratio and a time complexity which is linearly dependent on n), and Belbasi-Fürer’s algorithm [?] (i.e. a smaller coefficient of k in the exponent of the time complexity), while having the best known approximation ratio. In doing so, our algorithm is faster than all previous algorithms.

Before reviewing the complete history of the algorithms given for this problem, let us introduce our notations for some standard concepts.

Definition 4.1.1. A *tree decomposition* of a graph $G = (V, E)$ is a tree $\mathcal{T} = (V_{\mathcal{T}}, E_{\mathcal{T}})$ such that each node x in $V_{\mathcal{T}}$ is associated with a set B_x (called the bag of x) of vertices in G , and $\mathcal{T}$ has the following properties:

- $\bigcup_{x \in V_{\mathcal{T}}} B_x = V(G)$
- $\forall \{u, v\} \in E, \exists x \in V_{\mathcal{T}} : u, v \in B_x$
- $\forall x, y \in V_{\mathcal{T}}, \forall z \in V_{\mathcal{T}}$ on the path connecting $x, y \in V_{\mathcal{T}}: B_x \cap B_y \subseteq B_z$.

The *width* of a *tree decomposition* is the size of its largest bag minus one. The *treewidth* of a graph G is the minimum width over all tree decompositions of G , and is denoted by $\text{tw}(G)$.

Definition 4.1.2. A tree decomposition $\mathcal{T}$ is a *nice tree decomposition* if its nodes are of the following types:

- Leaf node; a node with no children.
- Introduce node; node x with only child y is an introduce node if $\exists v \in B_x$ such that $B_y = B_x \setminus \{v\}$.
- Forget node; node x with only child y is a forget node if $\exists v \in B_y$ such that $B_x = B_y \setminus \{v\}$.
- Join node; node x is a join node if it has exactly two children y_1 and y_2 and $B_{y_1} = B_{y_2} = B_x$.

Lemma 4.1.1 (Lemma 13.1.3 of [31]). Any tree decomposition $\mathcal{T} = (V_{\mathcal{T}}, E_{\mathcal{T}})$ of width k can be converted into a nice tree decomposition of width k^1 , in time $\mathcal{O}(|V_{\mathcal{T}}|)$.

Reference	Approximation Ratio	Dependence on k	Dependence on n
Robertson & Seymour [14]	4	3^{3k}	n^2
Lagergren [15]	8	$2^{\mathcal{O}(k \log k)}$	$n \log^2 n$
Reed [13]	7 or 5	$2^{24k} k!$	$n \log n$
Amir [17]	4.5	2^{3k}	n^2
Amir [17]	$\frac{11}{3}$	$2^{3.6982k}$	$n^3 \log^4 n$
Bodlaender et al. [19]	3	$2^{\mathcal{O}(k)}$	$n \log n$
Bodlaender et al. [19]	5	$2^{\mathcal{O}(k)}$	n
Belbasi-Fürer [21]	5	$2^{8.766k}$	$n \log n$
Belbasi-Fürer [22]	5	$2^{6.755k}$	$n \log n$
Korhonen [29]	2	$2^{10.755k}$	n
This chapter	2	2^{8k}	n

Table 4.1. An extended history of constant-factor FPT approximation of treewidth

Table 4.1 provides an extended history of the work done on constant-factor approximation of treewidth.

As we mentioned earlier, we focus only on the approximation algorithms with constant-factor approximation ratio. These can be used to solve many problems - including but not limited to Maximum Independent Set, Minimum Dominating Set, Hamiltonian Cycle, Traveling Salesperson, Longest Path, Connected Vertex Cover, Cycle Packing, Vertex Planarization, Feedback Vertex Set, Multiway Cut, Clique Cover, Max Leaf, Interval Completion, Steiner Tree, and Minimum k -Cut.

Furthermore, in Section ?? we discuss the bottleneck in solving some FPT graph problem given a graph G and $k \in \mathbb{N}$. Usually, FPT algorithms for solving NP-complete graph problems assume that a near-optimal tree decomposition is given as input. This is because often, finding the tree decomposition is a more computationally intensive task than solving the problem at hand.

We show that with our improvement to Korhonen’s algorithm, the bottleneck for some problems shifts from constructing the tree decomposition, to solving the problem on the constructed tree decomposition.

4.2 A Summary of Korhonen’s Algorithm

In this section we give a short summary of Korhonen’s algorithm. However, we leave out some of the details that can be found in [29]. In order to avoid repetition, we cover some of his ideas in the next section in a manner that fits our algorithm better.

¹with at most $4|V_{\mathcal{T}}|$ nodes

In the following, let $\text{tw}(G)$ be the treewidth of graph G . Korhonen starts with Bodlaender’s matching compression technique [16] and uses it as a blackbox. This technique relies on the following lemma.

Lemma 4.2.1 (Bodlaender’s Matching Compression Technique [16]). Given a graph $G = (V, E)$ and $k \in \mathbb{N}$, there is algorithm that in time $k^{\mathcal{O}(1)}n$ either

- determines that $\text{tw}(G) > k$,
- returns a matching in G of size $\leq \frac{|V|}{\mathcal{O}(k^6)}$, or
- returns a graph $G' = (V', E')$ with $|V'| = \mathcal{O}\left(n - \frac{n}{\mathcal{O}(k^6)}\right)$ such that if $\text{tw}(G) \leq k$, then $\text{tw}(G') \leq \text{tw}(G)$. Furthermore, any tree decomposition of G' of width $\leq k$ can be converted into a tree decomposition of G with width $\leq k$ in time $k^{\mathcal{O}(1)}n$.

Later on, Perković and Reed [32] improved Lemma 4.2.1 by replacing $\mathcal{O}(k^6)$ by $\mathcal{O}(k^2)$ in the denominator but Bodlaender’s version is sufficient for our purposes.

Lemma 4.2.1 can be used to prove the following lemma, which is a variant of a lemma given in [16]. The proof for the following proposition can be found in [16] and [29].

Lemma 4.2.2 (in [16] and [29]). Assume we are given graph $G = (V, E)$, $k \in \mathbb{N}$, and tree decomposition $\mathcal{T}$ of G of width $\leq 4k + 3$. If there exist an algorithm that in time $f(k)n$ either outputs a tree decomposition of G of width $\leq 2k + 1$ or determines that $\text{tw}(G) > k$, then there exist another algorithm that does the same job in time $f(k)k^{\mathcal{O}(1)}n$ without requiring a tree decomposition as a part of the input.

Korhonen’s algorithm starts with a tree decomposition of width at most $4(\text{tw}(G) + 1)$ and (if possible) splits a largest bag until no “large bag” remains. This way, he reduces either the treewidth or the number of maximum size bags in each iteration. He does this routine until no bag of size $\geq 2\text{tw}(G) + 3$ is found. Korhonen’s algorithm is inspired by a proof idea given by Bellenbaum and Diestel [30].

In the following, we use some of the notation and lemmas given in [29] and summarize the idea in order to prepare the reader for our improvement in Section 4.4.

Definition 4.2.1. A vertex set $W \subseteq V(G)$ is splittable if $V(G)$ can be partitioned into (C_1, C_2, C_3, S) such that:

- S separates C_i from C_j for $1 \leq i \neq j \leq 3$
- and $|W_i \cup S| < |W|$, where $W_i = W \cap C_i$ for $i = 1, 2, 3$.

We call such a 4-tuple a *split* of W .

Note that this definition immediately implies that at least two of the W_i 's and at least two of the C_i 's are non-empty.

Lemma 4.2.3 (Lemma 3.1 of [29]). Let $G = (V, E)$ be a graph with treewidth $\text{tw}(G)$. Then, every vertex set $W \subseteq V(G)$ with more than $2(\text{tw}(G) + 1)$ vertices is splittable.

Definition 4.2.2. For a given rooted tree decomposition $\mathcal{T}$ of graph $G = (V, E)$, the home of vertex $v \in V(G)$ is the closest node of $\mathcal{T}$ to the root of $\mathcal{T}$ that contains v in its bag. Let $h(v)$ denote the home node of v in $\mathcal{T}$.

Definition 4.2.3. For a given tree decomposition $\mathcal{T}$ with root r and root bag $B_r = W$ of a graph G , a split (C_1, C_2, C_3, S) of W is called a *minimum split* of W if

- S is a smallest separator among the separators of all splits of W , and
- S minimizes $\sum_{v \in S} d(h(v), r)$, where $d(x, y)$ is the distance between nodes x and $y \in \mathcal{T}$.

The *size of the split* (C_1, C_2, C_3, S) is $|S|$.

The main idea of this algorithm is to split a splittable bag until no splittable bag remains, and hence the resulting tree decomposition is a 2-approximation of the optimal tree decomposition of G since according to Lemma 4.2.3 any bag of size greater than $2\text{tw}(G) + 2$ is splittable.

4.2.1 Splitting a Splittable Bag

4.2.1.1 Finding a Minimum Split

Before splitting a bag, one has to find a minimum split. However, if we keep finding minimum splits from scratch, it increases the time complexity and defeats the whole purpose. For that reason, Korhonen uses dynamic programming to leverage the intermediate values. After we are done with splitting a bag W of node r , we can go find the next splittable bag W' of node r' and bring r' to the root.

Here we give a brief overview of how Korhonen updates the dynamic programming table. Let root r be the node that just obtained a minimal split. Let $p_1, \dots, p_\ell$ be the children of r and let the next splittable bag be W' of node r' . w.l.o.g. assume r' is a descendant of p_1 . In order to make the tree rooted at r' , we rotate the entire tree one step at a time until r' becomes root. After each rotation, we have to update the entries

of the dynamic programming table. Here is how one rotation is done: Rotate r to the right and bring p_1 to the root. i.e. we change the relationship between r and p_1 . Instead of r being parent of p_1 , now, p_1 becomes parent of r . (All the other relationships remain intact.) The updating of the dynamic programming table can be done by updating only the entries corresponding to p_1 and r . First, we have to “remove” the contribution of p_1 to the entry of r and then “add” the updated contribution of r to the entry of p_1 . We do not go into the details of this section since it has been described thoroughly in [29]. We just have to mention that Korhonen assigns a potential function to each editable bag to afford to pay for its rotation. The potential function for every bag B is $3^{|B|}$ in his method².

4.2.1.2 Performing the Split

In this section, we briefly discuss how Korhonen splits a splittable bag once a minimum split is found using dynamic programming.

Definition 4.2.4. A bag of tree decomposition $\mathcal{T}$ with root bag W is editable with respect to a minimum split (C_1, C_2, C_3, S) of W if it intersects at least two of C_i s and its parent is editable.

Lemma 4.2.4 (A variant of Lemma 3.3 of [29]). Given tree decomposition $\mathcal{T}$ with root bag W of graph G , and a minimum split (C_1, C_2, C_3, S) of W , C_i s can be re-arranged such that if a bag intersects l C_i s for $l \in \{2, 3\}$, then its parent definitely intersects at least l C_i s.

Let W be the root bag of $\mathcal{T}$ such that $|W| > 2(tw(G))$. Based on Lemma 4.2.3, W is splittable. Assume (C'_1, C'_2, C'_3, S) is its corresponding minimum split. Per Lemma 4.2.4 C'_i s can be re-arranged to obtain minimum split (C_1, C_2, C_3, S) of W satisfying the properties of Lemma 4.2.4. Now, make three copies of the editable part of $\mathcal{T}$, namely $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$. Then, hook every non-editable part to only its corresponding C_i . Add an auxiliary node r'' with bag S and make the roots of two of these copies a child of r'' . Then, add another auxiliary node with bag S such that the root of the third copy and r'' are its only children³. The result is a new tree decomposition $\mathcal{T}'$ with $|\mathcal{T}| + 2t + 2$ nodes, where t is the number of editable bags.

²Our algorithm’s most important advantage is that it obtains a polynomial potential function and this improves the entire running time.

³the reason to have r'' and r' at the same time is that we want to have a binary rooted tree

Let B_i be the corresponding bag in $\mathcal{T}_i$ to bag B of $\mathcal{T}$, for $i = 1, 2, 3$. From each bag B_i , remove the vertices that are not in $C_i \cup S$. Now, $\mathcal{T}'$ might not be a tree decomposition anymore since the subtree condition of the tree decomposition might be violated after the last step. The idea is to edit the bags of $\mathcal{T}'$ that are involved in this violation so that the result is a tree decomposition. The vertices whose absence causes this violation are all in S . Let B_i^S be the set of vertices of S that are in a descendent of B_i but not in B_i itself, i.e. $B_i^S = \{s \in S \setminus B_i \mid s \text{ is in a bag of a descendent of } B_i\}$. We can fix the violation by simply adding B_i^S (if it is non-empty) to B_i , i.e. $B_i \leftarrow B_i \cup B_i^S$. Notice that if $B_i^S = \emptyset$, then the bag B_i and all of its descendants are fine and there is no need to worry about them. Korhonen shows that $|B_i| < |B|$ for all $i = 1, 2, 3$ and for editable bag B . In particular, $|C_i \cup W_i| < |W|$.

Lemma 4.2.5 (Lemma 3.2. in [29]). Consider the construction that was described. For any bag B and any $i \in \{1, 2, 3\}$ such that $B_i^S \neq \emptyset$, then $|B_i^S| < |B \cap (C_j \cup C_k)|$, where $\{i, j, k\} = \{1, 2, 3\}$.

Corollary 4.2.1. For any $i \in \{1, 2, 3\}$, if B is editable, then $|B_i| < |B|$.

This means that not only do we not increase the width, but we also either decrease the width or decrease the number of maximum size bags. This gives us a 2-approximation algorithm.

Korhonen's algorithm always keeps the maximum degree of the tree decomposition ≤ 3 and replaces every editable bag with three copies. Hence, he has a potential function $\phi(x) = 3^{|B_x|}$ for every editable bag. He shows that the value of the potential function does not increase after every split and in fact decreases at least by the number of editable bags. This causes a factor of 3^k to show up in the running time since the algorithm terminates after spending the value of potential function time. We decrease this to only a polynomial factor, while preserving that the maximum degree of the tree decomposition ≤ 3 . Notice that each time that Korhonen makes three copies, most of the copied nodes actually do not carry valuable information since many adjacent bags are contained in each other. We prune these cases before making the copies to avoid 3^k in the running time. Like Korhonen, we use dynamic programming to find a minimum split, but we describe our version of the DP in our algorithm.

The details of the running time for Korhonen's algorithm can be found in [29]. We follow some of his approaches in the next section to analyze our running time. We also borrow some of the notations and definitions that he uses.

4.3 Dynamic programming

We begin by first describing our version of the dynamic programming, to find a minimum split.

For any subtree $\mathcal{T}_x$ rooted at node x of a tree decomposition $\mathcal{T}$, we define the graph G_x as the subgraph of G induced by the vertices in bags of $\mathcal{T}_x$. We assume, $V = \{1, 2, \dots, n\}$. Therefore, for each bag B_x of width ℓ_x , we have a canonical enumeration of its vertices. $B_x = \{v_0, v_1, \dots, v_{\ell_x}\}$, with $v_0 < v_1, \dots < v_{\ell_x}$.

The 4-partitions of B_x are encoded by $(c_0, c_1, \dots, c_{\ell_x}) \in \{1, 2, 3, 4\}^{|B_x|}$. We say that $(c_0, c_1, \dots, c_{\ell_x})$ defines the partition $(\tilde{C}_1^x, \tilde{C}_2^x, \tilde{C}_3^x, \tilde{C}_4^x)$ of B_x if $v_i \in \tilde{C}_{c_j}^x$ for $0 \leq i \leq \ell$. We use $\tilde{C}_j^x$ for parts or a partition of B_x and C_j for parts of partitions of the vertices of some G_x .

Two partitions $(C_1, \dots, C_4)$ and $(C'_1, \dots, C'_4)$ of subsets of V are *compatible* if $C_i \cap C'_j = \emptyset$ for $1 \leq i < j \leq 4$.

Our dynamic programming algorithm uses a large table A_x with $4^{|B_x|}$ entries for *every* node x . We call $I = (c_1, \dots, c_{\ell_x})$ the index of $A_x[c_1, \dots, c_{\ell_x}]$.

$$A_x[c_1, \dots, c_\ell] = (p, d, I_1, \dots, I_q) \text{ or } A_x[c_1, \dots, c_\ell] = \perp.$$

If $A_x[c_1, \dots, c_\ell] \neq \perp$, and the node x has q children $y_1, \dots, y_q$, then the following conditions are satisfied.

- (a) The number p is the size of a minimum split of the root bag B_x of G_x compatible with the partition $(\tilde{C}_1^x, \tilde{C}_2^x, \tilde{C}_3^x, \tilde{C}_4^x)$ of B_x defined by $(c_0, c_1, \dots, c_{\ell_x})$.
- (b) The number d is the sum of the distances of the homes of the vertices in S^x from the node x where $(C_1^x, C_2^x, C_3^x, S^x)$ is a minimum split of the root bag B_x of G_x compatible with $(\tilde{C}_1^x, \tilde{C}_2^x, \tilde{C}_3^x, \tilde{C}_4^x)$ for some sets C_1^x, C_2^x, C_3^x .
- (c) I_j is an index of y_j of x defining a partition $(\tilde{C}_1^{y_j}, \tilde{C}_2^{y_j}, \tilde{C}_3^{y_j}, \tilde{C}_4^{y_j})$ of B_{y_j} which is compatible with a minimum partition of B_x compatible with $(\tilde{C}_1^x, \tilde{C}_2^x, \tilde{C}_3^x, \tilde{C}_4^x)$.

We will show that if $A_x[c_1, \dots, c_\ell] \neq (\perp, \dots, \perp)$, then the table entries in $\mathcal{T}_x$ define a partition $(C_1^x, C_2^x, C_3^x, S^x)$ of the vertices of G_x compatible with the partition defined by $(c_1, \dots, c_\ell)$. $(C_1^x, C_2^x, C_3^x, S^x)$ is defined as follows.

- (a) If x is a leaf, then $(C_1^x, C_2^x, C_3^x, S^x)$ is the partition defined by the index $(c_0, c_1, \dots, c_{\ell_x})$.

- (b) Otherwise, x is a node with children $y_1, \dots, y_q$. Then, for $C_4 = S^x$, the partition $(C_1^x, C_2^x, C_3^x, S^x)$ is defined by

$$C_i^x = \tilde{C}_i^x \cup \bigcup_{j=1}^q C_i^{y_j} \text{ for } 1 \leq i \leq 4.$$

Lemma 4.3.1. Assume that the table entries $A_x[c_1, \dots, c_\ell]$ fulfill the properties (a) to (c) from above. Then the following statements hold.

- (a) The 4-tuple $(C_1^x, C_2^x, C_3^x, S^x)$ defined by the table entries in $\mathcal{T}_x$ is a partition of the vertices of G_x .
- (b) S^x separates C_i^x from C_j^x for $1 \leq i < j \leq 3$.
- (c) In $(C_1^x, C_2^x, C_3^x, S^x)$, the size of S_x is minimal among all partitions compatible with the partition $(\tilde{C}_1^x, \tilde{C}_2^x, \tilde{C}_3^x, \tilde{S}^x)$ of B_x which is defined by $(c_1, \dots, c_\ell)$, and $\sum_{v \in S_x} d(h(v), x)$ is minimal among those partitions with minimal $|S_x|$.

Proof. We prove this by induction on the height of x .

- (a) It is true for height 0 (leaves) by definition. Otherwise, by condition (c), the partitions $(\tilde{C}_1^{y_j}, \tilde{C}_2^{y_j}, \tilde{C}_3^{y_j}, \tilde{C}_4^{y_j})$ of B_{y_j} and $(\tilde{C}_1^x, \tilde{C}_2^x, \tilde{C}_3^x, \tilde{C}_4^x)$ are compatible with the same minimum partition of B_x , and therefore they are compatible with each other.
- (b) By the inductive hypothesis, S^{y_j} separates $C_i^{y_j}$ from $C_j^{y_j}$ for $1 \leq i < j \leq 3$, and $\tilde{S}^x = \tilde{C}_4^x$ separates $\tilde{C}_i^{y_j}$ from $\tilde{C}_j^{y_j}$.
- (c) Combining minimal solutions for the children y_j that agree with a common partition $(\tilde{C}_1^x, \tilde{C}_2^x, \tilde{C}_3^x, \tilde{S}^x)$ at the parent x , results in a minimal solution for x . □

Now, for our purpose, we define two new types of tree decompositions, namely *neighbor-reduced tree decomposition* and *grouped tree decomposition*.

Definition 4.3.1. We call a tree decomposition a *neighbor-reduced tree decomposition* if no bag is contained in the bags of any of its neighbors.

Lemma 4.3.2. Any tree decomposition $\mathcal{T}$ can be converted into a rooted neighbor-reduced tree decomposition in $\mathcal{O}(|V(\mathcal{T})|)$.

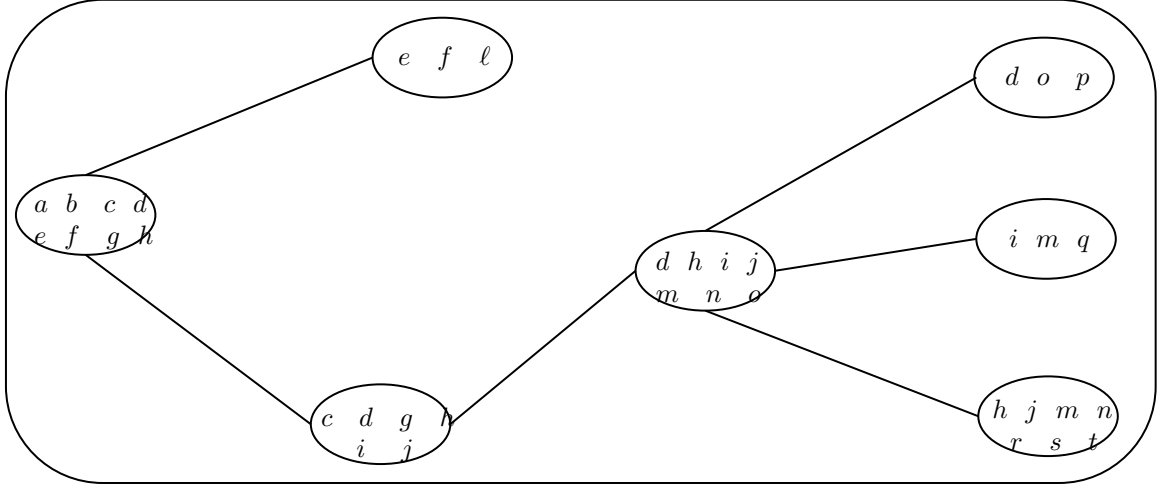


Figure 4.1. Neighbor-reduced tree decomposition before adding the intersection nodes and converting it into a grouped tree decomposition

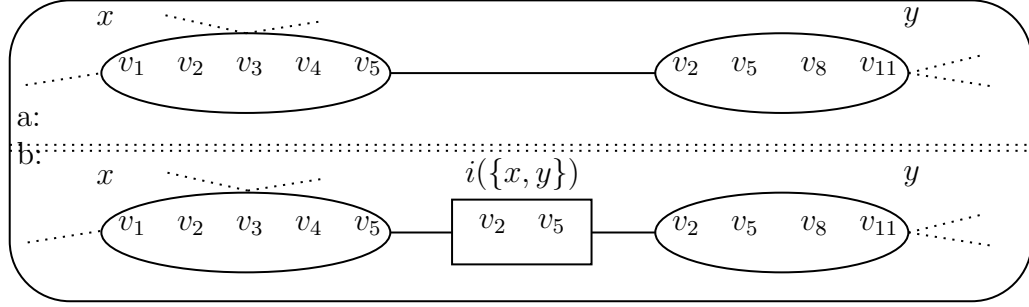


Figure 4.2. Intersection node

Proof. This can be done by a DFS on $\mathcal{T}$ starting from the root and checking the containment when post-visiting a node. We iterate over all $|V(\mathcal{T})|$ nodes in the tree, to check if the bags of any neighbors are contained within a node. Thus, the runtime is $\mathcal{O}(|V(\mathcal{T})|)$.⁴ $\square$

Now, we define a new type of tree nodes to help us define grouped tree decomposition.

Definition 4.3.2. Given a neighbor-reduced tree decomposition $\mathcal{T} = (V_{\mathcal{T}}, E_{\mathcal{T}})$, for any tree edge $\{x, y\} \in E_{\mathcal{T}}$, we define an intermediate intersection node $i(\{x, y\})$ with bag $B_{i(\{x, y\})} = B_x \cap B_y$.

We call the initial nodes of a neighbor-reduced tree decomposition $\mathcal{T}$, *original nodes* of $\mathcal{T}$ (see Fig. 4.2).

⁴There is an additional $O(k \cdot \log k)$ factor for checking containment that we are ignoring.

Definition 4.3.3. A *grouped tree decomposition* $\mathcal{T} = (V_{\mathcal{T}}, E_{\mathcal{T}})$ is a tree decomposition with the following properties in addition to the (ordinary) tree decomposition properties,

- $V_{\mathcal{T}}$ can be partitioned into two subsets $V_{\mathcal{T}}^O$ - the original nodes - and $V_{\mathcal{T}}^I$ - the intersection nodes - such that every edge of $\mathcal{T}$ has exactly one endpoint in both $V_{\mathcal{T}}^O$ and $V_{\mathcal{T}}^I$.
- No two neighbors of an original node have the same bags. i.e., $\forall p \in V_{\mathcal{T}}^O, \forall x, y \in N(p) : x \neq y \implies B_x \neq B_y$.
- Every intersection node $x \in V_{\mathcal{T}}^I$ with neighbors $p_1, \dots, p_c$ has the following properties:
 - $c \geq 2$.
 - For any distinct $i \in [c]$ and $j \in [c]$, $B_{p_i} \not\subseteq B_{p_j}$ and furthermore, $B_x = B_{p_i} \cap B_{p_j}$.

We write $\mathcal{T} = (\langle V_{\mathcal{T}}^O, V_{\mathcal{T}}^I \rangle, E_{\mathcal{T}})$. Furthermore, when drawing $\mathcal{T}$, we use ellipse shape for the original bags and rectangle shape for the intersection bags in order to make it more understandable.

In the following lemma, we introduce few properties of grouped tree decomposition.

Lemma 4.3.3. Let $\mathcal{T} = (\langle V_{\mathcal{T}}^O, V_{\mathcal{T}}^I \rangle, E_{\mathcal{T}})$ be a grouped tree decomposition. Then,

1. All the leaves are original nodes.
2. The bag of no original node is contained in the bag of any other original bag. (The local containment restriction over the original nodes implies the global containment restriction.)⁵
3. Every intersection node is properly contained in its neighbors. i.e., $\forall i \in V_{\mathcal{T}}^I, \forall p \in N(x) : B_i \subsetneq B_p$.
4. $\forall p \in V_{\mathcal{T}}^O, \deg(p) \leq 2^{|B_p|}$.

Proof. The proofs are straight-forward but they can be found in the conference version of this work that is going to be submitted soon. $\square$

Here, we show that converting a tree decomposition to a grouped tree decomposition can be done without increasing the width.

⁵Note that this does not hold for the intersection nodes. However, it is possible with some modifications.

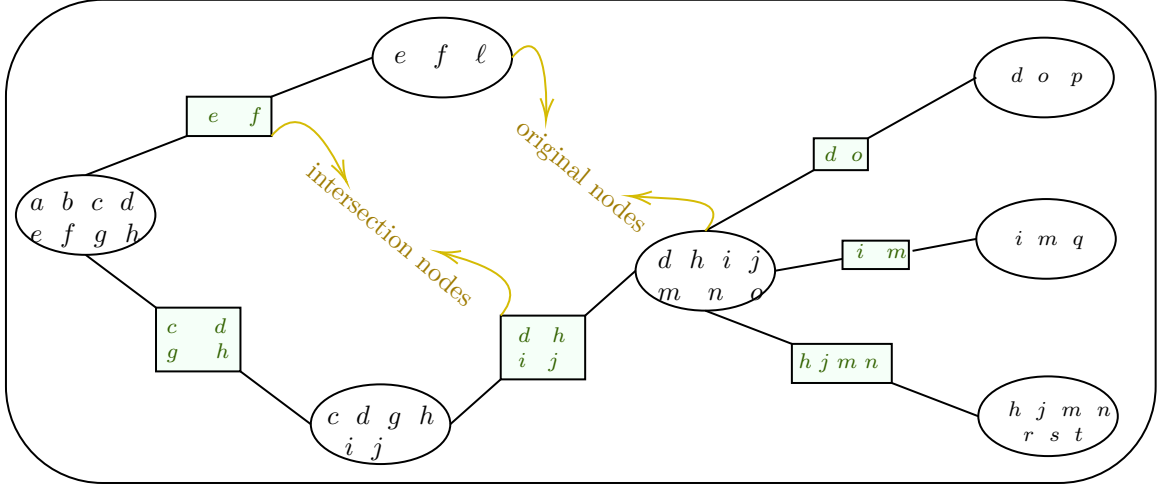


Figure 4.3. Grouped Tree Decomposition

Lemma 4.3.4. Any tree decomposition $\mathcal{T}' = (V_{\mathcal{T}'}, E_{\mathcal{T}'})$ of width k can be converted into an equivalent grouped tree decomposition $\mathcal{T} = (\langle V_{\mathcal{T}}^{\mathcal{O}}, V_{\mathcal{T}}^{\mathcal{I}} \rangle, E_{\mathcal{T}})$ with $\leq 2|V_{\mathcal{T}'}|$ nodes of the same width in time $\mathcal{O}(|V_{\mathcal{T}'}|)$.

Proof. The proofs are straight-forward but they can be found in the conference version of this work that is going to be submitted soon. $\square$

Definition 4.3.4. A minimum split (C_1, C_2, C_3, S) of W in rooted grouped tree decomposition $\mathcal{T} = (\langle V_{\mathcal{T}}^{\mathcal{O}}, V_{\mathcal{T}}^{\mathcal{I}} \rangle, E_{\mathcal{T}})$ rooted at node r with root bag W is called an *ideal minimum separator* of W if there exists no node $x \neq r$ with a child y such that y intersects l of C_i s and x intersect $\leq l - 1$ of C_i s, where $l \in \{2, 3\}$.

Definition 4.3.5. An *extended nice tree decomposition*, is a nice tree decomposition except that we allow a join node to have more than 2 children. Notice, still the bags of the children are identical to the bag of the parent in a join node. We call this type of node *extended-join node*, instead of join node.

Once the table A_r for the root has been computed, it is easy to find an ideal minimum split (as defined in Definition 4.3.4) of the root bag B_r . Pick any entry of the root table with minimum p and minimum d conditioned on p being a minimum. Then follow the indices in this entry in a depth-first manner. Whenever an index of a node x is reached with only one $\tilde{C}_i^x \neq \emptyset$ ($i \in \{1, 2, 3\}$), we stop on this branch, because for all descendants y of x , we would have $\tilde{C}_i^y = \emptyset$, i.e., y is not in the core (see Definition 4.4.1).

4.3.1 The Details of the Dynamic Programming Algorithm

For all nodes y with parent z in $\mathcal{T}$, we introduce a chain of intermediate nodes $y_1, \dots, y_q$ between y and z to obtain an extended nice tree decomposition $\hat{\mathcal{T}}$ (as defined in Definition 4.3.5). These nodes are useful for the dynamic programming algorithm. We store the same kind of tables in these nodes as in all the other nodes. For the potential function, however, we consider only the smaller tree $\mathcal{T}$ where each of these sequences of auxiliary nodes is replaced by just a tree edge.

For $n_{yz} = |B_y \setminus B_z|$, the sequence $y_1, \dots, y_{n_{yz}}$ consists of forget nodes, i.e., $|B_{i+1}| = |B_i| - 1$ and $B_{i+1} \subset B_i$. The remainder of the sequence $y_{n_{yz}+1}, \dots, y_q$ consists of introduce nodes, i.e., $|B_{i+1}| = |B_i| + 1$ and $B_i \subset B_{i+1}$. For $n_{zy} = |B_z \setminus B_y|$, we choose $q = n_{zy} + n_{yz}$ if z has more than 1 child, and $q = n_{zy} + n_{yz} - 1$ if z has a single child. Note that in the former case $B_{y_q} = B_z$, implying that all children of z have the same bag as z , i.e., z is an extended join node.

Now we have an extended nice tree decomposition $\hat{\mathcal{T}}$, and we can describe the dynamic programming algorithm in detail. As usual, it visits all nodes bottom-up. We describe what we does in every type of node.

Leaf

For a leaf x , and an index I defining a partition $(\tilde{C}_1^x, \tilde{C}_2^x, \tilde{C}_3^x, S^x)$, we define

$$A_x[I] = \begin{cases} (|\tilde{C}_4^x|, 0) & \text{if there are no edges between} \\ & \tilde{C}_i^x \text{ and } \tilde{C}_j^x \text{ in } G_x \text{ for } i \neq j, \\ \perp & \text{otherwise.} \end{cases}$$

Forget node

Let x with child y be a forget node, forgetting the vertex v . Let I be an index defining a partition $(\tilde{C}_1^x, \tilde{C}_2^x, \tilde{C}_3^x, \tilde{S}^x)$ of B_x , and let I'_j be obtained from I by adding a component for v with value $j \in \{1, 2, 3, 4\}$.

We define $A_x[I]$ with the help of the values p', d' and I' defined below.

$$A_x[I] = \begin{cases} \perp & \text{if } A_y[I'_j] = \perp \text{ for all } j \in \{1, 2, 3, 4\} \\ (p', d', I') & \text{otherwise,} \end{cases}$$

Let $J = \{j \in \{1, 2, 3, 4\} \mid A_y[I'_j] \neq \perp\}$. Assume $J \neq \emptyset$ and $A_y[I'_j] = (p'_j, d'_j, \dots)$. Here

and later, the entries represented by $\dots$ are a sequence of irrelevant indices, one for each child of y . We define $p = \min_{j \in J} p_j$. We will define j' be one of the values of j for which the minimum was obtained. If there is a tie, then we choose a j' with the additional properties $j' \in \{1, 2, 3\}$ and $\tilde{C}_{j'}^x \neq \emptyset$. In case of a tie, such a choice is always possible. Otherwise, it could be that $j' = 4$ (v goes into the separator) were the only option. This choice will ensure that we obtain an ideal minimum split. We define $d = d_j' + p - |\tilde{S}^x|$. Note that $p - |\tilde{S}^x|$ vertices of the separator S^x have their home below the root x of the subtree $\mathcal{T}$. As we move up from y to x , their distance from x increases by 1. Finally, I' is defined to be $I_{j'}$.

Introduce node

Let x with child y be an introduce node, introducing the vertex v . Let I be an index defining a partition $(\tilde{C}_1^x, \tilde{C}_2^x, \tilde{C}_3^x, \tilde{S}^x)$ of B_x , and let I' be obtained from I by removing the component for v . We define $A_x[I]$ with the help of $A_y[I'] = (p', d', \dots)$.

$$A_x[I] = \begin{cases} \perp & \text{if } A_y[I'] = \perp \text{ or } \exists i, j \in \{1, 2, 3\} \text{ with} \\ & i \neq j \text{ and } v \in \tilde{C}_i^x \text{ and } \exists u \in N(v) \cap \tilde{C}_j^x \\ (p', d' + p' - |\tilde{S}^x|, I') & \text{if } A_x[I] \neq \perp \text{ and } v \notin \tilde{S}^x \\ (p' + 1, d' + p' + 1 - |\tilde{S}^x|, I') & \text{if } A_x[I] \neq \perp \text{ and } v \in \tilde{S}^x \end{cases}$$

Extended join node

Let x be an extended join node with children $y_1, \dots, y_q$ with $B_x = B_{y_1}, \dots, B_{y_q}$. Let I be an index defining a partition $(\tilde{C}_1^x, \tilde{C}_2^x, \tilde{C}_3^x, \tilde{S}^x)$ of B_x . Let $A_{y_j} = (p_j, d_j, \dots)$. The definition of $A_x[I]$ is straightforward.

$$A_x[I] = \begin{cases} \perp & \text{if there is a } j \text{ with } A_{y_j}[I] = \perp \\ (p_1 + \dots + p_q - (q-1)|\tilde{S}^x|, d_1 + \dots + d_q, I, \dots, I) & \text{otherwise.} \end{cases}$$

Note that for p , we have to make sure, every element of $\tilde{S}^x$ is counted only once. For all elements of $\tilde{S}^x$ are counted q times. This does not matter, because there distance to the nodes y_j and x is 0.

4.4 Our Improved $\mathcal{O}^*(2^{8k})n$ -time 2-Approximation Algorithm

Korhonen [29] starts with any small-width tree decomposition with maximum degree of 3 and maintains this property, as mentioned in Section 4.2. Furthermore, in every split, he inserts $3t + 4$ nodes to the tree decomposition, where t is the number of editable bags as defined by Korhonen.

In our algorithm, we make sure in every step our tree decomposition is a grouped tree decomposition. This means that the original nodes have bounded degree (per Lemma 4.3.3) while the intersection nodes might have unbounded degree.

Lemma 4.4.1. The bag of every original node in a rooted grouped tree decomposition $\mathcal{T} = (\langle V_{\mathcal{T}}^{\mathcal{O}}, V_{\mathcal{T}}^{\mathcal{I}} \rangle, E_{\mathcal{T}})$ is the home bag of at least one vertex.

Proof. Since $\mathcal{T}$ is rooted, we can assign every non-root original node x a parent (intersection) node i with bag B_i .

Per Lemma 4.3.3, $|B_x \setminus B_i| \geq 1$. This implies that $\exists v \in B_x : v \notin B_i$. Hence, B_x is the home bag of v . $\square$

Lemma 4.4.2. Let graph $G = (V, E)$ and $W \subset V(G)$. In an ideal minimum separator (C_1, C_2, C_3, S) of W , no vertex $s \in S$ exists such that $\mathcal{N}(s)$ is contained within a single C_i .

Proof. Let us assume that such an $s \in S$ did exist, with $\mathcal{N}(s) \subseteq C_i$. Then, we could construct a *smaller* minimum split S' such that $s \in C_i$ and $|S'| = |S| - 1$.

However, since S is the minimum split, this cannot be. Thus, the lemma holds. $\square$

Definition 4.4.1. Given grouped tree decomposition $\mathcal{T} = (\langle V_{\mathcal{T}}^{\mathcal{O}}, V_{\mathcal{T}}^{\mathcal{I}} \rangle, E_{\mathcal{T}})$ rooted at node r with bag W and an ideal minimum separator (C_1, C_2, C_3, S) of W , node x of $\mathcal{T}$ is called *editable* if its bag B_x intersects at least two components C_i and C_j . Furthermore, the subgraph of $\mathcal{T}$ induced by the editable nodes is called *the editable part of the tree decomposition $\mathcal{T}$* or *the core* with respect to the ideal minimum separator (C_1, C_2, C_3, S) of the root bag W . This connected subgraph is denoted by $C(\mathcal{T})_{(C_1, C_2, C_3, S)}^W$. We refer to $|C(\mathcal{T})_{(C_1, C_2, C_3, S)}^W|$ as the *editable size* of $\mathcal{T}$ with respect to ideal minimum separator (C_1, C_2, C_3, S) of W (this is the same as the number of editable bags in Korhonen's notation).

In the following, we drop the superscript and the subscript of $C(\mathcal{T})_{(C_1, C_2, C_3, S)}^W$ while referring to the editable part of the tree decomposition.

Theorem 4.4.1. Given a graph $G = (V, E)$ and $k \in \mathbb{N}$, Algorithm 9 outputs either:

- a tree decomposition with width at most $2k + 1$, or
- indicates that $tw(G) > k$.

Proof. We use Lemma 4.2.1 to obtain a tree decomposition T of width $\leq 4k + 4$. Then, using Lemma 4.3.2 we convert it into a neighbor-reduced tree decomposition $\mathcal{T}$ rooted at node r with bag W , where W is the largest bag in $\mathcal{T}$.

We start with a grouped tree decomposition $\mathcal{T}$, rooted at node r with a maximum size bag W . As shown in Section 4.3, we find an ideal minimum separator (C_1, C_2, C_3, S) of W in time $\mathcal{O}(71^k)n$. However, the amortized running time is $\mathcal{O}(71^k)|C(\mathcal{T})|$ since we re-use the computed information for the non-editable part. Note that original-to-intersection rotation takes up to $\mathcal{O}(4^{|B_x|}) \leq 64^k$ time and intersection-to-original rotation takes up to $\mathcal{O}(|\mathcal{B}_\S|)$ time. So, overall, it takes $\mathcal{O}(64^k + 8^k)$, which is upper bounded by $\mathcal{O}(71^k)$.

Now, we use $\mathcal{T}$ to build a new tree decomposition $\mathcal{T}'$ with root r' such that $B_{r'} = S$. Using Algorithm 11 we create 3 copies of $\mathcal{T}$, namely $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$ with roots r_1 , r_2 , and r_3 , respectively. Then, make r_i s children of r' , for $i = 1, 2, 3$. We then iterate through all nodes in each $\mathcal{T}_i$ and ensure that every bag in $\mathcal{T}_i$ intersects only $C_i \cup S$, i.e. $\forall x_i \in V(\mathcal{T}_i)$, which corresponds to some $x \in \mathcal{T}$, $B_{x_i} = B_x \cap (C_i \cup S)$. While iterating, we ensure that we preserve the neighbor-reduced property as described in Lemma 4.3.2. We also ensure that if some element of the separator occurs in some $x_i \in \mathcal{T}_i$, it is added to every ancestor of x_i in $\mathcal{T}_i$.

Based on Corollary 4.2.1, we split W into bags that are strictly smaller than W . Finally, replace $\mathcal{T}$ with $\mathcal{T}'$. Then, find the next node z with maximum size bag W' . If $|W'| \geq 2k + 3$, we make $\mathcal{T}$ rooted at z , and recurs until the maximum size bag in the tree decomposition is of size at most $2k + 2$. Note that each time we recurse after the first split, we only make three copies of the editable part of the tree decomposition.

Notice that we prune the intermediate tree decomposition before making three copies of them by making them neighbor-reduced tree decompositions. Later, we show that this prevents exponential explosion by describing an optimal potential function.

Once we have the table recording the size of the minimum separator by the above dynamic programming algorithm given in Section 4.3, we continue to perform further splits on maximum size bags on the new three-partitioned tree decomposition, until the size of the largest bag is $\leq 2k + 2$.

Lemma 4.4.3. Given a rooted neighbor-reduced tree decomposition $\mathcal{T}$, with root r and at least one maximum size bag W , such that $|W| \geq 2k + 3$, every time we split W , either

the number of maximum size bags decreases by at least 1, or we decrease the width of $\mathcal{T}$ by at least 1.

Proof. Each split of W is a partitioning of G into three components C_1, C_2, C_3 and a separator S , where at least 2 of the C_i 's are non-empty. The root with bag W is then replaced by a new node n such that $B_n = S$ and the children of n are new nodes n_1, n_2, n_3 with $B_{n_i} = (W \cap C_i) \cup S$. By Lemma 4.2.3 we know that $|(W \cap C_i) \cup S| < |W|$ for all i . Thus the maximum size root bag is replaced by 4 smaller bags. Every other bag is replaced by 1 to 3 bags of smaller or equal size. If W is the only maximum size bag in $\mathcal{T}$, then the width of $\mathcal{T}$ decreases. Otherwise, since W is no longer a maximum size bag, the number of maximum size bags decreases by 1. $\square$

Now, we analyze the running of our algorithm.

Analysis using potential function: Every time we split some node q with bag W into smaller bags, we create *at most* 3 new nodes q_1, q_2, q_3 in place of q . If, for example, all the vertices with W as their home bag go into the same C_i , we create only one corresponding q_i and merge the others in the other sub-trees.

We define a potential function

$$\phi(T) = \sum_{q \in V(\mathcal{T})} w(q),$$

where $V(\mathcal{T})$ is the set of all nodes in T . For every non-root $q \in V(\mathcal{T})$ with parent p , we define

$$w(q) = \frac{|B_q| + |B_p|}{2} \cdot |B_q \setminus B_p|,$$

Lemma 4.4.4. Let $\mathcal{T}$ be a rooted neighbor-reduced tree decomposition. Given a node $q \in V(\mathcal{T})$, with the parent node $p \in V(\mathcal{T})$, the potential $w(q)$ in $\mathcal{T}$ is at least 1 more than the sum of the potentials of the replacement nodes q_i replicated in the trees $\mathcal{T}_i$ for $i \in I_q$ where $I_q = \{i \mid B_q \cap C_i \neq \emptyset\} \subseteq \{1, 2, 3\}$.

Proof. During the algorithm, the node q is replaced by a set of nodes q_i with $i \in I_q \subseteq \{1, 2, 3\}$. The node q_i exists if $B_q \cap C_i \neq \emptyset$ and $B_{q_i} \not\subseteq B_{p_i}$.

Let $S_q = S \cap (\bigcup_{q' \in V(\mathcal{T}_q)} B_{q'})$, where $\mathcal{T}_q$ is the subtree rooted at q . Then $B_{q_i} = (B_q \cap C_i) \cup S_q$ for some subset of S .

As in [29], we notice that $|B_{q_i}| < |B_q|$ because otherwise S_q could be replaced by $B \setminus C_i$ in S . The separator $S' = (S \setminus S_q) \cup (B \setminus C_i)$ would be of the same size as S , but S would not be minimal as S' is closer to the root.

Note that if p is the parent of q , then $B_q \setminus B_p = \bigcup_{i \in I_q} (B_{q_i} \setminus B_{p_i})$. This union is disjoint, implying $|B_q \setminus B_p| = \sum_{i \in I_q} |B_{q_i} \setminus B_{p_i}|$. Therefore the potential difference

$$\begin{aligned} w(q) - \sum_{i \in I_q} w(q_i) &= \\ &\geq |B_q \setminus B_p| \left(\frac{|B_q \setminus B_p|}{2} - \frac{\max |B_{p_i}| + \max |B_{q_i}|}{2} \right) \\ &\geq |B_q \setminus B_p| \end{aligned}$$

which is ≥ 1 for neighbor-reduced tree decompositions. This potential difference pays for the $\mathcal{O}(k)$ steps spent to create the nodes q_i from q and for the $\mathcal{O}(4^{3k})$ steps to do the dynamic programming steps in these nodes. $\square$

The following lemma allows to handle the change of root operation in an elegant way. It can be applied, if the dynamic programming algorithm has been done for all possible roots. This requires only twice the work, because for every edge, the same computation is done in both directions, instead of just upwards in the usual dynamic programming algorithm. With this preprocessing, a root change can be done in constant time. Alternatively, the procedure of Korhonen could be used.

Lemma 4.4.5. Let P be the path from some node p to another node q , and let P^r the reverse path from q to p in a tree decomposition. Then $\phi(P) - \phi(P^r) = |B_p|^2 - |B_q|^2$.

Proof. We prove this by induction on distance d (i.e. number of edges) between nodes p and q . Basis: For $d = 1$,

$$\begin{aligned} \phi((p, q)) - \phi((q, p)) &= (|B_p \setminus B_q| - |B_q \setminus B_p|)(|B_p| + |B_q|) \\ &= (|B_p \setminus (B_p \cap B_q)| - |B_q \setminus (B_p \cap B_q)|)(|B_p| + |B_q|) \\ &= (|B_p| - |B_q|)(|B_p| + |B_q|) \\ &= |B_p|^2 - |B_q|^2 \end{aligned}$$

Step: Let $\text{distance}(p, r) = d + 1$, $\text{distance}(p, q) = d$, and $\text{distance}(q, r) = 1$.

$$\begin{aligned} \phi((p, r)) - \phi((r, p)) &= \phi((p, q)) + (|B_q \setminus B_r|)(|B_q| + |B_r|) \\ &\quad - \phi((q, p)) + |B_r \setminus B_q|(|B_q| + |B_r|) \\ &= |B_p|^2 - |B_q|^2 - (|B_q| - |B_r|)(|B_q| + |B_r|) \\ &= |B_p|^2 - |B_r|^2 \end{aligned}$$

$\square$

Algorithm 9 TREEWIDTH APPROXIMATION(G, k)

Require: G is a graph with treewidth k

Ensure: Return either tree decomposition with width $\leq 2k + 1$ or state that $\text{tw}(G) > k$

```
1: Make  $\mathcal{T}$  a tree decomposition of  $G$  with treewidth  $4k$  (Lemma 4.2.1)
2: Make  $\mathcal{T}$  neighbor-reduced by running DFS and DFS_REDUCTION
3: while  $\exists$  node  $r \in V(\mathcal{T})$  with bag size  $\geq 2k + 3$  do
4:   Make  $\mathcal{T}$  rooted at  $r$ .
5:   Let  $W$  be the bag of  $r$ .  $\triangleright |W| \geq 2k + 3$ 
6:    $(C_1, C_2, C_3, S) \leftarrow \text{SPLIT}(W)$   $\triangleright$  Returns an ideal minimum separator of  $W$ 
7:   for  $i = 1, 2, 3$ , do
8:      $\mathcal{T}_i \leftarrow \text{REPLICATE\_REDUCED\_TREE}(\mathcal{T}, C_i, S)$ 
9:   end for
10:  Let  $r_i$  be the root of  $\mathcal{T}_i$ .
11:  Make  $r_i$ s the children of  $r'$ . Let  $\mathcal{T}'$  be this tree, which is rooted at  $r'$ .
12:  if  $|B_r| = |B_{r'}|$  then return  $\text{tw}(G) > k$ 
13:  end if
14:   $\mathcal{T} \leftarrow \mathcal{T}'$ 
15: end while
16: return  $\text{tw}(\mathcal{T})$ 
```

4.4.1 Runtime

Given a maximum size bag W , we can upper bound $|W| \leq 4k + 4$ - given Boedlander's algorithm. For $3k + 3 \leq |W| \leq 4k + 4$, we can split W into two parts of size at most $\frac{2}{3}|W|$ instead of three parts with size at most $\frac{1}{2}|W|$, by merging any two components. For $2k + 3 \leq |W| \leq 3k + 2$, we can split $|W|$ into three parts in time $\mathcal{O}(4^{3k}) = \mathcal{O}(2^{8k})$ - which is more time-intensive than the 2-way split, and thus dominates.

During the DP we create every 4-partition of the graph possible at every original node. This takes time $\mathcal{O}(2^{8k})n$ and makes the root-shifting operation a lookup operation. We then shift the root at most ϕ times, since we only need to split the nodes in the editable part of the tree decomposition, which is of size $\phi = \mathcal{O}(k^{o(1)})$. Note that rotating an intersection node takes only $\mathcal{O}(8^k)$ extra time.

Thus the total run-time of our algorithm is $\mathcal{O}((2^{8k} + 23k) \cdot k^{o(1)} \cdot n) = \mathcal{O}(71^k) n = \mathcal{O}(2^{8k}) n$.

□

Algorithm 10 SPLIT(W)

Require: $|W| > 2(tw(G))$ **Ensure:** The outcome is an ideal minimum separator of W Follow the approach given in Section 4.3

Algorithm 11 REPLICATE_REDUCED_TREE($\mathcal{T}, C_i, S$)

- 1: Let r and W be the root node and root bag of $\mathcal{T}$, respectively.
 - 2: Make a copy of $\mathcal{T}$, namely $\mathcal{T}_i$ with root node and root bag r_i and $W \cap (C_i \cup S)$, respectively.
 - 3: $Visited \leftarrow \emptyset$
 - 4: $(\mathcal{T}_i, S_{r_i}^S) \leftarrow \text{DFS_REDUCTION}(\mathcal{T}_i, r_i, C_i, S, Visited)$
 - 5: **return** $\mathcal{T}_i$
-

4.5 Conclusion

Any FPT approximation algorithm to find treewidth can be evaluated along three parameters: the dependence on n , the dependence on k , and the approximation ratio. Our algorithm draws from Korhonen's algorithm - which was already a 2-approximation optimal in n (linear) - and improves the dependence on k while maintaining the same approximation ratio and the dependence on n . Korhonen's algorithm has an exponential factor of $2^{10.755k}$ in the running time. We decrease this to a factor of 2^{8k} , which is the fastest across any constant ratio approximation faster than $O(n^2)$.

Algorithm 12 DFS_REDUCTION($\mathcal{T}_i, x, C_i, S, Visited$)

```

1:  $Visited.add(x)$ 
2:  $B_x \leftarrow B_x \cap (C_i \cup S)$ 
3:  $B_x^{S, \mathcal{T}_i} \leftarrow (B_x \cap S)$   $\triangleright$  The union of elements of  $S$  in the subtree  $\mathcal{T}_{i_x}$ 
4: for  $x' \in N(x)$  do  $\triangleright$  Start a DFS and check the containment as well as fixing the
   violations of the tree decomposition property
5:   if  $x' \notin Visited$  then
6:      $(\mathcal{T}_i, B_x^{S, \mathcal{T}_i}) \leftarrow \text{DFS\_REDUCTION}(\mathcal{T}_i, x', C_i, S, Visited)$ 
7:      $B_x^{S, \mathcal{T}_i} \leftarrow B_x^{S, \mathcal{T}_i} \cup B_{x'}^{S, \mathcal{T}_i}$ 
8:      $B_x \leftarrow B_x \cup B_{x'}$ 
9:     if  $B_x \subset B_{x'}$  then
10:      for  $y \in (N(x) \setminus \{x'\})$  do
11:        Make  $y$  a child of  $x'$ .
12:      end for
13:      Make  $x'$  a child of the parent of  $x$  and remove  $x$  from  $\mathcal{T}_i$ 
14:      Label  $x'$  with  $x$ .
15:     else if  $B_{x'} \subseteq B_x$  then
16:       for  $y \in N(x') \setminus x$  do
17:         Make  $y$  a child of  $x$  and remove  $x'$  from  $\mathcal{T}_i$ 
18:       end for
19:     end if
20:   end if
21: end for
22: return  $(\mathcal{T}_i, B_x^{S, \mathcal{T}_i})$ 

```

Chapter 5 |

Saving Space by Dynamic Algebraization Based on Tree Decomposition: Minimum Dominating Set

This chapter is based on a joint work with Martin Fürer [33].

5.1 Introduction

In this chapter, an algorithm is presented that solves the Minimum Dominating Set problem exactly using polynomial space based on dynamic programming for a tree decomposition. A direct application of dynamic programming based on a tree decomposition would result in an exponential space algorithm, but we use zeta transforms to obtain a polynomial space algorithm in exchange for a moderate increase of the time. This framework was pioneered by Lokshtanov and Nederlof 2010 and adapted to a dynamic setting by Fürer and Yu 2017. Our space-efficient algorithm is a parametrized algorithm based on tree-depth and treewidth. The naive algorithm for Minimum Dominating Set runs in $\mathcal{O}^*(2^n)$ time. Most of the previous works have focused on time complexity. But space optimization is a crucial aspect of algorithm design, since in several scenarios space is a more valuable resource than time.

Our parametrized algorithm runs in $\mathcal{O}^*(3^d)$, and its space complexity is $\mathcal{O}(nk)$, where d is the depth and k is the width of the given tree decomposition. We observe that Reed's 1992 algorithm constructing a tree decomposition of a graph uses only polynomial space.

So, even if the tree decomposition is not given, we still obtain an efficient polynomial space algorithm. There are some other algorithms which use polynomial space for this problem, but they are not efficient for graphs with small tree depth.

Improving the running time of exponential time algorithms for finding exact solutions of NP-complete problems like Minimum Dominating Set (MDS) have been studied for almost half a century. It seems that in the beginning they were emphasizing the running time but gradually some research started working on space optimization as well. In 1977, Tarjan and Trajnosky [34] improved the running time of Maximum Independent Set from $\mathcal{O}(2^n)$ to $\mathcal{O}(2^{\frac{n}{3}})$ using polynomial space. The Independent Set problem is strongly connected to Dominating Set since an independent set is also a minimal dominating set if and only if it is a maximal independent set. But we should mention that a dominating set is not necessarily an independent set. So we look for the results on the Maximum Independent Set problem as well, since it gives some ideas about the dominating set problem. Then in 1986, Jian improved Tarjan's algorithm and gave a better running time ($\mathcal{O}(2^{0.304n})$) again using polynomial space [35]. There is a trade-off between running time and space complexity. In the mentioned paper Jian states that there is a faster algorithm by Berman ($\mathcal{O}(2^{0.289n})$) but this time using exponential space (based on a private discussion between Jian and Berman).

Going back to the dominating set problem, in 2005, Grandoni [36] gave an algorithm which runs in $\mathcal{O}(1.9053^n)$ using polynomial space. Before this paper, the best algorithm for the dominating set problem was the trivial one which checks all possible subsets. He gives an algorithm with $\mathcal{O}(1.3803^k)$ running time for the minimum set cover problem where k is the dimension of the problem¹. Since the minimum dominating set problem can be converted to a minimum set cover problem by setting $k = 2n$, the running time of this algorithm for MDS is $\mathcal{O}(1.3803^{2n}) = \mathcal{O}(1.9053^n)$. Grandoni [36] gives an algorithm with better running time $\mathcal{O}(1.8021^n)$ for MDS. It is also based on converting the dominating set problem to a vertex cover problem but this time using exponential space. For MSC, the later algorithm runs in $\mathcal{O}(1.3424^k)$ and again by setting $k = 2n$ we have the algorithm for MDS with $\mathcal{O}(1.3424^{2n}) = \mathcal{O}(1.8021^n)$ time complexity. Grandoni uses a simple recursive algorithm for MSC and MDS. He uses dynamic programming which saves the answers for the partial problems. We present a framework to convert a dynamic programming algorithm to an algorithm which uses only polynomial space. Our algorithm do not save the results of the smaller problems and it computes them again

¹In the set cover problem, the dimension is $|S| + |U|$ where U is the universe and S is a collection of subsets of U . The goal is to find a minimum-size subset $S' \subseteq S$ that covers all elements in U .

whenever they are needed.

In 2005, Fomin, Grandoni and Kratsch [37] introduced an approach named “measure and conquer” based on the “branch and reduce” approach given by Davis and Putnam [38] in 1960. In the branch and reduce approach, there are some reduction rules which simplify the problem. Also, there are some branching approaches which are used when we cannot reduce the problem and we have to run it on some other branches. Measure and conquer is mostly used to measure sub-problems more accurately and giving tighter bounds. They chose Grandoni’s algorithm mentioned above (with running time $\mathcal{O}^*(2^{0.85n})$) and by choosing a better measuring method for sub-problems, they showed that actually its time complexity is $\mathcal{O}^*(2^{0.598n})$ using exponential space, and also $\mathcal{O}^*(2^{0.610})$ using polynomial space. This shows that in some cases, exponential recursive algorithms are not measured very well and by measuring them accurately we may gain better running time for the same algorithm. They gave a lower bound for the time complexity of this algorithm (not this problem) which is $\Omega(2^{0.333n})$.

Later, in 2008, Van Rooij and Bodlaender [39] gave two faster algorithms for the dominating set problem using polynomial space. The running time of these algorithms are $\mathcal{O}(1.5134^n)$ and $\mathcal{O}(1.5063^n)$ both using polynomial space. Like previous work by Grandoni [36], they took advantage of the minimum vertex cover problem and formulated MDS as a minimum vertex cover problem. In this paper, they modified the measure and conquer method in a way that it not only analyzes the running time more precisely but also modifies the algorithm by itself. For this purpose, they added a group of rules to the previous rules which tell when the algorithm should be improved.

In 2010, Lokshtanov and Nederlof [40] gave a framework turning a dynamic programming algorithm for several problems using exponential space to one which uses only polynomial space. They used DFT² and Zeta transform to simplify the complicated convolution operation and convert it to pointwise multiplication which is simpler. They gave conditions for turning the dynamic programming algorithms into algorithms with almost the same time complexity using much less space.

Then, in 2017 Fürer and Hu [41] combined this idea [40] with tree decompositions. They took Perfect Matching as a case study. Unlike [40], they handled dynamic underlying set. They introduced the zeta transforms for dynamic underlying sets. We take a similar approach to MDS.

²Discrete Fourier transform

5.1.1 Previous Work

We reviewed some of previous works on MDS in the previous section. Here we gather them in Table 5.1 and compare their running time and space complexity.

Author(s)	Year	Time Complexity	Space Complexity	Comments
Naïve Algorithm		$\mathcal{O}(2^n)$	polynomial	
Tarjan, and Trajonesky [34]	1977	$\mathcal{O}(2^{\frac{2}{3}n})$	polynomial	MIS
Grandoni [36]	2005	$\mathcal{O}(1.9053^n) \simeq \mathcal{O}(2^{0.9301n})$	polynomial	
Grandoni [36]	2005	$\mathcal{O}(1.8021^n) \simeq \mathcal{O}(2^{0.8497n})$	exponential	
Fomin et. al. [37]	2005	$\mathcal{O}(2^{0.610n})$	polynomial	
Van Rooji, and Bodlaender [39]	2008	$\mathcal{O}(1.5134^n) \simeq \mathcal{O}(2^{0.5978n})$	polynomial	
Van Rooji, and Bodlaender [39]	2008	$\mathcal{O}(1.5063^n) \simeq \mathcal{O}(2^{0.5911n})$	polynomial	
This chapter (Belbasi-Fürer [33])	2017	$\mathcal{O}(3^{dn^2})$	polynomial ($\mathcal{O}(nk)$)	d is tree-depth

Table 5.1. Previous work done on MDS, with a focus on time-space trade-off.

5.2 Notations

In this section we recall the problem and mention notations we use later. Some of the definitions in this section are based on the notation used in the “Parametrized Algorithms” [42] book.

Definition 5.2.1. Closed Neighbourhood of a Subset of Vertices Let X be a subset of vertices of a given graph $G = (V, E)$. Then the closed neighbourhood of X is defined as below:

$$N[X] = \bigcup_{v \in X} N[v],$$

where $N[v]$ is the closed neighbourhood of v .

Definition 5.2.2. Dominating Set: A subset of vertices D is a dominating set of a given graph $G = (V, E)$ if $N[D] = V$.

To simplify many application algorithms, the notion of a *nice tree decomposition* has been introduced which has the following properties. The tree is rooted and every node in a nice tree decomposition has at most two children. Any node x in a nice tree decomposition τ is of one of the following types (let c be the only child of x or let c_1 and c_2 be the two children of x):

- *Leaf* node, a leaf of τ .
- *Forget* node (forgetting vertex v), where $v \in B_c$ and $B_x = B_c \setminus \{v\}$,

- *Introduce vertex* node (introducing vertex v), where $v \notin B_c$ and $B_x = B_c \cup \{v\}$,
- *Join* node, where x has two children with the same bags as x , i.e. $B_x = B_{c_1} = B_{c_2}$.

It has been shown that any given tree decomposition can be converted to a nice tree decomposition with the same treewidth in polynomial time [31].

In 1987, Arnborg et al. showed that constructing a tree decomposition with the minimum treewidth is an NP-hard problem [6]. In our case we don't need the optimal tree decomposition, as a near optimal tree decomposition also works. A linear time algorithm for finding the minimum treewidth of a given graph with a bounded (by a constant) treewidth is given in [16]. Bodlaender et al. [43] gave an $\mathcal{O}(\log n)$ -approximation algorithm. The improved $\mathcal{O}(\log k)$ -approximation algorithm can be found in [17, 44, 45]. The result has been further improved to $\mathcal{O}(\sqrt{\log k})$ in [18]. It is well known that every tree decomposition can be modified in polynomial time into a nice tree decomposition with the same width and size $\mathcal{O}(n)$.

We use the notion of a *modified nice tree decomposition* which is defined in [41] with a small modification. Instead of introducing all edges in one node, we introduce them one by one. This helps us understand the procedure better. We also require leaf nodes to have empty bags.

We use $T(G)$ or simply T to denote a nice tree decomposition of a graph G .

For a node x in T , let T_x be the subtree rooted at x .

We define $\tau_x = (V_x, E_x)$ to be the graph with V_x being the union of all bags in T_x and E_x being the set of all edges introduced in T_x . With the traditional method of using the subgraphs induced by the already introduced vertices, i.e., by automatically including all edges, we would obtain an $O^*(k^d n)$ running time which is not desired.

We leverage the results of partial problems with different sizes and then solve the main problem according to the partial ones. Unlike previous works like [41], here for solving partial problems, we don't have only two sets. We maintain a partition of B_x into three sets:

- **“Dominating (Completed) set”**: In each partial problem, this set is the set of vertices which are in the dominating set. We denote this set by C (**Completed**).
- **“Dominated set”**: In each partial problem, this set is the set of vertices which are not contained in the dominating set C , and are dominated by vertices in C . We denote this set by D (**Dominated**).

- **“Waiting set”:** In each partial problem, this set is the set of vertices which are not contained in the dominating set C , and are not dominated by it (neither dominating nor dominated “yet”). We denote this set by W (**W**aiting to be dominated).

For each $x \in T$ (with the bag B_x), we define a function $f : B_x \rightarrow \{1, -1, 0\}$. This function tells for each vertex in B_x , in which of the sets C, D , or W it is. If $v \in B_x$, then:

- If $f(v) = 1$, it means that $v \in C$.
- If $f(v) = -1$, it means that $v \in D$.
- If $f(v) = 0$, it means that $v \in W$.

We follow a down-to-top approach. It means we start from the leaves and in each step, first we solve the problem for the child(ren) and then we use this computation for the parent node.

The underlying reason why we have the third option (set W) is that maybe some of the vertices of a bag which are in W would be dominated in upper nodes (in down-to-top approach). For a function f over B_x , let's define $c[x, f]$ to be minimum size of set $C \subseteq V_x$ such that:

- $D \cap B_x = f^{-1}(1)$.
- All vertices of $B_x \setminus f^{-1}(0)$, should be either in C or adjacent to one in C . It means, all vertices which don't have $f(v) = 0$, are dominated by some vertex or dominating some vertex (or vertices).

If there is no minimum set C associated with x , then we assign $c[x, f] = +\infty$. Our goal is to compute $c[r, \emptyset]$, where r is the root of our tree decomposition T . We start from the leaves and handle each node by handle its child (or children) first. Before introducing recursive formulas for each kind of nodes in T , let's define a specific version of f : Suppose $X \subseteq V(G)$, and $f : X \rightarrow \{1, -1, 0\}$, we define $f_{v \rightarrow p} : X \cup \{v\} \rightarrow \{1, -1, 0\}$ in this way:

$$f_{v \rightarrow p}(x) = \begin{cases} f(x), & \text{if } x \neq v \\ p & \text{if } x = v \end{cases} \quad (5.1)$$

Also, we use the restriction of f to some subset like $Y \subseteq X$ as follows: For a function f over X , we denote the restriction of f to Y by $f|_Y$.

Definition 5.2.3. For a universe $\mathcal{V}$ and a ring $\mathcal{R}$, the set $\mathcal{R}[2^\mathcal{V}]$ is the set of all function $f : 2^\mathcal{V} \rightarrow \mathcal{R}$.

Now we review the definition of the Zeta and Möbius Transforms.

Zeta Transform: The *zeta transform* of a function $f \in \mathcal{R}[2^\mathcal{V}]$ is defined by:

$$\zeta f[Y] = \sum_{X \subseteq Y} f[X]. \quad (5.2)$$

Möbius Transform/Inversion : The *Möbius transform* of a function $f \in \mathcal{R}[2^\mathcal{V}]$ is defined to be:

$$\mu f[Y] = \sum_{X \subseteq Y} (-1)^{Y \setminus X} f[X]. \quad (5.3)$$

We will see later that in the recursion formula for join nodes we need the union product [46].

Definition 5.2.4. Given $f, g \in \mathcal{R}[2^\mathcal{V}]$ and $X \in 2^\mathcal{V}$, the *Union Product* of f and g which is shown by $(f *_u g)$ is defined as:

$$(f *_u g)[X] = \sum_{X_1 \cup X_2 = X} f(X_1)g(X_2). \quad (5.4)$$

5.3 Saving Space Using Zeta and Möbius Transforms

In 2010, Lokshtanov and Nederlof [40] used the zeta and Möbius transforms to save space on some problems like the unweighted Steiner Tree problem. Then in 2017, Fürer and Yu [41] also used this approach in a dynamic setting, based on tree decompositions for the Perfect Matching problem.

Before going into the depth, let's review a few theorems.

Theorem 5.3.1. ([47, 48]) The Möbius transform is the inverse transform of the zeta transform, i.e. given $f \in \mathcal{R}[2^\mathcal{V}]$ and $X \in 2^\mathcal{V}$, $\mu(\zeta f)[X] = \zeta(\mu f)[X] = f[X]$.

Theorem 5.3.2. ([46]) Applying the zeta transform on a union product operation, will result in a simple point-wise multiplication of zeta transform outputs, i.e. given $f, g \in \mathcal{R}[2^\mathcal{V}]$ and $X \in 2^\mathcal{V}$,

$$\zeta(f *_u g)[X] = (\zeta f) \odot (\zeta g)[X], \quad (5.5)$$

where the operation $\odot$ is the point-wise multiplication.

All of the previous works which used either DFT or zeta transform (such as [41], and [40]), have one common central property that the recursive formula in join nodes, can be expressed as a formula using a union product operation. The union product and the subset convolution are complicated operations in comparison to point-wise multiplication or point-wise addition. That is why taking the zeta transform of a formula having union products makes computations easier. As noted earlier in Theorem 6.1.1, taking the zeta transform of such a term, will result in a term having point-wise multiplication which are much easier to handle than the union product. After doing a computation over the zeta transformed values, we can apply the Möbius transform on the outcome to get the main result (based on Theorem 5.3.1). In other words, the direct computation has a mirror image using the zeta transforms of the intermediate values instead of the original ones. While the direct computation keeps track of exponentially many intermediate values, the computation over the zeta transformed values partitions into exponentially many branches and they can be executed one after another. We show later that this approach only moderately affects the exponential factor in the time complexity, while improving the space complexity using only polynomial space instead of exponential space.

5.3.1 Counting the dominating sets of all sizes

For each node (of any kind) we introduce a polynomial to compute the number of the dominating sets of all sizes as follows:

$$P_x^C[D] = \sum_j a_{x,j}^C[D] y^j, \quad (5.6)$$

where $a_{x,j}^C[D]$ is the number of the dominating sets C^* of size j in τ_x with $C^* \cap B_x = C$, and where D is the set of all dominated vertices (excluding vertices in C) in B_x . We apply the zeta transform on the coefficients of this polynomial.

Now, we show how to compute $a_{x,j}^C[D]$.

- **Leaf node.** Assume x is a leaf node, then $B_x = \emptyset$. So, $a_{x,j}^\emptyset[\emptyset] = 0$ for all $j \neq 0$ since we do not have j vertices in the bag and $a_{x,0}^\emptyset[\emptyset] = 1$ since there is an empty dominating set of size 0. This implies that:

$$(\zeta a_{x,j}^\emptyset)[\emptyset] = \begin{cases} 0 & \text{if } j \neq 0, \\ 1 & \text{if } j = 0. \end{cases} \quad (5.7)$$

- **Introduce vertex node.** Assume x is an introduce vertex node and let x' be the child of x such that $\exists v \in B_x, v \notin B_{x'} : B_x = B_{x'} \cup \{v\}$. Then, if $v \notin D$, dominating sets of any size would be the same for both subtrees (τ_x and $\tau_{x'}$). Otherwise ($v \in D$), there would be no dominating set through τ_x to dominate D since v is isolated through τ_x which means $a_{x,j}^C[D] = 0$.

$$a_{x,j}^C[D] = \begin{cases} a_{x',j}^{C \setminus \{v\}}[D] & \text{if } v \notin D, \\ 0 & \text{if } v \in D. \end{cases} \quad (5.8)$$

Now by applying the zeta transform on the equation above, if $v \in D$, then $(\zeta a_{x,j}^C)[D] = 0$ and otherwise:

$$(\zeta a_{x,j}^C)[D] = \sum_{Y \subseteq D} a_{x,j}^C[Y] = \sum_{Y \subseteq D} a_{x',j}^{C \setminus \{v\}}[Y] = (\zeta a_{x',j}^{C \setminus \{v\}})[D]. \quad (5.9)$$

- **Forget node.** Assume x is a forget node and let x' be the child of x such that $\exists v \in B_{x'} : B_x = B_{x'} \setminus \{v\}$. Then, v should be either in D or C , so:

$$a_{x,j}^C[D] = a_{x',j}^C[D \cup \{v\}] + a_{x',j}^{C \cup \{v\}}[D]. \quad (5.10)$$

Now, by applying zeta transform we have:

$$\begin{aligned} (\zeta a_{x,j}^C)[D] &= \sum_{Y \subseteq D} a_{x,j}^C[Y] = \sum_{Y \subseteq D} (a_{x',j}^C[Y \cup \{v\}] + a_{x',j}^{C \cup \{v\}}[Y]) = \\ &= ((\zeta a_{x',j}^C)[D \cup \{v\}] - (\zeta a_{x',j}^C)[D]) + ((\zeta a_{x',j}^{C \cup \{v\}})[D]). \end{aligned} \quad (5.11)$$

- **Join node.** Assume that x is a join node and let x' and x'' be the children of x and also we know that $B_x = B_{x'} = B_{x''}$.

Then, if a vertex $v \in B_x$ is in dominating set (C), then it should be in dominating set in $B_{x'}$ and $B_{x''}$ as well and vice versa ($C = C' = C''$, where C' and C'' are the dominating sets through $\tau_{x'}$ and $\tau_{x''}$ respectively intersecting $B_{x'}$ and $B_{x''}$). By the way, if a vertex is dominated through $\tau_{x'}$ or $\tau_{x''}$ (or both), then it is dominated through τ_x . So, $D = D' \cup D''$, where D' (and D'') is the dominated set through $\tau_{x'}$ (and $\tau_{x''}$) intersecting $B_{x'}$ (and $B_{x''}$).

In order to compute $a_{x,j}^C$ based on pre-computed $a_{x',k}^{C'}$ and $a_{x'',l}^{C''}$ values, we have the

recursion below:

$$a_{x,j}^C[D] = \sum_{\substack{j' > 1, j'' > 1 \\ j' + j'' = j + |C|}} \sum_{D' \cup D'' = D} a_{x',j'}^C[D'] \cdot a_{x'',j''}^C[D''] = \sum_{\substack{j' > 1, j'' > 1 \\ j' + j'' = j + |C|}} (a_{x',j'}^C *_{\mathbf{u}} a_{x'',j''}^C)[D]. \quad (5.12)$$

By applying zeta transform, we will have the equation below:

$$\zeta(a_{x,j}^C)[D] = \sum_{\substack{j' > 1, j'' > 1 \\ j' + j'' = j + |C|}} (\zeta a_{x',j'}^C)[D] \cdot (\zeta a_{x'',j''}^C)[D] \quad (5.13)$$

- **Introduce edge node.** Assume that x is an introduce edge node introducing edge $\{u, v\}$.

Definition 5.3.1. An *auxiliary leaf node* x , is a leaf node which has some vertices in its bag (at least 2) and is used to introduce an edge (regularly leaf nodes have empty bag). It has only one introduced edge through B_x . We use auxiliary leaf nodes to introduce an edge. This helps us to handle introduce edge nodes.

We convert the introduce edge node to a join node such that x has two children l and x' where $B_x = B_l = B_{x'}$ but edges are different. In auxiliary leaf node l , we will have only $\{u, v\}$ edge introduced and other vertices are isolated through τ_l which has only one node (l itself). On the other hand, all other edges are present in x' except $\{u, v\}$. We saw how to handle join nodes, so we can handle introduce edge nodes as well. Remember, this branching does not affect our running time since branching is not balanced and in such a branch we have only one auxiliary leaf node which can be done very easily and has no children. Now, let's see how to handle auxiliary leaf node.

- **Auxiliary leaf node.** Assume x is an auxiliary leaf node with only $\{u, v\}$ edge, then we have four cases. In all cases, $a_{x,j}^C[D] = 0$ if $j \neq |C|$, so we do not consider this condition and assume that $j = |C|$ in all cases:

- If $D = \emptyset$, then,

$$a_{x,j}^C[\emptyset] = 1 \quad (5.14)$$

Any given set C , dominates an empty set D . As a reminder, if $j \neq |C|$, $a_{x,j}^C[D]$ is zero and we do not repeat this case from so on and only consider the cases where $j = |C|$.

- If $D = \{u, v\}$, then $a_{x,j}^C[D] = 0$ since dominating u forces v to be in the dominating set and on the other hand, dominating v forces u to be in the dominating set. So it means that $D = \{u, v\} \subseteq C$ but we know that D and C are disjoint sets so it is impossible. Therefore, there is no dominating set C to dominate D .
- If D is either $\{u\}$ or $\{v\}$, then,

$$a_{x,j}^C[D] = \begin{cases} 1 & \text{if } \{u, v\} \setminus D \subseteq C, \\ 0 & \text{otherwise.} \end{cases} \quad (5.15)$$

Without loss of generality, assume that $D = \{u\}$, then C can be any subset of B_x including v (the only way to dominate u) and excluding u (D and C are disjoint sets).

- And finally if $D \setminus \{u, v\} \neq \emptyset$, then we cannot dominate elements of $D \setminus \{u, v\}$ because they are isolated and they themselves cannot be in C (remember we assumed C and D are distinct and if they are not distinct, then there no let $a_{x,j}^C[D]$ to be zero). So, $a_{x,j}^C[D] = 0$.

Now, let us compute the zeta transform of the above cases:

- In case 1:

$$(\zeta a_{x,j}^C)[D] = (\zeta a_{x,j}^C)[\emptyset] = a_{x,j}^C[\emptyset] = 1 \quad (5.16)$$

- In case 2:

$$\begin{aligned} (\zeta a_{x,j}^C)[D] &= a_{x,j}^C[D] + a_{x,j}^C[\{u\}] + a_{x,j}^C[\{v\}] + a_{x,j}^C[\emptyset] \\ &= \begin{cases} 0 + 1 + 0 + 1 = 2 & \text{if } C = \{v\}, \\ 0 + 0 + 1 + 1 = 2 & \text{if } C = \{u\}, \\ 0 + 0 + 0 + 1 = 1 & \text{otherwise.} \end{cases} \end{aligned} \quad (5.17)$$

- In case 3:

$$\begin{aligned} (\zeta a_{x,j}^C)[D] &= a_{x,j}^C[D] + a_{x,j}^C[\emptyset] \\ &= \begin{cases} 1 + 1 = 2 & \text{if } \{u, v\} \setminus D \subseteq C, \\ 0 + 1 = 1 & \text{otherwise.} \end{cases} \end{aligned} \quad (5.18)$$

– In case 4:

$$(\zeta a_{x,j}^C)[D] = \left(\sum_{Y \subseteq D: Y \setminus \{u,v\} \neq \emptyset} a_{x,j}^C[Y] \right) + (\zeta a_{x,j}^C)[D \cap \{u,v\}] = (\zeta a_{x,j}^C)[D \cap \{u,v\}]. \quad (5.19)$$

which is one of the above cases.

By looking at Eq. 5.13, we see that we have n -fold branching in join nodes which is not desired. When computing an $a_{x,j}^C[D]$ for a fixed C and D , we always want to compute the whole vector for all j and store it. So for all fixed C and D , we store an array $Res[1..n]$ of size n , where $Res[j] = a_{x,j}^C[D]$. Now, even though we have n branches but we do not need to recompute them each time. Instead, we can compute them once and store them in array of size n which is linear. We have only one copy of this array because we handle any fixed C and D one after another.

5.3.2 Finding The Minimum Dominating Set

In this section we are going to find the minimum dominating set for a given graph $G = (V, E)$.

As mentioned before, assume we have a modified nice tree decomposition T with the root r where $B_r = \emptyset$. Our goal is to compute

$$\operatorname{argmin}_{j=0}^n \{a_{r,j}^\emptyset[\emptyset] > 0\},$$

where $n = |V(G)|$.

We start from $j = 0$ and increment j while $a_{r,j}^\emptyset[\emptyset] = 0$.

Algorithm 13 Minimum dominating sets of all sizes on a modified nice tree decomposition

Require: a modified nice tree decomposition τ with root r of a graph $G = (V, E)$.

Ensure: Size of a minimum dominating set of G . $\zeta a(x, D, C, j)$ represents $(\zeta a_{x,j}^C)[D]$. *Comment* D : a subset of $B_x \setminus C$, and $0 \leq j \leq n = |V(G)|$

1: $j \leftarrow 0$

2: initialize $Res[1..n] \leftarrow (\overbrace{0, \dots, 0}^n) \triangleright f(x, D, C, Res, j)$ is computed by Algorithm 14.

3: **while** $f(r, \emptyset, \emptyset, Res, j) = 0$ **do**

4: $j \leftarrow j + 1$

5: **end while** **return** j

Theorem 5.3.3. The Algorithm 13, outputs the size of the minimum dominating set correctly.

Proof. First of all, Algorithm 13 starts from $j = 0$ and checks if there a minimum dominating set to dominate everything in the tree by increasing j by 1 step by step. In the while loop, it calls Algorithm 14, to compute $f(x, D, C, j)$, where it is an alias $(\zeta_{x,j}^C)[D]$.

The while loops will end as soon as the Algorithm 13 finds the first j to make $f(x, D, C, j)$ non-zero. So as we explained before, $f(x, D, C, j)$ (or $(\zeta_{x,j}^C)[D]$), it is a function to compute the number of possible dominating sets of size j in the sub-tree rooted at x . Here $x = r$ and the sub-tree is our original tree. So if $j > 0$, then it means we have a dominating set for the whole tree of size j and the Algorithm 13 will return it.

Now, let's see how Algorithm 14 computes $f(x, D, C, j)$.

This algorithm gets a subtree rooted at x , a set $D \subseteq B_x$ that we want to dominate, a subset C to be the intersection of the whole dominating set in τ_x (which dominated D) named C^* with the bag of x .

As we explained before, keeping track of the number of such C^* s of size j will lead us a complicated convolution operation in join nodes. Therefor, we will keep track of the zeta transform of all computation instead on the original ones. This will save space for us. Assume the tree decomposition is given, we will make a copy of it to save $(\zeta a_{x,j}^C)[D]$ instead of $a_{x,j}^C[D]$. The procedure of computing zeta transforms $f(x, D, C, j)$, is exactly as shown in previous section.

In this step, the algorithm checks the type of x . Here we explain the cases:

- x is a **leaf** node: Then the only possible D is an empty set and $C^* = \emptyset$ will dominate it. So there is one way to dominate it and $f(x, D, C, j) = 1$.
- x is an **introduce vertex** node: Suppose x' is the only child of x and $B_x = B_{x'} \cup \{v\}$. As we showed previously $(\zeta a_{x,j}^C)[D] = (\zeta a_{x',j}^{C \setminus \{v\}})[D]$ if $v \notin D$. Otherwise it would be zero.
- x is a **forget** node: Suppose x' is the only child of x and $B_x \cup \{v\} = B_{x'}$. As shown before, $((\zeta a_{x',j}^C)[D \cup \{v\}] - (\zeta a_{x',j}^C)[D]) + ((\zeta a_{x',j}^{C \cup \{v\}})[D])$.
- x is a **join** node: Suppose x' and x'' are the children of x and $B_x = B_{x'} = B_{x''}$. As shown before, $(\zeta a_{x,j}^C)[D] = \sum_{p=0}^j (\zeta a_{x',p}^C)(\zeta a_{x'',j-p-|C|}^C)[D]$.
- x is a **auxiliary leaf** node: There are four cases (In all cases if $j \neq |C|$, then $a_{x,j}^C[D] = 0$):

Algorithm 14 Computing $f(x, D, C, j)$ (which actually is $(\zeta a_{x,j}^C)[D]$).

Require: a sub-tree of a modified nice tree decomposition τ_x rooted at x , a set $C \subseteq B_x$ to be the intersection of dominating set of D through τ_x with B_x , the vector of all $a_{x,j}^C[D]$ for any fixed C and D , and size of the dominating set C to dominate D through τ_x .

Ensure: $f(x, D, C, Res, j)$. $\triangleright f(x, D, C, Res, j)$ represents $(\zeta a_{x,j}^C)[D]$.

```

1: if  $j \neq |C|$  then
2:    $Res[j] \leftarrow 0$ 
3:   return 0
4: end if
5: if  $x$  is a leaf node then
6:   return 1
7: end if
8: if  $x$  is an introduce vertex node  $\triangleright$  introducing vertex  $v$  then
9:   if  $v \in D$  then return 0
10:  end if
11:  set  $Res[1..n] = [0..0]$ 
12:  return  $f(x', D, C \setminus \{v\}, Res, j)$   $\triangleright x'$  is the child of  $x$ 
13: end if
14: if  $x$  is a forget node  $\triangleright$  forgetting vertex  $v$  then return
     $f(x', D \cup \{v\}, C, j) - f(x', D, C, Res, j) + f(x', D, C \cup \{v\}, Res, j)$   $\triangleright x'$  is the child of  $x$ .
15: end if
16: if  $x$  is a join node then
17:   return  $\sum_{\substack{j', j'' > 1 \\ j' + j'' = j - |C|}} f(x', D, C, j') f(x'', D, C, j'')$   $\triangleright x'$  and  $x''$  are the children of  $x$ .
18: end if
19: if  $x$  is an auxiliary leaf node  $\triangleright$  having  $\{u, v\}$  as its only edge then
20:   if  $D = \emptyset$  then
21:     if  $j = |C|$  then return 1
22:     end if
23:     return 0
24:   end if
25:   if  $D = \{u, v\}$  then return 0
26:   end if
27:   if  $D = \{u\}$  or  $D = \{v\}$  then
28:     if  $j = |C|$  and  $\{u, v\} \setminus D = C$  then return 2
29:     end if
30:     if  $j = |C|$  and  $\{u, v\} \setminus D \neq C$  then return 1
31:     end if
32:     return 0
33:   end if
34:   return  $f(x, D \cap \{u, v\}, C, j)$ 
35: end if

```

– Case 1: $D = \emptyset$:

$$(\zeta a_{x,j}^C)[D] = 1. \quad (5.20)$$

– Case 2: $D = \{u, v\}$:

$$(\zeta a_{x,j}^C)[D] = \begin{cases} 2 & \text{if } C = \{v\} \text{ or } \{u\}, \\ 1 & \text{Otherwise.} \end{cases} \quad (5.21)$$

– Case 3: $D = \{u\}$ or $\{v\}$:

$$(\zeta a_{x,j}^C)[D] = \begin{cases} 2 & \text{if } \{u, v\} \setminus D \subseteq C, \\ 1 & \text{Otherwise.} \end{cases} \quad (5.22)$$

– Case 4: $D \setminus \{u, v\} \neq \emptyset$:

$$(\zeta a_{x,j}^C)[D] = (\zeta a_{x,j}^C)[D \cap \{u, v\}]. \quad (5.23)$$

Starting from leaf nodes, we compute only one strand at a time and continue the path to the root and we showed how parents can be computed based on their child(ren). $\square$

5.4 Analyzing Time and Space Complexity of the Algorithm

In the previous sections, we talked about our algorithm. Now, it is the time to explain the time and the space complexity of the algorithm given above. As we explained beforehand, we have an exact algorithm using only polynomial space.

Theorem 5.4.1. Given a tree decomposition τ of a graph G , the Algorithm 13, outputs the size of minimum dominating set in $\mathcal{O}(n^2 3^d)$ time using $\mathcal{O}(nk)$ space, where k is the treewidth and d is the tree-depth.

Proof. We know that the size of any dominating set of a Graph $G = (V, E)$ is at most $|V| = n$. So, the while loop in Algorithm 13 runs at most n times (j is the size of minimum dominating set). In each iteration, we call function f recursively. This function is computed by Algorithm 14. So, we need to look at the running time of the Algorithm 14: In each node of the tree decomposition, we have at most three branches and it happens

at forget nodes. On the other hand, we have $\mathcal{O}(n)$ forget nodes in the tree. Thus, the total running time of Algorithm 14 is $\mathcal{O}(n3^d)$. So finally, the total running time of Algorithm 13 is $\mathcal{O}(n^23^d) = \mathcal{O}^*(3^d)$.

Space complexity: We do not store the intermediate values and only compute one strand at a time, thus the space complexity is $\mathcal{O}(nk)$. $\square$

5.5 Extension

So far, we saw how to find a minimum dominating set when the tree decomposition is given. Now, we describe how to find the size of a minimum dominating set using polynomial space when the graph itself is given.

Remark 5.5.1. ([13]) In 1992, B. Reed gave an approximated vertex separator algorithm that gives another algorithm determining whether a given graph G has a tree decomposition of width at most k . Then it finds such a tree decomposition if it exists. The running time of this algorithm is $\mathcal{O}(n \log n)$ for any fixed k using polynomial space.

It is not mentioned that the space complexity of the algorithm is polynomial but this algorithm saves only the results for each strand and has a polynomial space complexity.

So, even if the tree decomposition is not given, still we can use Reed's algorithm to find a tree decomposition with width of k . We can start from $k = n$ and set k to $\frac{k}{2}$ (binary search) if this algorithm says there is a tree decomposition of width k . At some point, the answer would be no and then the algorithm will output the tree decomposition of size k . Now, we can use the output to compute the size of minimum dominating set of the original graph by Algorithm 13.

We also should mention that, the same framework can be applied on the Maximum Independent Set (MIS) problem to covert a dynamic programming algorithm using exponential time to a parametrized algorithm taking tree depth as its parameter using only polynomial space. On the other hand, if we look closer to the Maximum Independent Set problem and write down the recursions, we see that also by applying zeta transform we save space, but we sacrifice huge amount of time. But the good point about MIS is that, if we solve it using recursion on tree decomposition, it has already polynomial space complexity and we do not need to apply zeta transform.

One good problem would be to see if this framework works on the Hamiltonian Cycle problem. This one is much complicated because the nature of the problem is different. In previous problems, only pairs of vertices and edges (individually) were important but here one should look for all of possible disjoint paths.

In general, any problem of graphs (even if the problem is not on graph, (maybe) we can convert it) which has convolution in the join nodes, can be a candidate for our framework.

5.6 Conclusion

We applied the dynamic algebraization approach ([41]) to the Minimum Dominating Set problem to give a space-efficient dynamic programming algorithm using polynomial space. Our algorithm runs in time $\mathcal{O}(n^2 3^d)$ using $\mathcal{O}(nk)$ space where d and k are depth and width of the tree decomposition respectively. Even if the tree decomposition is not given, as mentioned in Remark 5.5.1, we can construct a tree decomposition with sufficiently small treewidth by using polynomial space and then apply our algorithm. Again, the space complexity is polynomial, and for some function f , the running time is $\mathcal{O}(f(k)n \log n) = \mathcal{O}(f(d)n \log n)$ parametrized by the treewidth k or tree-depth d .

The essential part is to do the computation on the zeta transformed mirror image of the tree decomposition to save space. As in [41], it is important to introduce the edges in auxiliary leaves to avoid an exponential blow-up.

Chapter 6 |

Space-Efficient Parameterized Algorithms for Counting the Number of solutions to Hamiltonian cycles, and Traveling Salesperson by Dynamic Algebraization

This chapter is based on a joint work with Martin Fürer [49].

An NP-hard graph problem may be intractable for general graphs but it could be efficiently solvable using dynamic programming for graphs with bounded treewidth (or tree-depth or some other structural parameter). Dynamic programming is a well-known approach used for finding exact solutions for NP-hard graph problems based on tree decompositions. It has been shown that there exist algorithms using linear time in the number of vertices and single exponential time in the treewidth (tree-depth or other parameters) of a given tree decomposition for many connectivity problems. Employing dynamic programming on a tree decomposition usually uses exponential space. In 2010, Lokshtanov and Nederlof introduced an elegant framework to avoid exponential space by algebraization. Later, Fürer and Yu modified the framework in a way that even works when the underlying set is dynamic, thus applying it to tree decompositions.

In this work, we design space-efficient algorithms to solve the Hamiltonian Cycle and the Traveling Salesman problems, using polynomial space while the time complexity is only slightly increased. This might be inevitable since we are reducing the space usage from an exponential amount (in dynamic programming solutions) to polynomial. We give an algorithm to solve Hamiltonian cycle in time $\mathcal{O}((4k)^d nM(n \log n))$ using

$\mathcal{O}(kdn \log n)$ space, where $M(r)$ is the time complexity to multiply two integers, each of which being represented by at most r bits. Then, we solve the more general Traveling Salesman problem in time $\mathcal{O}((4k)^d \text{poly}(n))$ using space $\mathcal{O}(\mathcal{W}kdn \log n)$, where k and d are the width and the depth of the given tree decomposition and $\mathcal{W}$ is the sum of weights. Furthermore, this algorithm counts the number of Hamiltonian Cycles.

Dynamic programming (DP) is largely used to avoid recomputing subproblems. It may decrease the time complexity, but it uses auxiliary space to store the intermediate values. This auxiliary space may go up to exponential in the value of the parameter. This means both the running time and the space complexity are exponential for some algorithms solving those NP-complete problems. Space complexity is a crucial aspect of algorithm design, because we typically run out of space before running out of time. To fix this issue, Lokshtanov and Nederlof [40] introduced a framework which works on a static underlying set. The problems they considered were Subset Sum, Knapsack, Traveling Salesman (in time $\mathcal{O}(2^nk)$ using polynomial space), Weighted Steiner Tree, and Weighted Set Cover. They use DFTs, zeta transforms and Möbius transforms [47, 48], leveraging the fact that working on zeta (or discrete Fourier) transformed values is significantly easier since the subset convolution operation converts to a pointwise multiplication operation. In all their settings, the input is a set or a graph which means the underlying set for the subproblems is static. Fürer and Yu [41] changed this approach modifying a dynamic programming algorithm applied to a tree decomposition (instead of the graph itself). The resulting algorithm uses only polynomial space and the running time does not increase drastically. By working with tree decompositions, they obtain parameterized algorithms which are exponential in the tree-depth and linear in the number of vertices. If the tree decomposition has a bounded width, then the algorithm is both fast and space-efficient. In this setting, the underlying set is not static anymore, because they are working with different bags of nodes. They show that using algebraization helps to save space even if the underlying set is dynamic. They consider perfect matchings in their paper. In recent years, there have been several results in this field where algebraic tools are used to save space when DP algorithms are applied to NP-hard problems. In 2018, Pilipczuk and Wrochna [50] applied a similar approach to solve the Minimum Dominating Set problem. Although they have not directly used these algebraic tools but it is a similar approach (in time $\mathcal{O}(3^d \text{poly}(n))$ using $\text{poly}(n)$ space).

We have to mention that there is no general method to automatically transform dynamic programming solutions to polynomial space solutions while increasing the tree-width parameter to the tree-depth in the running time.

One of the interesting NP-hard problems in graph theory is Hamiltonian Cycle. It seems harder than many other graph problems. We are given a graph and we want to find a cycle visiting each vertex exactly once. The naive deterministic algorithm for the Hamiltonian Cycle problem and the more general Traveling Salesman problem runs in time $\mathcal{O}(n!)$ using polynomial space. Later, deterministic DP and inclusion-exclusion algorithms for these two problems running in time $\mathcal{O}^*(2^n)$ ¹ using exponential space were given in [51–53]. The existence of a deterministic algorithm for Hamiltonian cycle running in time $\mathcal{O}((2 - \epsilon)^n)$, for a fixed $\epsilon > 0$ is still an open problem. There are some randomized algorithms which run in time $\mathcal{O}((2 - \epsilon)^n)$, for a fixed $\epsilon > 0$ like the one given in [54]. Although, there is no improvement in deterministic running time, there are some results on Parameterized algorithms. In 2011, Cygan et al. [55] designed a Parameterized algorithm for the Hamiltonian Cycle problem, which runs in time $4^k |V|^{\mathcal{O}(1)}$, where k is the treewidth. They also presented a randomized algorithm for planar graphs running in time $\mathcal{O}(2^{6.366\sqrt{n}})$. In 2015, Bodlaender et al. [56] introduced two deterministic single exponential time algorithms for Hamiltonian Cycle: One based on pathwidth running in time $\tilde{\mathcal{O}}(6^p p^{\mathcal{O}(1)} n^2)$ ² and the other is based on treewidth running in time $\tilde{\mathcal{O}}(15^k k^{\mathcal{O}(1)} n^2)$, where p and k are the pathwidth and the treewidth respectively. The authors also solve the Traveling Salesman problem in time $\mathcal{O}(n(2 + 2^\omega)^p p^{\mathcal{O}(1)})$ if a path decomposition of width p of G is given, and in time $\mathcal{O}(n(7 + 2^{\omega+1})^k k^{\mathcal{O}(1)})$, where ω denotes the matrix multiplication exponent. One of the best known upper bound for ω is 2.3727 [57]. They do not consider the space complexity of their algorithm and as far as we checked it uses exponential space.

Recently, Curticapean et al. [58] showed that there is no positive ϵ such that the problem of counting the number of Hamiltonian cycles can be solved in $\mathcal{O}^*((6 - \epsilon)^p)$ time assuming SETH. Here p is the width of the given path decomposition of the graph. They show this tight lower bound via matrix rank.

Our contribution: Given a graph G along with one of its tree decompositions τ , we present algorithms computing:

- the number of Hamiltonian cycles in time $\mathcal{O}((4k)^d n M(n \log n))$ using $\mathcal{O}(kdn \log n)$ space,
- the answer to the Traveling Salesman problem (TSP) in time $\mathcal{O}((4k)^d \text{poly}(n))$ using space $\mathcal{O}(Wkdn \log n)$,

¹ $\mathcal{O}^*$ notation hides the polynomial factors of the expression.

² $\tilde{\mathcal{O}}$ notation hides the logarithmic factors of the expression.

where k and d are the width and the depth of a given tree decomposition τ , respectively. $M(r)$ is the time complexity to multiply two integers with r bits, and $\mathcal{W}$ is the sum of weights of the edges in TSP.

6.1 Preliminaries

In this section we review notations that we use later.

Definition 6.1.1. The *depth* of a tree decomposition is the maximum number of distinct vertices in the union of all bags on a path from the root to the leaf. We use d to denote the depth of a given tree decomposition (Although, this is not the standard definition of the depth of a tree decomposition, it is equivalent to the standard definition and is more applicable in our case).

We have to mention that this is different from the depth of a tree. This is the depth of a tree decomposition as defined above.

Definition 6.1.2. The *tree-depth* of a graph G , is the minimum over the depths of all tree decompositions of G . We use $d(G)$ (simply d) to denote the tree-depth of a graph.

After defining the treewidth and the tree-depth, now we have to talk about the relationship between these parameters in a given graph G :

Lemma 6.1.1. (see [59, Corollary 2.5] and [43])

For any connected graph G , $d(G) \geq k + 1 \geq \frac{d(G)}{\log_2 |V(G)|}$.

Example: The path P_n with n vertices, has treewidth 1 and tree-depth of $\log_2(n+1)$. One should note that finding the treewidth and the tree-depth of a given graph G are known to be an NP-hard problems. A question which arises here, is whether there exists a tree decomposition such that its width is the treewidth and its depth is the tree-depth of the original graph? In other words, is it possible to construct a tree decomposition of a given graph which minimizes both the depth and the width? If the answer is yes, how to gain such a tree decomposition. Although we are not focusing on this question here, but it seems an interesting problem to think about.

6.1.1 Algebraic tools to save space

When we use dynamic programming to solve a graph problem on a tree decomposition, it usually uses exponential space. Lokshtanov and Nederlof converted some algorithms

using subset convolution or union product into transformed version in order to reduce the space complexity. Later, Fürer and Yu [41] also used this approach in a dynamic setting, based on tree decompositions for the Perfect Matching problem. In this work, we introduce algorithms to solve the Hamiltonian Cycle and the Traveling Salesman problems. First, let us recall some definitions.

Let $\mathcal{R}[2^{\mathcal{U}}]$ be the set of all functions from the power set of the universe $\mathcal{U}$ to the ring $\mathcal{R}$. The operator $\oplus$ is the pointwise addition and the operator $\odot$ is the pointwise multiplication.

Definition 6.1.3. A *relaxation* of a function $f \in \mathcal{R}[2^{\mathcal{U}}]$ is a sequence of functions $\{f^i : f^i \in \mathcal{R}[2^{\mathcal{U}}], 0 \leq i \leq |\mathcal{U}|\}$, where $\forall 0 \leq i \leq |\mathcal{U}|$ and $\forall X \subseteq \mathcal{U}$, $f^i[X]$ is defined as:

$$f^i[X] = \begin{cases} 0 & \text{if } i < |X|, \\ f[X] & \text{if } i = |X|, \\ \text{arbitrary value} & \text{if } i > |X|. \end{cases} \quad (6.1)$$

Definition 6.1.4. The *zeta transform* of a function $f \in \mathcal{R}[2^{\mathcal{U}}]$ is defined as:

$$\zeta f[X] = \sum_{Y \subseteq X} f[Y]. \quad (6.2)$$

Definition 6.1.5. The *Möbius transform* of a function $f \in \mathcal{R}[2^{\mathcal{U}}]$ is defined as:

$$\mu f[X] = \sum_{Y \subseteq X} (-1)^{|X \setminus Y|} f[Y]. \quad (6.3)$$

Lemma 6.1.2. The Möbius transform is the inverse of the zeta transform and vice versa, i.e.

$$\mu(\zeta f[X]) = \zeta(\mu f[X]) = f[X]. \quad (6.4)$$

See [47, 48] for the proof.

Instead of storing exponentially many intermediate results, we store the zeta transformed values. Instead of working on the original computation (bottom up on a nice tree decomposition), we are working on a mirrored computation on the same nice tree decomposition where the zeta-transformed values in a node are computed directly from the zeta-transformed values at the children. At the end, we recover the original values using Equation 6.4. The zeta transform converts the “hard” union product operation ($*_u$) to the “easier” pointwise multiplication operation ($\odot$) which results in saving space.

Definition 6.1.6. Given $f, g \in \mathcal{R}[2^{\mathcal{U}}]$ and $X \in 2^{\mathcal{U}}$, the *Subset Convolution* of f and g denoted $(f *_R g)$ is defined as:

$$(f *_R g)[X] = \sum_{X_1 \subseteq X} f(X_1)g(X \setminus X_1). \quad (6.5)$$

Definition 6.1.7. Given $f, g \in \mathcal{R}[2^{\mathcal{U}}]$ and $X \in 2^{\mathcal{U}}$, the *Union Product* of f and g denoted $(f *_u g)$ is defined as:

$$(f *_u g)[X] = \sum_{X_1 \cup X_2 = X} f(X_1)g(X_2). \quad (6.6)$$

Theorem 6.1.1. ([46]) Applying the zeta transform to a union product operation, results in the pointwise multiplication of zeta transforms of the outputs, i.e., given $f, g \in \mathcal{R}[2^{\mathcal{U}}]$ and $X \in 2^{\mathcal{U}}$,

$$\zeta(f *_u g)[X] = (\zeta f) \odot (\zeta g)[X]. \quad (6.7)$$

All of the previous works, which either used DFT or the zeta transform on a given tree decomposition (such as [41], and [40]), have one common central property that the recursion in the join nodes, can be represented by a formula using a union product operation. The union product and the subset convolution are complicated operations in comparison with pointwise multiplication or pointwise addition. That is why taking the zeta transform of a formula having union products makes the computation easier. As noted earlier in Theorem 6.1.1, taking the zeta transform of such a term, will result in a term having pointwise multiplication instead of union product. After doing a computation over the zeta transformed values, we can apply the Möbius transform on the outcome to get the main result (based on Theorem 6.1.2). In other words, the direct computation has a mirror image using the zeta transforms of the intermediate values instead of the original ones. While the direct computation keeps track of exponentially many intermediate values, the computation over the zeta transformed values partitions into exponentially many branches, and they can be executed one after another. Later, we show that this approach improves the space complexity using only polynomial space instead of exponential space.

6.2 Counting the number of Hamiltonian cycles

We are given a connected graph $G = (V, E)$ and a nice tree decomposition τ of G of width w . If H is a Hamiltonian cycle, then the induced subgraph of H on τ_x (called

$H[V_x]$, where τ_x is the subtree rooted at node x , and V_x is the union of all bags in τ_x) is a set of disjoint paths with endpoints in B_x (see Figure 6.1).

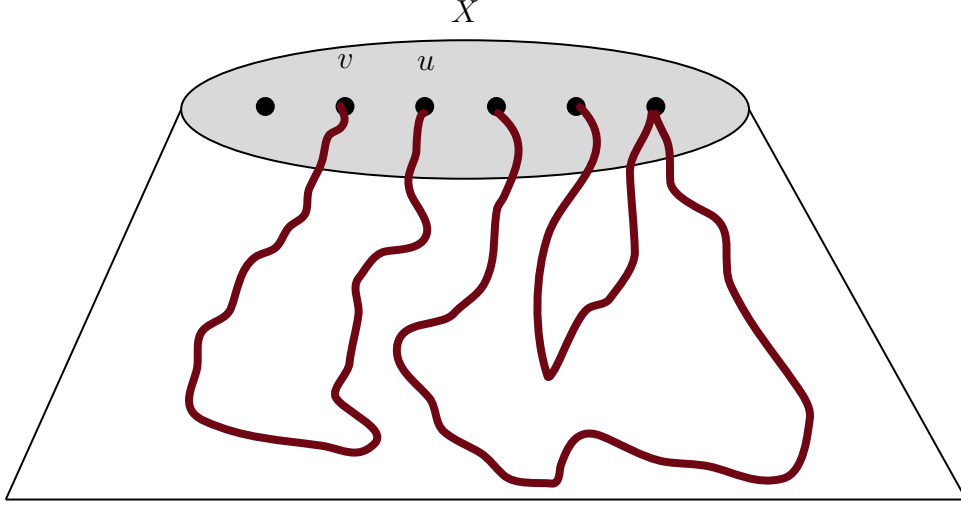


Figure 6.1. $H[V_x]$ is a set of paths with endpoints in B_x

Definition 6.2.1. A *pseudo-edge* is a pair of endpoints of a path of length ≥ 2 with endpoints in B_x and interior vertices in $V_x \setminus B_x$. We use the $\langle, \rangle$ notation for the pseudo-edges. E.g., in Figure 6.1, $p = \langle u, v \rangle$ is a pseudo-edge (it does not imply that there is an edge between u and v , it just says that there is a path of length at least two in $H[V_x]$ where u and v are its endpoints). The $\langle, \rangle$ notation is a symmetrical notation since our paths are undirected, i.e., $\langle u, v \rangle = \langle v, u \rangle$. Each path is associated with a pseudo-edge.

Lemma 6.2.1. The degree of all vertices in $H[V_x]$ is at most 2.

Proof. $H[V_x]$ is a subgraph of the cycle H . □

Let X be the union of pseudo-edges (in B_x). Let $[B_x]^2$ be the set of two-element subsets of B_x , and let $X \subseteq [B_x]^2$. Let S_X be the union of vertices involved in X . Then, S_X is a subset of B_x . Let F_x be the vertices of $V_x \setminus B_x$ which are not present in the bag of the parent of x . Define $f_x[X]$ to be the number of sets of disjoint paths (except in their endpoints where they can share a vertex) whose pseudo-edge set is X (remember S_X is the union of vertices involved in X) visiting vertices of F_x exactly once (they can also visit vertices which are not in F_x , but we require them to visit at least vertices in F_x since they are not present in the proper ancestors of x). we require the root to have empty bag, thus $f_r[\emptyset]$ gives us the number of possible Hamiltonian cycles in G , where r is the root of τ . Now, we show how to compute the values of f_x for all types of nodes.

6.2.1 Computing $f_x[X]$

We first show how $f_x[X]$ could be computed by dynamic programming using exponential space. We introduce the recursive formula for any kind of nodes. In the next section, we will show how to compute $\zeta f_x[X]$ directly.

- **Leaf node:** Assume x is a leaf node.

$$f_x[X] = \begin{cases} 1 & \text{if } X = \emptyset, \\ 0 & \text{otherwise.} \end{cases} \quad (6.8)$$

Since x is a leaf node, there is no path through $V_x \setminus B_x$, so for all non-empty sets X of pseudo-edges, $f_x[X]$ is zero, and for $X = \emptyset$, there is only one set of paths, which is empty.

- **Forget node:** Assume x is a forget node (forgetting vertex v) with a child c , where $B_x = B_c \setminus \{v\}$. Any pseudo-edge $\langle u, w \rangle \in X \subseteq [B_x]^2$ can define a path starting from u , going to v possibly through $V_c \setminus B_c$ and then going to w possibly through $V_c \setminus B_c$. Here, either or both pieces of the path (from u to v , and/or from v to w) can consist of single edges.

$$f_x[X] = \sum_{\langle u, w \rangle \in X} \sum_{Q \subseteq \{\langle u, v \rangle, \langle v, w \rangle\}} d_Q f_c[X \setminus \{\langle u, w \rangle\} \cup Q], \quad (6.9)$$

$$\text{where } d_Q = \begin{cases} 1 & \text{if } \langle u, v \rangle \in Q \text{ or } \{u, v\} \in E, \text{ and } \langle v, w \rangle \in Q \text{ or } \{v, w\} \in E \\ 0 & \text{otherwise.} \end{cases}$$

- **Introduce vertex node:** Assume x is an introduce vertex node (introducing vertex v) with a child c , where $B_x = B_c \cup \{v\}$. The vertex v cannot be an endpoint of a pseudo-edge because v has no neighbors in V_c .

$$f_x[X] = \begin{cases} f_c[X] & \text{if } v \notin S_X, \\ 0 & \text{otherwise.} \end{cases} \quad (6.10)$$

- **Join node:** Assume x is a join node with two children c_1 and c_2 , where $B_x = B_{c_1} = B_{c_2}$. For any given $X \subseteq [B_x]^2$, X can be partitioned into two sets X' and $X \setminus X'$ and each of them can be the set of pseudo-edges for one of the children.

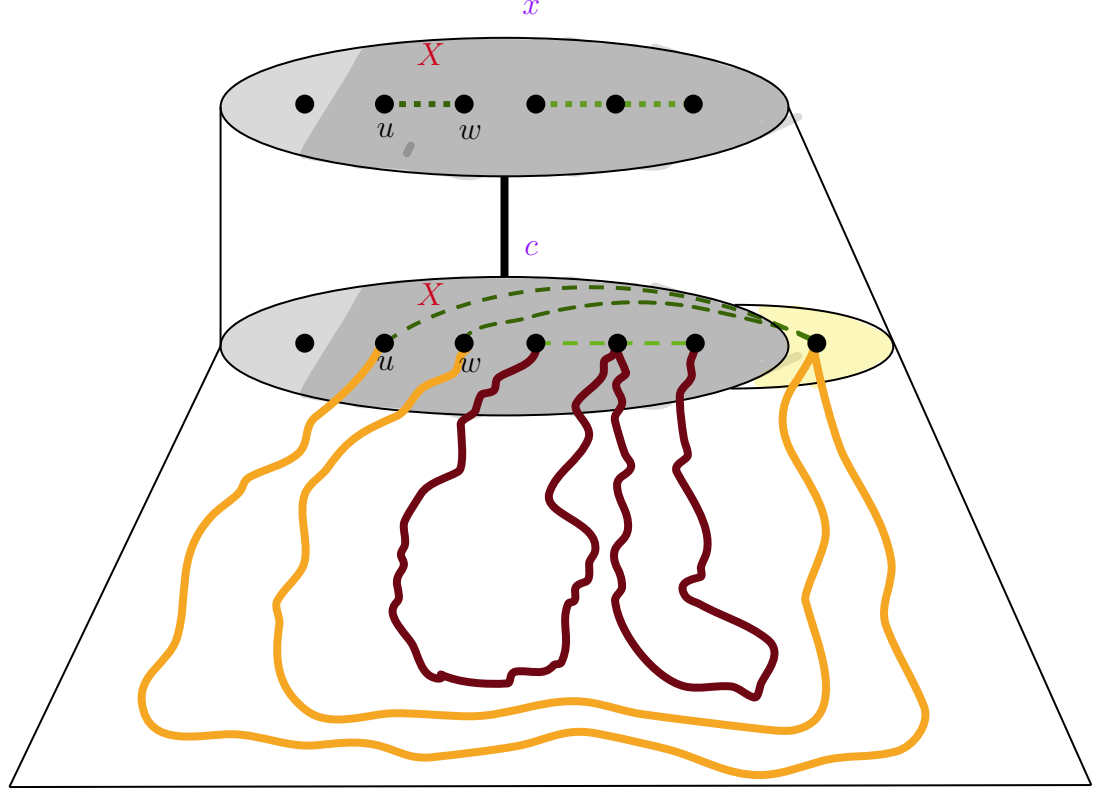


Figure 6.2. Forget node x forgetting vertex v with the child c .

The number of such paths associated with X through V_x is equal to the sum of the products of the number of corresponding paths associated with X' and $X \setminus X'$ through V_{c_1} and V_{c_1} respectively.

$$f_x[X] = \sum_{X' \subseteq X} f_{c_1}[X'] f_{c_2}[X \setminus X'] = (f_{c_1} *_{\mathcal{R}} f_{c_2})[X]. \quad (6.11)$$

Here we get subset convolution and we have to convert it to union product to be able to use zeta transform. We do this conversion in the next subsection.

6.2.2 Computing $\zeta f_x[X]$

In this subsection (the second round of computation), first we compute the relaxations of $f_x[X]$ (Definition 6.1.3) for all kinds of nodes, and then we apply the zeta transform to the relaxations. In the following section let $\{f_x^i\}_{0 \leq i \leq \frac{k+1}{2}}$ be a relaxation of f_x .

- **Leaf node:** Assume x is a leaf node. Since $f_x[\emptyset] = 1$ and for any $X \neq \emptyset : f_x[X] = 0$,

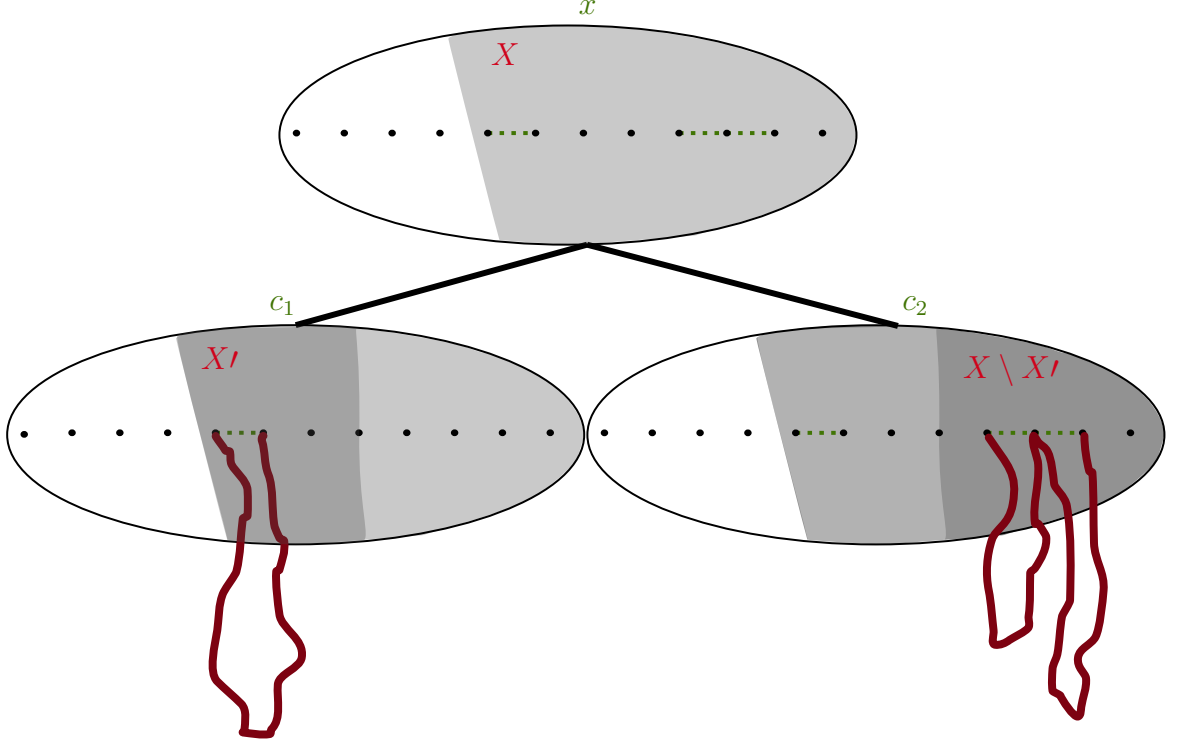


Figure 6.3. Join node x with two children c_1 , and c_2 .

we can choose $f_x^i[X] = f_x[X]$ for all i and X . Then

$$(\zeta f_x^i)[X] = 1, \text{ for all } i \text{ and } X. \quad (6.12)$$

- **Forget node:** Assume x is a forget node (forgetting vertex v) with a child c , where $B_x = B_c \setminus \{v\}$. Thus,

$$f_x^i[X] = \sum_{\langle u, w \rangle \in X} \sum_{Q \subseteq \{\langle u, v \rangle, \langle v, w \rangle\}} d_Q f_c^{i'(Q)}[X \setminus \{\langle u, w \rangle\} \cup Q], \quad (6.13)$$

where $i'(Q) = i - 1 + |Q|$, i.e., $i'(Q)$ is the number of pseudo-edges in $X \setminus \{\langle u, w \rangle\} \cup Q$.

Now we apply the zeta transform:

$$\begin{aligned} (\zeta f_x^i)[X] &= \sum_{Y \subseteq X} f_x^i[Y] \\ &= \sum_{Y \subseteq X} \sum_{\langle u, w \rangle \in Y} \sum_{Q \subseteq \{\langle u, v \rangle, \langle v, w \rangle\}} d_Q f_c^{i'(Q)}[Y \setminus \{\langle u, w \rangle\} \cup Q] \\ &= \sum_{\langle u, w \rangle \in X} \sum_{Q \subseteq \{\langle u, v \rangle, \langle v, w \rangle\}} \sum_{\{\langle u, w \rangle\} \subseteq Y \subseteq X} d_Q f_c^{i'(Q)}[Y \setminus \{\langle u, w \rangle\} \cup Q] \end{aligned}$$

$$= \sum_{\langle u,w \rangle \in X} \sum_{Q \subseteq \{\langle u,v \rangle, \langle v,w \rangle\}} d_Q \sum_{Y \subseteq (X \setminus \{\langle u,w \rangle\})} f_c^{i'(Q)}[Y \cup Q] \quad (6.14)$$

We now express $e_Q = \sum_{Y \subseteq (X \setminus \{\langle u,w \rangle\})} f_c^{i'(Q)}[Y \cup Q]$ by ζ -transforms, depending on the size of Q . We use the abbreviation $X' = X \setminus \{\langle u,w \rangle\}$

If $Q = \emptyset$, then

$$e_Q = \sum_{Y \subseteq X'} f_c^{i-1}[Y] = \zeta f_c^{i-1}[X']. \quad (6.15)$$

If $Q = \{\langle u,v \rangle\}$ or $Q = \{\langle v,w \rangle\}$, then

$$e_Q = \sum_{Y \subseteq X'} f_c^i[Y \cup Q] = \zeta f_c^i[X' \cup Q] - \zeta f_c^i[X']. \quad (6.16)$$

If $Q = \{\langle u,v \rangle, \langle v,w \rangle\}$, then

$$\begin{aligned} e_Q &= \sum_{Y \subseteq X'} f_c^{i+1}[Y \cup Q] \\ &= \zeta f_c^{i+1}[X' \cup Q] - \zeta f_c^{i+1}[X' \cup \{\langle u,v \rangle\}] - \zeta f_c^{i+1}[X' \cup \{\langle v,w \rangle\}] + \zeta f_c^{i+1}[X']. \end{aligned} \quad (6.17)$$

With these sums computed, we can now express $\zeta f_x^i[X]$ more concisely.

$$\begin{aligned} (\zeta f_x^i)[X] &= \sum_{\langle u,w \rangle \in X} ((d_\emptyset - d_{\{\langle u,v \rangle\}} - d_{\{\langle v,w \rangle\}} + d_{\{\langle u,v \rangle, \langle v,w \rangle\}}) \zeta f_c^{i-1}[X'] \\ &\quad + d_{\{\langle u,v \rangle\}} \zeta f_c^i[X' \cup \{\langle u,v \rangle\}] + d_{\{\langle v,w \rangle\}} \zeta f_c^i[X' \cup \{\langle v,w \rangle\}] \\ &\quad + d_{\{\langle u,v \rangle, \langle v,w \rangle\}} \zeta f_c^{i+1}[X' \cup \{\langle u,v \rangle, \langle v,w \rangle\}]). \end{aligned} \quad (6.18)$$

Note that $d_\emptyset = 1$ if $\{\langle u,v \rangle, \langle v,w \rangle\} \subseteq E$, $d_{\{\langle u,v \rangle\}} = 1$ if $\langle v,w \rangle \in E$, $d_{\{\langle v,w \rangle\}} = 1$ if $\langle u,v \rangle \in E$, and $d_{\{\langle u,v \rangle, \langle v,w \rangle\}}$ is always 1, while otherwise $d_Q = 0$. This implies that for every $\langle u,w \rangle$ in the previous equation at least one of the 4 coefficients is 0. Therefore, in each forget node, we have at most a $3|X|$ fold branching.

- **Introduce vertex node:** Assume x is an introduce vertex node (introducing vertex v) with a child c , where $B_x = B_c \cup \{v\}$. Let X_v be the set of pseudo-edges

having v as one their endpoints. Therefore,

$$f_x^i[X] = \begin{cases} f_c^i[X] & \text{if } v \notin S_X, \\ 0 & \text{otherwise.} \end{cases} \quad (6.19)$$

$$(\zeta f_x^i)[X] = \sum_{Y \subseteq X} f_x^i[Y] = \sum_{Y \subseteq (X \setminus X_v)} f_c^i[Y] = (\zeta f_c^i)[X \setminus X_v]. \quad (6.20)$$

- **Join node:** Assume x is a join node with two children c_1 and c_2 , where $B_x = B_{c_1} = B_{c_2}$. To compute f_x on a join node, we can use Equation 6.11. In order to convert the subset convolution operation to pointwise multiplication, first we need to convert the subset convolution to a union product, and then we are able to use Theorem 6.1.1. To convert subset convolution to union product we introduce a relaxation of f_x . Let $f_x^i[X] = \sum_{j=0}^i (f_{c_1}^j *_{\cup} f_{c_2}^{i-j})[X]$ be a relaxation of f_x .

$$(\zeta f_x^i)[X] = \sum_{j=0}^i (\zeta f_{c_1}^j)[X] \cdot (\zeta f_{c_2}^{i-j})[X], \text{ for } 0 \leq i \leq k, \quad (6.21)$$

where k is the treewidth of τ .

To summarize, we present the following algorithm (Algorithm 15) for the Hamiltonian Cycle problem where a tree decomposition τ is given.

Algorithm 15 Counting the total number of the possible Hamiltonian cycles in a graph given by a nice tree decomposition

Require: A nice tree decomposition τ with root r

Ensure: $(\zeta f)(r, \emptyset, 0)$

```

1: procedure  $(\zeta f)(x, X, i)$ :  $\triangleright (\zeta f)(x, X, i)$  is the representation of  $(\zeta f_x^i)[X]$ 
2:   if  $x$  is a leaf node then
3:     return 1.
4:   end if
5:   if  $x$  is a forget node then  $\triangleright$  forgetting  $v$ 
6:     return the value according to Equation 6.18.
7:   end if
8:   if  $x$  is an introduce vertex node then  $\triangleright$  introducing  $v$ 
9:     return  $(\zeta f)(c, X \setminus X_v, i)$ .
10:  end if
11:  return  $\sum_{j=0}^i (\zeta f)(c_1, X, j) \cdot (\zeta f)(c_2, X, i - j)$ .  $\triangleright x$  is a join node
12: end procedure

```

Theorem 6.2.1. Given a graph $G = (V, E)$ and a tree decomposition τ of G , we can compute the total number of the Hamiltonian cycles of G in time $\mathcal{O}((4k)^d n M(n \log n))$ and in space $\mathcal{O}(kdn \log n)$ by using Algorithm 15 where k and d are the width and the depth of τ respectively, and $M(n)$ is the time complexity to multiply two numbers which can be encoded by at most n bits.

Proof. Branching happens only in forget nodes ($\leq d$ forget nodes from a path from the root to a leaf), where we branch into $\leq k$ branches. According to the formula for forget node: $\{u, v\}$ and $\{v, w\}$ each can be an edge or a pseudo-edge (four possibilities). There are at most $n!$ Hamiltonian cycles, so the value of f_x can be presented by at most $n \log n$ bits. It takes $M(n \log n)$ time to multiply numbers with this length. On the other hand, there are n vertices in G . All together, the running time is $\mathcal{O}((4k)^d n M(n \log n))$.

For space complexity, we only keep results for one strand at a time. There are $\leq d$ vertices on each path from the root to a leaf. On each path, we keep track of vertices in a bag (at most $k + 1$), and the number of disjoint paths can be presented by $n \log n$ bits. Therefore, the space complexity is $\mathcal{O}(kdn \log n)$.

The correctness of the algorithm is shown in Section 6.2.1 and Section 6.2.2. Now, we show the running time and the space complexity of our algorithm.

Running time: The only case that branching happens in where we are handling a forget node. We have at most k branches in forget nodes since the number of pseudo-edges in each bag is bounded by $k + 1$ because of Lemma 6.2.1. Furthermore, based on the formula for forget node, the two unordered pairs of (u, v) and (v, w) can contribute as an edge or a pseudo-edge (four possible cases). On the other hand the number of forget nodes in a path from the root to one leaf is bounded by the d (the depth of tree decomposition τ). Also, the number of vertices is n and we work with numbers of size at most $n!$ (number of paths) which can be represented by at most $n \log n$ bits. We handle multiplication of these numbers which happens in $\mathcal{O}(M(n \log n))$. All being said, the total running time is $\mathcal{O}((4k)^d n M(n \log n))$.

Space complexity: We keep the results for each strand at a time. The number of nodes in a path from the root to a leaf is bounded by the depth of the tree decomposition (d). Along the path, we keep track of bags (of size k) and number of disjoint paths (at most $n!$ paths exist which can be shown by at most $n \log n$ bits). Therefore, the total space complexity is $\mathcal{O}(kdn \log n)$. $\square$

6.3 The traveling Salesman problem

In the previous section, we showed how to count the total number of possible Hamiltonian cycles of a given graph. In this section, we discuss a harder problem. We know that the Hamiltonian Cycle problem is reducible to the Traveling Salesman problem (TSP) by setting all cost of edges to be 1. We could have just explained how to solve TSP but we chose to first explain the easier one to help understanding the process. Now, we have most of the notations. First, we recap the formal definition of the Traveling Salesman problem.

Definition 6.3.1. Traveling Salesman. Let $G = (V, E)$ be an undirected graph with weighted edges, where for all $\{u, v\} \in E$, $c_{u,v}$ is the weight (= cost) of $\{u, v\}$ (nonnegative integer). In the traveling Salesman problem we are asked to find a cycle (if there is any) that visits all of the vertices exactly once (it is a Hamiltonian cycle) with minimum cost.

As mentioned, the output should be a minimum cost Hamiltonian cycle. We have the same notations as before. Thus, we are ready to explain our algorithmic framework: The difference between counting the number of Hamiltonian cycles and finding the minimum cost of a Hamiltonian cycle (answer to the Traveling salesman problem), is that we should work with lengths instead of the numbers of solutions. In order to solve this problem, we work with the ring of polynomials $\mathbb{Z}[x]$, where x is a variable. Our algorithm computes the polynomial $P_x(y)[X] = \sum_{i=0}^{\mathcal{W}} a_i^x[X] y^i$, where $a_i^x[X]$ is the number of solutions in τ_x associated with X with cost i , and $\mathcal{W}$ is the sum of the weights of all edges.

6.3.1 Computing the Hamiltonian Cycles of All Costs

To find the answer to the traveling salesman problem, we have to find the first nonzero coefficient of $P_r(x)$, where r is the root of the given nice tree decomposition. As we did for the Hamiltonian cycle problem, we show how compute this polynomial recursively for all kinds of nodes.

- **Leaf node:** Assume x is a leaf node.

$$P_x(y)[X] = \begin{cases} 1 & \text{if } X = \emptyset, \\ 0 & \text{otherwise.} \end{cases} \quad (6.22)$$

- **Forget node:** Assume x is a forget node (forgetting vertex v) with a child c , where $B_x = B_c \setminus \{v\}$.

$$P_x(y)[X] = \sum_{\langle u, w \rangle \in X} \sum_{Q \subseteq \{\langle u, v \rangle, \langle v, w \rangle\}} d_Q(P_c(y))[X \setminus \{\langle u, w \rangle\} \cup Q], \quad (6.23)$$

where d_Q is defined in a similar way as in Equation 6.9.

- **Introduce vertex node:** Assume x is an introduce vertex node (introducing vertex v) with a child c , where $B_x = B_c \cup \{v\}$.

$$P_x(y)[X] = \begin{cases} P_c(y)[X] & \text{if } v \notin S_X, \\ 0 & \text{otherwise.} \end{cases} \quad (6.24)$$

- **Join node:** Assume x is a join node with two children c_1 and c_2 , where $B_x = B_{c_1} = B_{c_2}$.

$$f_x[X] = \sum_{X' \subseteq X} P_{c_1}(y)[X'] P_{c_2}(y)[X \setminus X']. \quad (6.25)$$

We skip the zeta transform part because it is similar to the Hamiltonian Cycle case.

Theorem 6.3.1. Given a graph $G = (V, E)$ and a tree decomposition τ of G , we can solve the Traveling Salesman problem for G in time $\mathcal{O}(4k)^d \cdot \text{poly}(n)$ and in space $\mathcal{O}(\mathcal{W}kdn \log n)$ where $\mathcal{W}$ is the sum of the weights, k and d are the width and the depth of the tree decomposition τ respectively.

The proof is omitted due to space constraints.

6.3.2 Conclusion

In this work, we solved Hamiltonian Cycle with polynomial space complexity and Traveling Salesman with pseudo-polynomial space complexity, while the running time is polynomial in size of the given graph and exponential in the tree-depth. Our algorithms for both problems rely on modifying a DP approach such that instead of storing all possible intermediate values, we keep track of zeta transformed values which was first introduced in [40], and then in [41] for dynamic underlying sets.

Chapter 7

Conclusion

During my PhD, my co-authors and I presented three algorithms to solve treewidth approximation in FPT time. The most recent algorithm is the state of the art and runs in time $\mathcal{O}(2^{6.15k}n)$, which is a 2-approximation. It is not obvious if one can achieve an approximation ratio better than 2 while having time complexity of form $c^k n$. A possible (lower bound) hardness result could be a huge break through here. The last worked is inspired by [29].

On the second part of my PhD, I focused on optimizing space complexity. Often times when DP is used on tree decomposition, the algorithm uses exponential space, which is impractical. My advisor and I showed how one can avoid the exponential space usage by dynamic algebraization for the following problems:

1. counting the number of minimum domination sets,
2. counting the number of Hamiltonian cycles, and
3. counting the number of solutions to travelling salesman problem.

Dynamic algebraization was introduced in [41], which itself was inspired by [40].

Bibliography

- [1] KORHONEN, T. (2021) “Single-Exponential Time 2-Approximation Algorithm for Treewidth,” *arXiv preprint arXiv:2104.07463*.
- [2] KNUTH, D. E. (1976) “Big omicron and big omega and big theta,” *ACM Sigact News*, **8**(2), pp. 18–24.
- [3] CORMEN, T. H., C. E. LEISERSON, R. L. RIVEST, and C. STEIN (2022) *Introduction to algorithms*, MIT press.
- [4] COURCELLE, B. (1990) “The monadic second-order logic of graphs. I. Recognizable sets of finite graphs,” *Information and Computation*, **85**(1), pp. 12–75.
- [5] COURCELLE, B. and J. ENGELFRIET (2012) *Graph structure and monadic second-order logic: a language-theoretic approach*, vol. 138, Cambridge University Press.
- [6] ARNBORG, S., D. G. CORNEIL, and A. PROSKUROWSKI (1987) “Complexity of finding embeddings in ak-tree,” *SIAM Journal on Algebraic Discrete Methods*, **8**(2), pp. 277–284.
- [7] WOEGINGER, G. J. (2004) “Space and Time Complexity of Exact Algorithms : Some Open Problems,” .
- [8] BELBASI, M. and M. FÜRER (2020), “An Improvement of Reed’s Treewidth Approximation,” [2010.03105](#).
- [9] BERTELE, U. and F. BRIOSCHI (1973) “On non-serial dynamic programming,” *Journal of Combinatorial Theory, Series A*, **14**(2), pp. 137–148.
- [10] HALIN, R. (1976) “S-functions for graphs,” *Journal of Geometry*, **8**(1-2), pp. 171–186.
- [11] ROBERTSON, N. and P. D. SEYMOUR (1984) “Graph minors. III. Planar tree-width,” *Journal of Combinatorial Theory, Series B*, **36**(1), pp. 49–64.
- [12] BODLAENDER, H. L. (2005) “Discovering treewidth,” in *International Conference on Current Trends in Theory and Practice of Computer Science*, Springer, pp. 1–16.

- [13] REED, B. A. (1992) “Finding approximate separators and computing tree width quickly,” in *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, ACM, pp. 221–228.
- [14] ROBERTSON, N. and P. D. SEYMOUR (1995) “Graph minors. XIII. The disjoint paths problem,” *Journal of Combinatorial Theory, Series B*, **63**(1), pp. 65–110.
- [15] LAGERGREN, J. (1996) “Efficient Parallel Algorithms for Graphs of Bounded Tree-Width,” *Journal of Algorithms*, **20**(1), pp. 20–44.
URL <http://www.sciencedirect.com/science/article/pii/S0196677496900024>
- [16] BODLAENDER, H. L. (1996) “A linear-time algorithm for finding tree-decompositions of small treewidth,” *SIAM Journal on Computing*, **25**(6), pp. 1305–1317.
- [17] AMIR, E. (2010) “Approximation Algorithms for Treewidth,” *Algorithmica*, **56**(4), pp. 448–479.
URL <https://doi.org/10.1007/s00453-008-9180-4>
- [18] FEIGE, U., M. HAJIAGHAYI, and J. R. LEE (2008) “Improved Approximation Algorithms for Minimum Weight Vertex Separators,” *SIAM Journal on Computing*, **38**(2), pp. 629–657.
- [19] BODLAENDER, H. L., P. G. DRANGE, M. S. DREGI, F. V. FOMIN, D. LOK-SHTANOV, and M. PILIPCZUK (2016) “A $\mathcal{O}(c^k n)$ 5-Approximation Algorithm for Treewidth,” *SIAM Journal on Computing*, **45**(2), pp. 317–378.
- [20] FOMIN, F. V., I. TODINCA, and Y. VILLANGER (2015) “Large induced subgraphs via triangulations and CMSO,” *SIAM Journal on Computing*, **44**(1), pp. 54–87.
- [21] BELBASI, M. and M. FÜRER (2021) “An improvement of Reed’s treewidth approximation,” in *International Workshop on Algorithms and Computation*, Springer, pp. 166–181.
- [22] ——— (2021) “Finding All Leftmost Separators of Size $\leq k$,” in *International Conference on Combinatorial Optimization and Applications*, Springer, pp. 273–287.
- [23] FLUM, J. and M. GROHE (2006) *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series)*, Springer-Verlag, Berlin, Heidelberg.
- [24] MARX, D. (2006) “Parameterized graph separation problems,” *Theoretical Computer Science*, **351**(3), pp. 394–406.
- [25] CHITNIS, R., M. HAJIAGHAYI, and D. MARX (2013) “Fixed-parameter tractability of directed multiway cut parameterized by the size of the cutset,” *SIAM Journal on Computing*, **42**(4), pp. 1674–1696.

- [26] LOKSHTANOV, D. and D. MARX (2013) “Clustering with local restrictions,” *Information and Computation*, **222**, pp. 278–292.
- [27] MENGER, K. (1927) “Zur allgemeinen Kurventheorie,” *Fundamenta Mathematicae*, **10**(1), pp. 96–115.
- [28] MARX, D. and I. RAZGON (2014) “Fixed-parameter tractability of multicut parameterized by the size of the cutset,” *SIAM Journal on Computing*, **43**(2), pp. 355–388.
- [29] KORHONEN, T. (2022) “A Single-Exponential Time 2-Approximation Algorithm for Treewidth,” in *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE, pp. 184–192.
- [30] BELLENBAUM, P. and R. DIESTEL (2002) “Two short proofs concerning tree-decompositions,” *Combinatorics, Probability and Computing*, **11**(6), pp. 541–547.
- [31] KLOKS, T. (1994) *Treewidth: computations and approximations*, Springer.
- [32] PERKOVIĆ, L. and B. REED (2000) “An improved algorithm for finding tree decompositions of small width,” *International Journal of Foundations of Computer Science*, **11**(03), pp. 365–371.
- [33] BELBASI, M. and M. FÜRER (2017) “Saving Space by Dynamic Algebraization Based on Tree Decomposition: Minimum Dominating Set,” *arXiv preprint arXiv:1711.10088*.
- [34] TARJAN, R. E. and A. E. TROJANOWSKI (1977) “Finding a maximum independent set,” *SIAM Journal on Computing*, **6**(3), pp. 537–546.
- [35] JIAN, T. (1986) “An $O(2.304n)$ algorithm for solving maximum independent set problem,” *IEEE Transactions on Computers*, **100**(9), pp. 847–851.
- [36] GRANDONI, F. (2006) “A note on the complexity of minimum dominating set,” *Journal of Discrete Algorithms*, **4**(2), pp. 209–214.
- [37] FOMIN, F. V., F. GRANDONI, and D. KRATSCH (2005) “Measure and conquer: Domination—a case study,” in *International Colloquium on Automata, Languages, and Programming*, Springer, pp. 191–203.
- [38] DAVIS, M. and H. PUTNAM (1960) “A computing procedure for quantification theory,” *Journal of the ACM (JACM)*, **7**(3), pp. 201–215.
- [39] VAN ROOIJ, J. M. and H. L. BODLAENDER (2008) “Design by measure and conquer, a faster exact algorithm for dominating set,” *arXiv preprint arXiv:0802.2827*.
- [40] LOKSHTANOV, D. and J. NEDERLOF (2010) “Saving space by algebraization,” in *Proceedings of the Forty-second ACM Symposium on Theory of Computing*, pp. 321–330.

- [41] FÜRER, M. and H. YU (2017) “Space saving by dynamic algebraization based on tree-depth,” *Theory of Computing Systems*, **61**(2), pp. 283–304.
- [42] CYGAN, M., F. V. FOMIN, Ł. KOWALIK, D. LOKSHTANOV, D. MARX, M. PILIPCZUK, M. PILIPCZUK, and S. SAURABH (2015) *Parameterized algorithms*, vol. 5, Springer.
- [43] BODLAENDER, H. L., J. R. GILBERT, H. HAFSTEINSSON, and T. KLOKS (1995) “Approximating treewidth, pathwidth, frontsize, and shortest elimination tree,” *Journal of Algorithms*, **18**(2), pp. 238–255.
- [44] AMIR, E. (2013) “Efficient approximation for triangulation of minimum treewidth,” *arXiv preprint arXiv:1301.2253*.
- [45] BOUCHITTÉ, V., D. KRATSCH, H. MÜLLER, and I. TODINCA (2004) “On treewidth approximations,” *Discrete Applied Mathematics*, **136**(2-3), pp. 183–196.
- [46] BJÖRKLUND, A., T. HUSFELDT, P. KASKI, and M. KOIVISTO (2007) “Fourier meets Möbius: fast subset convolution,” in *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pp. 67–74.
- [47] ROTA, G.-C. (1964) “On the foundations of combinatorial theory I. Theory of Möbius functions,” *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, **2**(4), pp. 340–368.
- [48] STANLEY, R. P. and G.-C. ROTA (1997) *Enumerative Combinatorics*, vol. 1 of *Cambridge Studies in Advanced Mathematics*, Cambridge University Press.
- [49] BELBASI, M. and M. FÜRER (2019) “A space-efficient parameterized algorithm for the hamiltonian cycle problem by dynamic algebraization,” in *International Computer Science Symposium in Russia*, Springer, pp. 38–49.
- [50] PILIPCZUK, M. and M. WROCHNA (2018) “On space efficiency of algorithms working on structural decompositions of graphs,” *ACM Transactions on Computation Theory (TOCT)*, **9**(4), pp. 1–36.
- [51] KARP, R. M. (1982) “Dynamic programming meets the principle of inclusion and exclusion,” *Operations Research Letters*, **1**(2), pp. 49–51.
- [52] KOHN, S., A. GOTTLIEB, and M. KOHN (1977) “A generating function approach to the traveling salesman problem,” in *Proceedings of the 1977 annual conference*, pp. 294–300.
- [53] BELLMAN, R. (1962) “Dynamic programming treatment of the travelling salesman problem,” *Journal of the ACM (JACM)*, **9**(1), pp. 61–63.
- [54] BJÖRKLUND, A., P. KASKI, and I. KOUTIS (2016) “Directed Hamiltonicity and out-branchings via generalized Laplacians,” *arXiv preprint arXiv:1607.04002*.

- [55] CYGAN, M., J. NEDERLOF, M. PILIPCZUK, M. PILIPCZUK, J. M. VAN ROOIJ, and J. O. WOJTASZCZYK (2011) “Solving connectivity problems parameterized by treewidth in single exponential time,” in *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, IEEE, pp. 150–159.
- [56] BODLAENDER, H. L., M. CYGAN, S. KRATSCH, and J. NEDERLOF (2015) “Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth,” *Information and Computation*, **243**, pp. 86–111.
- [57] WILLIAMS, V. V. (2012) “Multiplying matrices faster than Coppersmith-Winograd,” in *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pp. 887–898.
- [58] CURTICAPEA, R., N. LINDZEY, and J. NEDERLOF (2018) “A tight lower bound for counting Hamiltonian cycles via matrix rank,” in *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, pp. 1080–1099.
- [59] NEŠETŘIL, J. and P. O. DE MENDEZ (2006) “Tree-depth, subgraph coloring and homomorphism bounds,” *European Journal of Combinatorics*, **27**(6), pp. 1022–1041.

Vita

Mahdi Belbasi

EDUCATION

- **Pennsylvania State University** University Park, PA
PhD in Computer Science and Engineering Jan 2017 - May 2022
- **Sharif University of Technology** Tehran, Iran
BSc in Computer Engineering and minor in Mathematics Sep 2011 - June 2016

EXPERIENCE

- **Research Assistant (PhD candidate)** Penn State, University Park, PA
 - Designs and analyzes algorithms to obtain time and space-efficient solutions
 - Bridges between theory and application
 - Designed the fastest parameterized treewidth approximation algorithm
- **Instructor** Penn State, University Park, PA
 - Taught Theory of Computation (CMPSC 464) to over 200 students
- **Part-Time Researcher** Penn State, University Park, PA
 - Estimated k-mer abundance histogram using Bayes classifiers
 - Designed and simulated Hard-core model under different boundary conditions
 - Proved that sketch-based Jaccard estimators are biased

AWARDS AND INVOLVEMENT

- **Best Teaching Assistant Award in EECS Department**, Pennsylvania State University Spring 2021
- **Best Paper Award**, the 15th Annual International Conference on Combinatorial Optimization and Applications Dec 2021
- **Counselor at AI Summer Camp for Middleschool Girls (Volunteer)**, Pennsylvania State University Summer 2021
- **Treasurer of the Iranian Student Association (Volunteer)**, Pennsylvania State University Fall 2017

PUBLICATION

- **The Minimizer Jaccard Estimator Is Biased and Inconsistent** (submitted to ISMB 2022), Mahdi Belbasi, Antonio Blanca, Robert Harris, David Koslicki, Paul Medvedev
- **Faster Treewidth Approximation** (to be submitted to SODA 2023), Mahdi Belbasi, Martin Fürer, Medha Kumar
- **Finding All Leftmost Separators of Size $\leq k$** (COCOA 2021), Mahdi Belbasi, Martin Fürer
- **An Improvement of Reed's Treewidth Algorithm** (WALCOM 2021), Mahdi Belbasi, Martin Fürer
- **A Space-Efficient Parameterized Algorithm for the Hamiltonian Cycle Problem by Dynamic Algebraization** (CSR 2019), Mahdi Belbasi, Martin Fürer
- **Saving Space by Dynamic Algebraization Based on Tree Decomposition: Minimum Dominating Set** (arXiv preprint 2017), Mahdi Belbasi, Martin Fürer