Computation of Treespan

A Generalization of Bandwidth to Treelike Structures

by

Markus Sortland Dregi

Thesis

for the degree

Master of Science in Informatics

Faculty of Mathematics and Natural Sciences
University of Bergen

June, 2012

Abstract

Motivated by a search game, Fomin, Heggernes and Telle [Algorithmica, 2005] defined the graph parameter treespan, a generalization of the well studied parameter bandwidth. Treespan is the maximum number of appearances of a vertex in an ordered tree decomposition, i.e. a tree decomposition introducing at most one new vertex in each bag. In this thesis, we investigate the computational tractability of the problem Treespan, which aims to decide whether the treespan of a given graph is at most a given integer k. First we introduce a new perspective to the problem, with an equivalent parameter which we call adjacencyspan. It provides, in our opinion, a clearer understanding of the nature of the problem.

We provide structural results related to adjacencyspan, and combine these with dynamic programming to solve Treespan in polynomial time for fixed values of k and hence prove the problem to be in XP. Fomin et al. [Algorithmica, 2005] asked whether Treespan is polynomial time solvable for trees of degree higher than 3 as their final open problem. We solve this problem by proving Treespan to be polynomial time solvable for trees of bounded maximum degree d, for every fixed d. In the area of fixed parameter tractability we give a polynomial kernel for Treespan parameterized by both the required treespan and the vertex cover number of the input graph.

It is a classical result, first proven by Lenstra [Mathematics of Operations Research, 1983], that p-Integer Linear Programming Feasibility is fixed parameter tractable. In his book "Invitation to Fixed-Parameter Algorithms", Niedermeier specifically asks for more applications of this result. In this thesis we provide another application by using it to obtain a fixed parameter tractable algorithm for Treespan parameterized by the vertex cover number.

The thesis do not only have theoretical implications, but we give algorithms that by far outperform previously known algorithms in practical terms.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor Professor Pinar Heggernes for believing in me, encouraging me and being my academic compass for the last year. Your always open door and willingness to spend time on me and my questions when I needed it, never stopped to amaze me. Thank you!

My deepest thanks to Dr. Daniel Lokshtanov for introducing me to the beauty of algorithms and discrete mathematics in the first place. To Jing, Torgeir and the rest of my fellow students at the University of Oslo for making my years as a bachelor student unforgettable. To Pål[1], Sigve and Erik for warmly welcoming me to the study hall at the University of Bergen and introducing me to various algorithmic concepts at the whiteboard. And to all of the algorithm group for including me in the lot, in both academics and at a personal level. I cannot express how much I have learned during our discussions, Friday seminars and winter schools!

There is also a lot of people outside academia which deserves to be mentioned. My family and friends, in particular my parents Terje and Marita and my siblings Nikolai and Sofie, for supporting me, challenging me and making me laugh. All my fellow karatekas from Bergen Karate Klubb, which I have shared so many great experiences with in the dojo. And to the girl who mean the world to me, Anette, I cannot thank you enough. I will always be grateful for your love and support, both when I am at home and when I am working late.

I would also like to use this opportunity to express my gratitude to Norway for providing excellent, free education and being such a nice place to live.

[1] A bonus $\star$ to Pål for reading my thesis and providing me with valuable comments.

Contents

Part I

Preliminaries

Chapter 1

Introduction

1.1 Graphs and algorithms, why?

In this chapter we will explain and motivate key notions in the fields of algorithms and graph theory. Our aim is that it will be both interesting and manageable for the reader, independent of his or her background.

Less is more

Imagine that you and five others are hanging out. Some of them are your friends and some you have never met before. By chance, you stumble upon the Annual Team Competition, ATC. The prize being a boat trip, you see this as a perfect opportunity to get to know each other better. The contest is, as the name says, a team competition. You take a look in the ATC brochure and in the rules section it says that the team must consist of exactly three people. You read further to the tip section and it is written that former years, teams consisting of only people who know each other or people not knowing each other at all tend to do very well. They believe this is because everybody is just as likely to communicate, as everyone knows each other equally. We now claim that within your group there is such a good team of size three.

But how can this be? You have some information, but none saying anything about winning teams within your group. You know that there is an Annual Team Competition, that there is a prize, a brochure consisting of among others a rules and a tip section and that you are hanging out with some people. But how can you use this information to deduce truths? To state facts, like there being a homogeneous team of size three? To be able to do this, we prefer to give a model. A model abstracts away some information and by this, if you pick the right model, the interesting information becomes clearer. We will use a well studied model called graphs, which dates back to the time of a famous mathematician named Euler from the 18th century,

to describe our information. A graph can be seen as some dots, with lines drawn between some of the pairs.

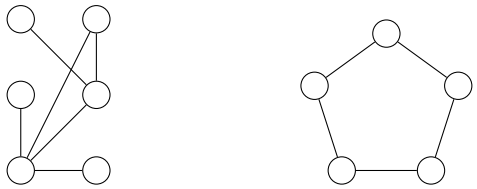


Figure 1.1: Two examples of graphs.

For each of the people in the group we will use a dot to represent her. And between each two dots, we will draw a line if these two are friends. Hence the left example graph in Figure 1.1 can be a model of six people hanging out, while in the right example there would only be five people. We lose sight of the brochure and prize, but maybe we still have the information we need to gain the prize in the end. In the same fashion you could use graphs to model your friends in some social network like Google+ or Facebook. Even computers connected by ethernet cables or any other kind of objects with some relation between them. Your navigation system uses graphs, with weights on the lines, to model roads and intersections. Euler himself used graphs to solve an old mathematical problem called the seven bridges of Königsberg, where he used graphs to model islands and bridges. Further into this thesis, we will refer to the dots as vertices and to the lines as edges. For a formal definition of graphs, we refer to Chapter 2. But let us get back to the team choosing process. We want to argue that there is some good team within your group. The only information we have about your friendship graph is that there are six dots. We actually know nothing about the friendships.

Figure 1.2: The two types of homogenous teams.

From your point of view, you might know some people of the group and not the rest. As there are five others in the group, you either have at least three friends or at least three non-friends. Because, if you have at most two of each of them, how can there be five people? To make a general argument, we will now use the color red to describe the way you relate to the most of the group (it might be friendships or non-friendships) and blue to describe the other. Then the graph, from your point of view looks like Figure 1.3.

Note that if two of the dots you connect with in red also connect in red,

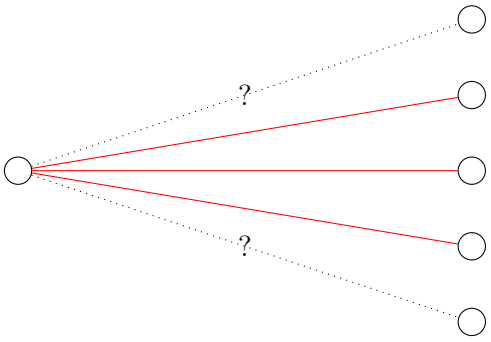


Figure 1.3: The graph from your point of view, with you on the left. The two question marks are relations we do not need any information about.

you get a red triangle. This red triangle would either be a team of only friends or non-friends, depending on what red represents. And hence we would have a solution. Assume that none of these three relate in red. Then they must all relate in blue and hence you get the situation in Figure 1.4, a blue triangle. This completes the argument. No matter how you and the rest of the group relate friendship wise, it is possible to pick a good team for the competition. The best of luck!

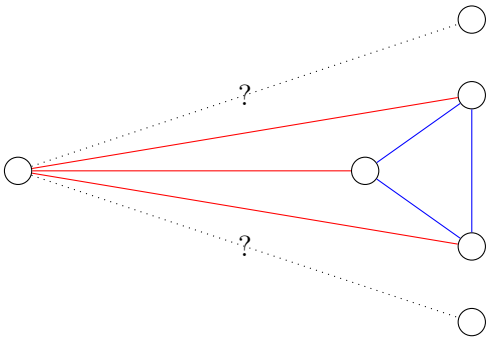


Figure 1.4: A blue triangle occurs when we try to avoid the red ones.

It was not a coincidence that we asked you to imagine to be part of a group of six people. Note that the right example in Figure 1.1 has neither a team of friends nor non-friends. If we combine this observation with our argument we see that graphs of size six are the smallest graphs we can give this kind of guarantee for.

From now on, a set of vertices which are all pairwise connected by edges will be called a clique. And those without any edges will be called independent sets. Given a number k, the smallest number such that every graph with this number of vertices contains a clique or an independent set of size k is called $R(k, k)$ or the diagonal Ramsey numbers. We have just proved

that $R(3,3)$ is 6. It is known that $R(4,4)$ is 18 [Bón06], while for $R(5,5)$ we only know that it lies between 43 and 49 [Rad09]. This seems quite strange. Since we are proving properties for finite, small graphs, could we not just use a computer to generate all graphs of this size and check whether they all satisfy this property or not? This brings us to our next topic, algorithms.

Algorithms

> "Erdős asks us to imagine an alien force, vastly more powerful than us, landing on Earth and demanding the value of R(5, 5) or they will destroy our planet. In that case, he claims, we should marshal all our computers and all our mathematicians and attempt to find the value. But suppose, instead, that they ask for R(6, 6). In that case, he believes, we should attempt to destroy the aliens."
> – Joel Spencer [Spe94]

This quote says something about how fast problems can get hard to solve. The limitation of the computation of solutions for these problems does not lie in making a computer compute, but in the humongous amount of time it would require. Most decent first year computer science students would be able to write such a program. Generate all graphs of a certain size and test whether they contain a clique or independent set of size k. Assume that you build a computer, which has the computational power of the fastest computer on earth multiplied with the number of particles in the universe multiplied with the age of the universe. This computer would still require more time than the age of the universe to generate all graphs on 30 vertices.[1] There are more clever ways to do the computations, but still, they are not good enough to decide the number $R(5,5)$.

This example should motivate us to care about the time complexity of our computations. Because who has as much time as the age of the universe but the universe itself? Before we can say anything about time complexity though, we need to describe how we do computing. And even before this, we must define how we aks questions. A question can be asked in so many ways, but the questions of our concern will be the ones we can answer either with a yes or a no: The decision problems. For solving decision problems we have several models, two of them being the Turing machine and the equivialent notion of algorithms. The Turing machine was introduced by Alan Turing in 1936 [Tur38] and consists of a tape, a head that can read and write on this tape and a finite amount of different states the machine can take. As simple as this device is, it is believed to be as powerful as any other mechanical

[1] The time is based on the K computer in Japan with particles and age estimates from wikipedia.

procedure. And for any other device we know of, it is. This is known as the Church-Turing thesis.

The Turing machine is a very useful construction, but gets cumbersome very fast. Because of this, our computations will be described in the form of algorithms. A higher level description of step-by-step mechanical work, where each step would be possible to implement with a Turing machine. This gives us a better overview and intuition about our computations and also frees us from doing tedious and technical work when it is not needed. When we analyze the running time of our algorithms, we will assume standard operations like arithmetic operations, comparing numbers and reading numbers from a specific place in the memory to take constant amount of time. This could be exploited by encoding of data in big numbers and then do arithmetics. However we will not go down this path, but carefully enjoy the ease of such analysis without careless use.

Tractability

As we have seen, for some problems there is as far as we know, no fast algorithm that solves it. This has motivated the notion of complexity classes, classifying problems according to the resources their solution consume while being computed. In this text we will focus on time usage, as this often is the most limited one. But it also makes sense to study the use of other resources, like space. Two of the most widely known such classes are P and NP. The problems in P we can solve in time bounded by some polynomial of the input size. While the problems in NP have solutions that can be verified in polynomial time. As verifying a solution seems much easier than actually solving the problem, most scientists today believe that there are problems in NP that we cannot solve in polynomial time. However, there exists no proof of this. In fact, this problem is regarded so important that it is one of the so-called millennium problems, i.e. eight problems picked in end of the last millennium of great importance within mathematics. For each of these problems, a prize of one million dollars is awarded for a solution, by the Clay Institute.

The notion of NP-completeness was introduced[2] by Cook in 1971 [Coo71]. A class of problems capturing the hardest problems in NP. If one of these were to be solved in polynomial time, all problems in NP could be. The Cook-Levin theorem states that Boolean Satisfiability is NP-complete [GJ79]. And after this many problems have been proved to be NP-complete, by reducing already known NP-complete problems to them [Kar10, GJ79]. This work is important for several reasons, among them the fact that many of the natural, interesting problems are NP-complete. Polynomial time solvability tends to draw the line between the problems we can solve in general in a

[2] Although the term NP-complete did not appear until later.

feasible amount of time and those we cannot. Hence, if you get to know that a problem you are trying to solve is NP-complete, this will tell you something about the huge difficulties you will face if you try to obtain a polynomial time algorithm. It might not even be possible and thousands of researchers world wide have already tried to do the exact same thing.

Coping with intractability

Many of the interesting problems that we would like a computer to solve are NP-complete. So what do we do? The need to solve these problems does not go away by the fact that they are hard. And in fact, solutions are being computed for NP-complete problems every day. So how does this work? One option is to relax the requirements on the solution. Maybe you do not need an optimal solution; a somewhat good solution would be sufficient. This is the area of approximation algorithms. Another option occurs when you are interested in solutions only if they are small. Imagine that you have collected data, some of which are wrong. You want to know how much data you would have to remove to make it consistent. This is often only interesting if you can remove few data points, because if you have to remove a big portion of your data this reflects badly on all of the data. When this is the situation we tend to algorithms which are polynomial if we fix parts of the input, algorithms in XP or FPT. These topics will be discussed further in the thesis, both in Part I and in the results later on. A third option would be to realize that you do not need to solve the problem in its general formulation. If your problem is a graph problem, there might be some specific structure in your input. Your graph might be cycle free or without big cliques. Or maybe each vertex has a limited amount of edges incident to it as it is unlikely that hundreds of roads connect in the same intersection.

Graph parameters

This method of assuming extra structure on the input graph and try to exploit it can be generalized to the concept of graph parameters. A graph parameter gives us a measure of some property of a graph. For example, the cycle-free graphs which we often call trees, form the basis for one of the most famous graph parameters, namely treewidth. Treewidth gives us a measure of how much a graph resembles a tree. Treewidth of a graph is 1 if and only if the graph is a tree and grows bigger the further we move away from trees. Many problems that are hard to solve on general graphs, become polynomial time solvable on graphs of bounded treewidth. For the formal definition of treewidth we refer to Chapter 2.

Another graph parameter is bandwidth. If we line up all the vertices in a graph, the edge that is stretched over the largest number of vertices gives us the bandwidth of that line up. The bandwidth of a graph is then

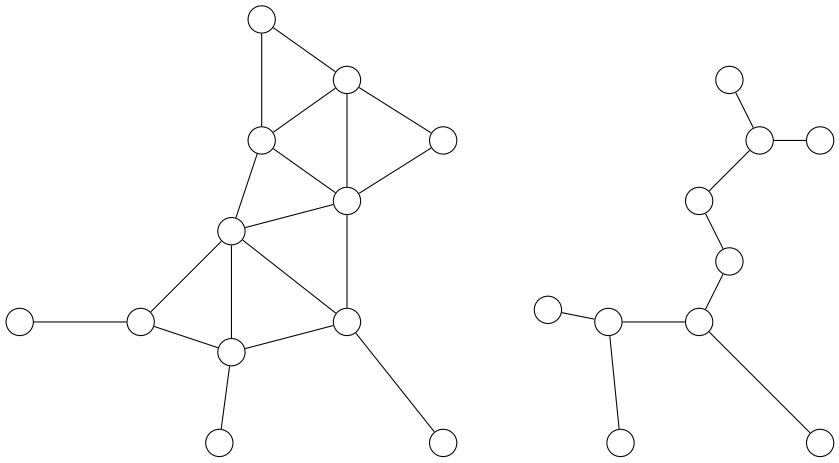


Figure 1.5: The graph on the left is not a tree, as it contains cycles. But, if we contract every triangle into a vertex and connect two triangles if they share an edge we end up with the tree on the right. As we only contract small objects, the graph on the left actually has treewidth 2.

the minimum bandwidth over all such possible ways to line up our graph. Bandwidth has many applications, but it is unfortunately hard to compute. It is NP-complete even on trees of maximum degree 3 [GGJK78], it is hard to approximate [Ung98, BKW97], and it does not admit a fixed parameter tractable algorithm [BFH94] unless the W-hierarchy collapses. One of the applications of bandwidth arises in sparse matrix computations. In the same way as we find it easier to calculate $1 \cdot 90131$ than $467 \cdot 193$, even though the result is the same, algorithms also prefer some representations over others. For a symmetric matrix A, one would like to find a permutation of its rows, and symmetrically its columns, such that in the permuted matrix all non-zero elements lie close to the diagonal. This is equivalent to the bandwidth problem and if one manages to find such a permutation, this can speed up matrix operations like multiplication, inversion and Gaussian elimination considerably [GHK$^+$09].

Many graph parameters naturally arise in various different applications. Pathwidth and treewidth, for example, can be independently defined from search games in graphs.

Graph searching

Imagine that we are the system administrators for a computer network. Occasionally computer malware manages to pass through our fine firewall. Then, between any two connected computers the malware can spread. When it is detected, a cleaning protocol is initiated. A computer is cleaned by disconnecting it from the network and executing cleaning software. After this

it is connected to the network again when we decide it to be. It is not a good idea to reconnect a computer if it is connected to an uncleaned part of the network, as it might immediately become infected again. Hence a set of computers separating the clean part from the infected part of the network must be disconnected.

As services from this computer network are highly requested, we want the number of disconnected computers to be as low as possible at any time. We call the maximum number of simultaneously disconnected computers during the protocol the protocol cost. We are to design such a virus searching and cleaning protocol for the network and our employers will be very happy if we manage to minimize the protocol cost. If G is the graph representing the computer network, any optimal solution with respect to the protocol cost requires exactly the pathwidth of $G + 1$ disconnected computers during the search [KP86]. The pathwidth of a graph is similar to treewidth, the difference being that we want our graph to decompose into a path instead of a tree. For many graph parameters, there is such a relation between the parameter and these kinds of games in graphs. In fact, if our virus is inert, meaning that it will only spread from a computer if that computer is about to get cleaned, the number of disconnected computers needed to ensure a full clean is related to the treewidth of the graph [DKT97]. See for yourself if you manage to search through the graph in Figure 1.5 with only three searchers, knowing that the virus is inert.

The cost considered above was the maximum number of disconnected computers at any time. Let us instead consider the sum of how long each of the computers are disconnected during the protocol as the cost, the total disconnection time. Then an optimal solution with respect to this cost relates to either the fill-in [FHT05] or the profile [FG00] of the network graph, depending on whether the virus is inert or not. In this spirit, one can also define the cost function, individual disconnection time, where we would like to minimize the maximum total time a single computer is disconnected during the protocol. For a standard virus this is equivalent to the bandwidth of the graph [RS83].

The first cost, the protocol cost, can be motivated from wanting the network to stay as responsive as possible by minimizing the number of computers disconnected at any time. The second one fits a network which is doing something continuously and the most important thing is how much time can be spent on computations while doing the cleaning. The third cost function can be motivated by each computer having certain tasks, or maybe they are in different locations, and the time each task or location do not have a computer available should be as small as possible while keeping the system somewhat fair.

One piece is missing in the picture and that is a solution that minimizes individual disconnection time when the virus is inert. This motivated the definition of treespan by Fomin et al. [FHT05]. It generalizes the well studied

notion of bandwidth from linear orderings to tree structures. This makes it an interesting field of study and hence the computational tractability of this parameter will be the topic of this thesis.

Note that when minimizing the protocol cost for both standard or inert viruses, the total disconnection time for both types of viruses or the individual disconnection time for a standard virus, one can prove that there is a monotone optimal protocol [FHT05], meaning that no computer will be disconnected twice. However, Dereniowski [Der09] proved monotone protocols to be non-optimal in the case of an inert virus when minimizing individual disconnection time.

The problems discussed above are referred to as node searching games in the literature. The name originates from the fact that we lift our cleaners between vertices in the graph. Other search games also exists, as edge searching, where the cleaners are slided along the edges cleaning the edge while moving. Another version is mixed searching, which is a combination where both lifting and sliding are permitted. All of these variations fits into the broader class of graph searching problems, which has been studied extensively [AG02, Par78, Pet82, MHG$^+$88, BS91, Mih10].

1.2 Outline of the thesis

In the rest of Part I we give the necessary background. In Chapter 2 we provide notation and the definitions of basic concepts used throughout the thesis. Chapter 3 provides an introduction to the complexity theory needed. The rest of the thesis is original work, unless otherwise is stated.

Part II introduces the graph parameter adjacencyspan and proves it to be equivalent to treespan. Furthermore we provide some structural results.

In Part III we provide numerous algorithmic results: In Chapter 5 we prove Treespan to be polynomial time solvable for every fixed k. In Chapter 6 we revise some work from Fomin et al. [FHT05] and answer one of their open problems regarding Treespan on trees of maximum degree higher than 3. In Chapter 7 we prove Treespan to be fixed parameter tractable and admitting a polynomial kernel when parameterized by both the vertex cover number of the input graph and the requested treespan. Furthermore, we provide a fixed parameter tractable algorithm for the case when the problem is only parameterized by the vertex cover number.

In Part IV give a conclusion of our work and provide open problems and a discussion around their possible solutions.

Chapter 2

General background

2.1 Notation and definitions

A *graph* $G = (V, E)$ consists of a set of *vertices* $V(G) = V$ and a set of *edges* $E(G) = E \subseteq V^2$. Throughout the text n will denote the number of vertices and m the number of edges in a graph. For readability we will denote the edge (u, v) by uv. A graph is *undirected* if given any edge uv in G then vu is also an edge in G and we will then consider uv and vu to be the same edge. A graph is *simple* if there is no vertex v such that $vv \in E$. All graphs in this text will be both undirected and simple unless otherwise is stated. Two vertices are *adjacent* if there is an edge between them, and in that case we call them neighbors. An edge is *incident* to a vertex v if v is one of its endpoints.

For a vertex v in a graph $G = (V, E)$, the *open neighborhood* of v is the set of all neighbors of v and is denoted $N_G(v) = \{u \mid vu \in E\}$. The *closed neighborhood* of v is the open neighborhood and v itself, denoted $N_G[v] = N_G(v) \cup \{v\}$. For a set of vertices $C \subseteq V$, we define $N_G[C] = \bigcup_{v \in C} N_G[v]$ and $N_G(C) = N_G[C] \setminus C$. The *degree* of v is the size of its open neighborhood, denoted $d_G(v) = |N(v)|$ and by $\Delta(G)$ we mean the maximum degree of any vertex in G. When there is no risk of ambiguity we will omit subscripts.

A *path* P in a graph G is a sequence of vertices $P = (p_1, \ldots, p_k)$ such that every pair of consecutive vertices in P are adjacencent in G. A *cycle* is a closed path, meaning that p_1 and p_k are adjacent. A path or cycle is said to be simple if it contains no vertex more than once. The equivalence relation "there is a path from u to v in G" partitions $V(G)$ into its *connected components*. We call a graph *connected* if it has only one connected component. The *distance* between two vertices u and v belonging to the same connected component in G, $\text{dist}_G(u, v)$, is $\min\{k - 1 \mid (p_1, \ldots, p_k)$ is a path in G from u to $v\}$, (the length of a shortest path between u and v). The *diameter* of a graph G, $\text{diam}(G)$, is the maximum shortest distance between any two vertices.

A *tree* is a connected graph containing no cycles. A *rooted tree* (T, r) is a tree T with a *root* $r \in V(T)$. For every vertex $v \in V(T) \setminus \{r\}$ we call the unique vertex in $N(v)$ closest to the root the *parent* of v, denoted $\rho(v)$. All other neighbors of v are called *children* of v. We say that u is an *ancestor* of v if there is a simple path from r to v containing u, and if u is an ancestor of v then v is a *descendant* of u. Vertices u and v have an *ancestor-descendant relationship* if u is an ancestor or descendant of v. By T_v we mean the subtree of (T, r) rooted in v.

For two graphs G and H we say that H is a *subgraph* of G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Furthermore H is an *induced subgraph* of G if $V(H) \subseteq V(G)$ and $E(H) = E(G) \cap V(H)^2$. The induced subgraph of G with vertex set $C \subseteq V(G)$ is denoted $G[C]$. For a set of vertices $V' \subseteq V(G)$ we define $G \setminus V'$ as $G[V(G) \setminus V']$ and for a set of edges $E' \subseteq E(G)$ we define $G \setminus E'$ as the graph $(V(G), E \setminus E')$.

We say that a graph is *complete* if every pair of vertices are adjacent and *edgeless* if there are no edges in the graph. A *clique* K in a graph G is a set of vertices such that $G[K]$ is complete and an *independent set* I is a set of vertices such that $G[I]$ is edgeless. A set S of vertices in $G = (V, E)$ is called a *vertex cover* if every edge in G is incident to a vertex of S. Observe that if S is a vertex cover in G then $G[V \setminus S]$ contains no edges, hence is $V \setminus S$ an independent set. The *vertex cover number* of a graph G is the smallest cardinality of a vertex cover of G.

A subset S of the vertices of the graph G is a *uv-separator* if u and v belong to different connected components in $G \setminus S$. S is a separator if it is a uv-separator for some u and v.

2.2 Graph decompositions and parameters

In this section we introduce several classical graph decompositions and related parameters, including tree decomposition, ordered tree decomposition, treewidth, bandwidth and the parameter we will study in this thesis, namely treespan. Each section will be supplemented by examples to give the reader an intuition of how the decompositions and parameters behave. However, this intuition should not be trusted blindly.

By Treewidth, Pathwidth, Bandwidth and Treespan we denote the problems of, given a graph G and an integer k, determining if G has treewidth, pathwidth, bandwidth or treespan at most k, respectively.

Tree decomposition and treewidth

A *tree decomposition* of a graph $G = (V, E)$ is a pair (X, T), where $T = (I, M)$ is a tree and $X = \{X_i \mid i \in I\}$ is a collection of subsets of V called *bags*, such that:

1. $\bigcup_{i \in I} X_i = V$,
2. if $uv \in E$, then there is a bag X_i such that both u and v are in X_i and
3. for every vertex $v \in V$, $\{i \in I \mid v \in X_i\}$ induces a connected subtree of T.

For a tree decomposition (X, T) of a graph G, we define the *width* of (X, T) as $\mathrm{w}(X, T) = \max\{|X_i| - 1 \mid X_i \in X\}$ and the *treewidth* of G as:

$$\mathrm{tw}(G) = \min\{\mathrm{w}(X, T) \mid (X, T) \text{ is a tree decomposition of } G\}.$$

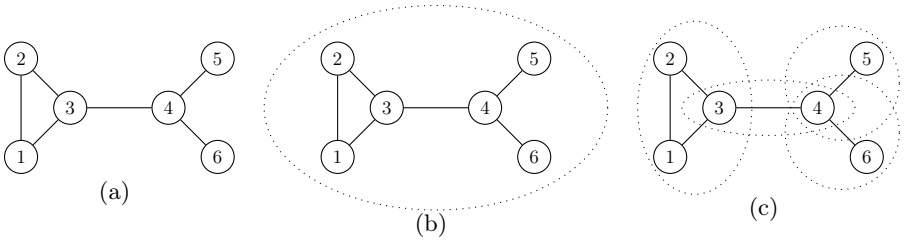


Figure 2.1: For the graph G in (a), (b) displays the trivial tree decomposition with width 5 and (c) is an optimal tree decomposition demonstrating G to have treewidth 2.

A *path decomposition* is a tree decomposition (X, T) such that T is a path and the *pathwidth* of a graph G is:

$$\mathrm{pw}(G) = \min\{\mathrm{w}(X, T) \mid (X, T) \text{ is a path decomposition of } G\}.$$

Bandwidth

A *linear ordering* σ_G of a graph G is a bijection from $V(G)$ to $\{1, 2, \ldots, n\}$. Let $\max\{\sigma_G(u) - \sigma_G(v) \mid uv \in E(G)\}$ be the *cost* of σ_G. The *bandwidth* of G, $\mathrm{bw}(G)$ is then the minimum cost over all linear orderings of G.

BANDWIDTH	
Input:	A graph G and an integer k.
Question:	Is $\mathrm{bw}(G) \leq k$?

Another definition of bandwidth is as follows. An *ordered path decomposition* is a path decomposition (X, P) of G such that if we enumerate the bags $X_1, X_2, \ldots, X_n$ from one leaf bag to the other, then $|X_1| = 1$ and for all $2 \leq i \leq n$, we have that $|X_i \setminus X_{i-1}| = 1$, meaning that exactly one new vertex is introduced in each bag. Let $\mathrm{l}(v)$ be the number of bags in (X, P) containing v and $\alpha(X, P) = \max\{\mathrm{l}(v) - 1 \mid v \in V(G)\}$. It is easy to show that $\mathrm{bw}(G)$ is the minimum $\alpha(X, P)$ over all ordered path decompositions (X, P) of G.

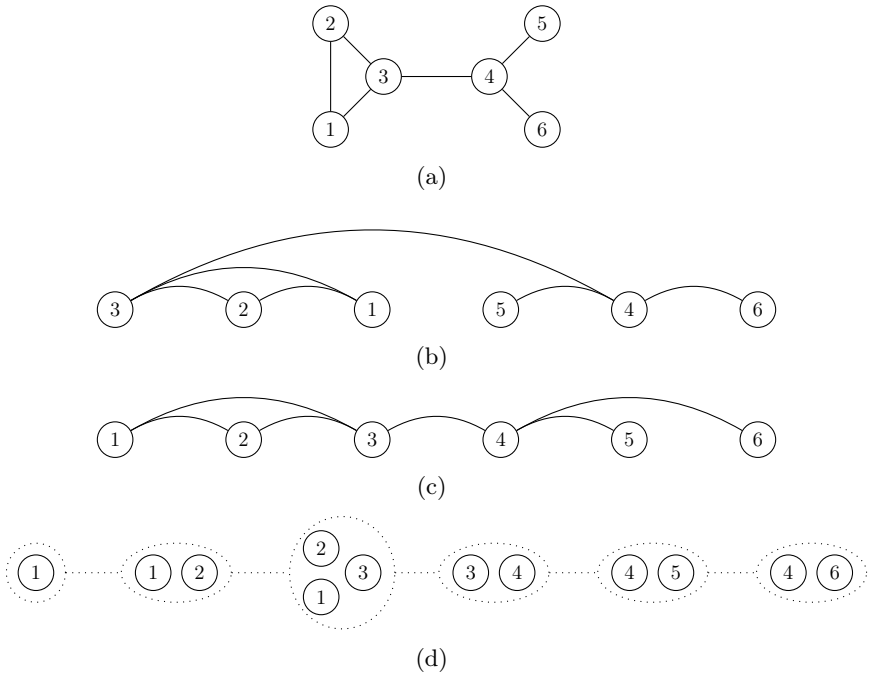


Figure 2.2: For the graph (a), (b) is a linear ordering of cost 4, (c) is an optimal linear ordering of cost 2 and (d) is an ordered path decomposition representation of (c).

Treespan

Fomin et al. [FHT05] extended bandwidth to ordered tree decompositions, thereby introducing the notion of *treespan*. An *ordered tree decomposition* (X, T, r) of a graph $G = (V, E)$ is a rooted tree decomposition with root $r \in I$ such that: $|X_r| = 1$ and for all $i, j \in I$ such that j is the parent of i it holds that $|X_i \setminus X_j| = 1$. We say that the bag closest to the root containing v *introduces* v in (X, T, r).

Let $G = (V, E)$ be a graph and (X, T, r) an ordered tree decomposition of G. Furthermore, let the *span* of a vertex v, denoted $\text{span}(v)$, be the number of bags containing v and $\text{ts}(X, T, r) = \max\{\text{span}(v) - 1 \mid v \in V\}$ (the maximal number of occurrences of a vertex). The treespan of G is then defined as

$$\text{ts}(G) = \min\{\text{ts}(X, T, r) \mid (X, T, r) \text{ is an ordered tree decomposition of } G\}.$$

TREESPAN	
Input:	A graph G and an integer k.
Question:	Is $\text{ts}(G) \leq k$?

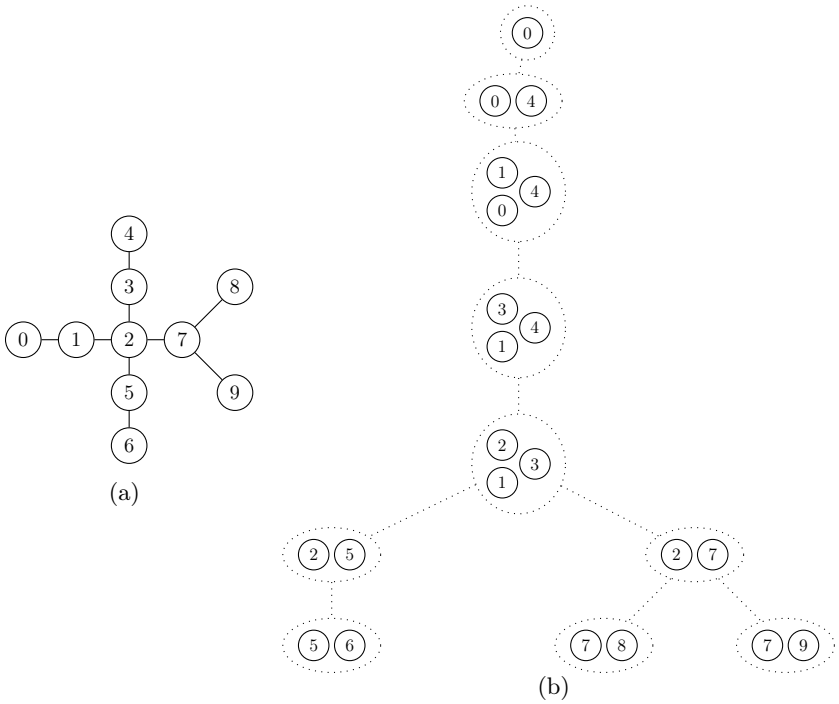


Figure 2.3: An optimal ordered tree decomposition, proving that $\text{ts}(H) = 2$. Note that that $\text{bw}(H) = 3$.

Note that any ordered path decomposition is an ordered tree decomposition, implying that $\text{ts}(G) \leq \text{bw}(G)$ for any graph G. The graph in Figure 2.2 is an example of a graph where $\text{ts}(G) = \text{bw}(G)$. Rautenbach [Rau05] proved that for a graph G and a subgraph H of G we have that $\text{ts}(H) \leq \text{ts}(G)$. From the thesis of Watnedal [Wat05] we have the following two bounds, $\text{ts}(G) \geq \lceil \Delta(G)/2 \rceil$ and $\text{ts}(G) \leq n - 1$. As the treespan of a graph G is equal to the maximum treespan of its connected components, we will assume from now on that G is connected.

2.3 Additional problems

For self containment we will give the definition of the other problems that will be mentioned during this thesis.

CLIQUE	
Input:	A graph G and an integer k.
Question:	Does G contain a clique of cardinality k?

INDEPENDENT SET	
Input:	A graph G and an integer k.
Question:	Does G contain an independent set of cardinality k?

GRAPH COLORABILITY	
Input:	A graph G and an integer k.
Question:	Is there a function f from $V(G)$ to $\{1, \ldots, k\}$ such that for every edge $uv \in E(G)$ it holds that $f(u) \neq f(v)$?

VERTEX COVER	
Input:	A graph G and an integer k.
Question:	Does G have a vertex cover of size at most k?

DIAGONAL RAMSEY	
Input:	A graph G and an integer k.
Question:	Does G contain a clique or independent set of size k?

Chapter 3

Complexity theory

3.1 Complexity

Our computations, as explained in Chapter 1, will be described and analyzed in form of algorithms. We will assume that standard arithmetic operations and relations like addition, multiplication and equality testing can be done in constant time. Also memory lookup at specific indices is assumed to be a constant time operation. By P we mean the problems which are computable in polynomial time by a deterministic Turing machine and by NP the problems which admit polynomial time verifiable certificates. The NP-complete problems are the hardest problems in NP with respect to polynomial time reductions. If it turns out that one of these problems is solvable in polynomial time then all problems in NP would be in P and hence $\mathsf{P} = \mathsf{NP}$. For more on this topic we refer to the classical book by Garey and Johnson [GJ79]. This chapter will be dedicated other complexity classes relevant for this thesis.

Time

Rather than giving the exact running time of an algorithm, we often use what is called *big O* notation to denote how algorithms behave asymptotically.

Definition 3.1. For two real functions defined on some subset of $\mathbb{N}$ we write $f(n) = O(g(n))$ if there exists n_0 and $c \in \mathbb{N}$ such that

$$|f(n)| \leq c|g(n)| \text{ for every } n \geq n_0.$$

While big O says that f should not grow asymptotically faster than g, we have another notation that says that f grows asymptotically slower than g, namely the *little o* notation.

Definition 3.2. For two real functions defined on some subset of $\mathbb{N}$ we write $f(n) = o(g(n))$ if for every positive ϵ there exists an $n_0 \in \mathbb{N}$ such that

$$|f(n)| \leq \epsilon|g(n)| \text{ for every } n \geq n_0.$$

Let $\text{TIME}(f(n))$ be the set of all problems solvable by a deterministic Turing machine in $O(f(n))$ time, where n is the size of the input. By this definition $\mathsf{P} = \bigcup_{c\in\mathbb{N}} \text{TIME}(n^c)$.

3.2 Exponential time solvability

We have already defined the class of problems solvable in polynomial time. But what about the problems outside of this class? Some of them are solvable in exponential time and those, are contained in the class EXP. More formally,

$$\mathsf{EXP} = \bigcup_{c\in\mathbb{N}} \text{TIME}(2^{O(n^c)}).$$

From the definitions and the Cook-Levin theorem, we know that $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXP}$. From the time hierarchy theorem, it follows that $\mathsf{P} \subset \mathsf{EXP}$ [Sip96]. Hence there are problems solvable in exponential time which are not solvable in polynomial time.

Recall the situation in Chapter 1, where we had a set of data points containing errors. We want to remove at most k data points such that the data becomes consistent. We can model the problem with a graph by representing each data point by a vertex and add an edge between a pair of vertices if the two data points are non-consistent. The problem is now reduced to remove k vertices from the graph such that there are no edges left. This is the classical problem Vertex Cover and it will follow us throughout this chapter in our examples.

Vertex Cover is trivially solvable in $O(2^n m)$ time and hence is in EXP. For each vertex v, we branch on the two possibilities of putting v in the vertex cover or not. When this has been done for all vertices one can check if the vertex cover is of size at most k and then iterate over the edges in the graph and ensure that each edge has at least one of its endpoints in the vertex cover. In total this takes $O(2^n m)$ time. For more on exponential algorithms we refer to the book by Fomin and Kratsch [FK10].

3.3 Parameterized problems

Many of the interesting problems we would like to solve are NP-complete. For these problems it is quite likely that we will have to settle with exponential time algorithms. But the exponential explosion in the running time does not remove the need to solve these problems. This motivated the study of parameterized problems. By considering not only the input size, but also some other parameter of the input we divide the instance space into slices, each containing instances of a fixed parameter value. We try to design algorithms which behave polynomially when restricting the input to a specific slice. And then, by assuming that the value of the parameter is not too

big, we make specific large instances tractable. We adopt the notions and definitions within parameterized complexity from the classical book in the field by Flum and Grohe [FG06].

Definition 3.3. For a finite alphabet Σ, we define

- a *parameterization* over Σ^* as a polynomial time computable function $\kappa : \Sigma^* \to \mathbb{N}$.
- a *parameterized problem* (over Σ) is a pair (Q, κ) consisting of a language $Q \subseteq \Sigma^*$ and a parameterization κ of Σ^*.

For an example let Q be the pairs (G, k) such that G is a graph with a vertex cover of size k and let the parameterization be $\kappa(G, k) = k$.

p-Vertex Cover	
Input:	A graph G and an integer k.
Parameter:	k.
Question:	Does G have a vertex cover of size at most k?

The p in the problem name will be used to differentiate the parameterized from the classical version. In the next two sections we will present two complexity classes which contain problems that get polynomial time solvable when we fix the value of the parameterization. For completeness we also give the definition of p-Treespan and p-Bandwidth.

p-Treespan	
Input:	A graph G and an integer k.
Parameter:	k.
Question:	Is $\text{ts}(G) \leq k$?

p-Bandwidth	
Input:	A graph G and an integer k.
Parameter:	k.
Question:	Is $\text{bw}(G) \leq k$?

3.4 Fixed k gives polynomial time algorithm

The class XP is the class of all the parameterized problems which admit polynomial time algorithms when we fix the value of the parameterization.

Definition 3.4. A parameterized problem (Q, κ) is in XP if there exists a function f, such that for every instance x we can decide if x is in Q in $O(|x|^{f(\kappa(x))})$ time.

Note that any algorithm with running time $O(g(\kappa(x)) \cdot |x|^{h(\kappa(x))})$ is an XP algorithm as $O(g(\kappa(x)) \cdot |x|^{h(\kappa(x))}) = O(|x|^{g(\kappa(x))+h(\kappa(x))}) = O(|x|^{f(\kappa(x))})$ for $f = g + h$. One can prove that p-Vertex Cover is in XP. Observe that if there is a solution of size less than k, we can trivially extend this to a solution of size k (assuming $k \leq n$, otherwise the instance is trivial). Hence we only search for a solution of size exactly k. There are $O(n^k)$ many such sets of k vertices in a graph. For each of these sets one can verify in $O(m)$ time whether the given set is a vertex cover by iterating over all the edges and checking that at least one of the endpoints is in the vertex cover. In total this gives us a running time of $O(n^k m) = O(n^{k+2})$ and hence it follows that p-Vertex Cover is in XP. In fact, most subset problems in NP are trivially in XP when parameterized by the size of the subset. Another example is p-Clique.

p-Clique	
Input:	A graph G and an integer k.
Parameter:	k.
Question:	Does G contain a clique of cardinality k?

In the same manner as for p-Vertex Cover, we try all possible subsets of size k. For each of these sets we check if it is a clique or not. This yields an $O(k^2 n^k)$ time algorithm for p-Clique.

It seems like XP is a big class of problems. Later in this thesis we will see that p-Treespan is in XP aswell, but this is not as trivial as our two examples in this section. There are also problems in NP which are not in XP under the assumption that $\mathsf{P} \neq \mathsf{NP}$. An example is Graph Colorability parameterized by the number of colors, as this problem is NP-complete even for $k = 3$ [Sto73].

3.5 Fixed parameter tractability

While the problems in XP are polynomial time solvable for every fixed k, the polynomials can get quite bad as k increases. Imagine running an $O(n^{100})$ algorithm on any reasonable instance. We would therefore like to achieve a running time, polynomial for any fixed k, such that the polynomial does not get worse when the value of the parameterization increases. This leads us to fixed parameter tractability.

Definition 3.5. A parameterized problem (Q, κ) is *fixed parameter tractable* if there exist a function f and a constant c such that every instance x is solvable in $O(f(\kappa(x)) \cdot |x|^c)$ time. The class FPT is the set of parameterized problems which admit fixed parameter tractable algorithms.

In this thesis the value of $\kappa(x)$ will always be a part of the input x. Hence we can see the input as a pair $x = (y, k)$ where $\kappa(x) = k$. A fixed

parameter tractable algorithm will then be an algorithm with running time $O(f(k) \cdot |y|^{O(1)})$. This is closer to the definition of Niedermeier [Nie06] and will be sufficient for this thesis. Note that a parametrized problem (Q, κ) solvable in $O(g(k) + |x|^{O(1)})$ for any instance x also is in FPT since

$$g(k) + |x|^c \leq (g(k) + |x|^c)^2 = g(k)^2 + 2g(k)|x|^c + |x|^{2c} \leq (g(k)^2 + 2g(k))|x|^{2c}$$

under some reasonable assumptions about $g(k)$, $|x|$ and c.

It is well-known that p-Vertex Cover is fixed parameter tractable [Nie06]. To see this, we observe that for every edge uv, at least one of u and v must be contained in any vertex cover. Algorithm 3.6 is based on this observation.

Algorithm 3.6 FPT algorithm for p-Vertex Cover

```
1: function hasVC(G, k)
2:     if G is edgeless then
3:         return true
4:     if k ≤ 0 then
5:         return false
6:     Let uv be some edge in G.
7:     return hasVC(G \ {v}, k − 1) or hasVC(G \ {u}, k − 1)
```

At each recursive step of the algorithm, checking if G is edgeless, finding uv and removing u and v can be done in $O(n)$ time with a reasonable graph representation. At each recursive call the parameter k decreases and as we stop if k reaches zero, the depth of our recursion is bounded by $k + 1$. At each level we branch into two, implying that the size of our branching tree is bounded by 2^{k+1}. Hence, our algorithm runs in $O(2^k n)$ time and it follows that p-Vertex Cover is in FPT.

3.6 The W-hierarchy

Recall that Graph Colorability is NP-complete for $k = 3$ and hence not believed to be in XP. The same argument applies for the class FPT. So how about the problems in XP? Are all of them in FPT or are there some problems which does not admit fixed parameter tractable algorithms? In fact, we do not know. But just as for P, we have problems that seem unlikely to belong to FPT. In fact we have a hierarchy of classes which are all supersets of FPT. We call it the W-hierarchy and it is displayed below.

$$\mathsf{FPT} \subseteq \mathsf{W}[1] \subseteq \mathsf{W}[2] \subseteq \cdots \subseteq \mathsf{XP}.$$

If a problem is complete for some class in the hierarchy it is considered unlikely that it belongs to any of its subclasses. p-Clique is known to be W[1]-complete [DF95] and hence not believed to admit a fixed parameter tractable algorithm.

3.7 Kernels

For years the computer science community has been using reduction heuristics, meaning that one in certain cases reduces the size of the input without changing the answer. But there has been no theoretical framework to analyze how these heuristics perform. Instead performance of implementations have been considered. Assume that for some NP-complete problem there is a set of reduction rules applicable on any instance x in polynomial time such that the new instance x' has a size bounded by $c|x|$ for some $c < 1$. Then by applying these rules $O(\log |x|)$ times, we would get an instance of constant size. Hence we could solve the problem in sub-exponential $O(x^{O(\log |x|)}) = O(2^{O(\log |x|)})$ time. As such a framework for NP-complete problems within traditional complexity would imply all problems in NP to be solvable in sub-exponential time, contradicting the exponential time hypothesis, it is considered unlikely to exist [Woe03].

The notion of parameterized problems provides us with such a framework, namely *kernels*. A kernel is a reduced instance of a parameterized problem of size bounded only by the parameterization of the original instance. Furthermore, it preserves the answer and should be computable in polynomial time.

Definition 3.7. Let (Q, κ) be a parameterized problem over a finite alphabet Σ. A function $K : \Sigma^* \to \Sigma^*$ is a *kernelization algorithm* or simply a *kernelization* of (Q, κ) if there is a computable function $h : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \Sigma^*$ and $x' = K(x)$ we have

$$(x \in Q \Leftrightarrow x' \in Q) \text{ and } |x'|, \kappa(x') \leq h(\kappa(x)).$$

Furthermore $K(x)$ should be computable in time polynomial in $|x| + \kappa(x)$. $K(x)$ is called the *kernel* of x (under K) and the function h denotes the size of the kernel. If h is a polynomial we say that the kernel is a *polynomial kernel*.

Before we give any examples of kernels we will present the folklore result of how FPT and the set of problems with kernels relate.

Lemma 3.8 (Folklore)**.** *A decidable parameterized problem is fixed parameter tractable if and only if it admits a kernel.*

Proof. Suppose the problem admits a kernel. Then there is a polynomial time kernelization algorithm which transforms the instance x into an instance x' of size bounded by $h(\kappa(x))$ for some function h. Any algorithm applied to the new instance will run in time $O(f(\kappa(x)))$ for some function f and hence the problem is fixed parameter tractable.

Now suppose the problem is fixed parameter tractable and hence solvable in $O(f(\kappa(x)) \cdot |x|^c)$ for some f and c. If $|x| \leq f(\kappa(x))$, the original instance x

is a kernel. Otherwise $|x| > f(\kappa(x))$, but then the time required to solve the problem is $O(f(\kappa(x)) \cdot |x|^c) = O(|x|^{c+1})$. Hence we can solve the problem in polynomial time and return an equivalent trivial instance as a kernel. This completes our proof. □

We revise the problem of Ramsey numbers from the introduction, stating a formal definition of the parameterized diagonal Ramsey problem. Recall that p-Clique and by this also p-Independent Set is W[1]-hard.

p-Diagonal Ramsey	
Input:	A graph G and an integer k.
Parameter:	k.
Question:	Does G contain a clique or independent set of size k?

By Ramsey [KAM30] we know that there is a function $f(k)$ such that if $|V(G)| \geq f(k)$ the answer is yes. From the proof we can obtain an upper bound on f, lets call it $\bar{f}$. By this we get a trivial kernelization algorithm. If the instance if bigger than $\bar{f}(k)$ the answer is yes and we return a trivial yes-instance, otherwise the instance is bounded by $\bar{f}$ and hence a kernel. As this kernel is very big, one often tends to be more interested in polynomial kernels. But by the paper on co-nondeterministic compositions by Kratsch [Kra12] we know that this problem does not admit a polynomial kernel unless $\mathsf{NP} \subseteq \mathsf{coNP/poly}$, which is regarded to be unlikely. We will spend the rest of the section giving the polynomial kernel for p-Vertex Cover by Buss [Nie06].

Rule 1: If a vertex v in G is isolated, let new instance be $(G \setminus \{v\}, k)$.

Rule 1 is sound as clearly no isolated vertex will cover any edge and hence it will not be in any optimal solution.

Rule 2: If a vertex v in G has degree at least $k+1$, let new instance be $(G \setminus \{v\}, k-1)$.

Observe that for every vertex v and vertex cover S, S must either contain v or $N(v)$, otherwise it will not cover all edges. Hence if the neighborhood of a vertex is too big to be included in the vertex cover, we will have to pick the vertex itself. It follows that Rule 2 is also sound. We now obtain a kernel for p-Vertex Cover with $k(k+1)$ vertices and k^2 edges. Assume that we have applied Rule 1 and 2 exhaustively on our instance. We then argue as following, any vertex has degree at most k and can therefore at most cover all edges incident to $k+1$ vertices, the vertex and all of its neighbors. As every vertex is incident to some edge and our vertex cover is of size at most k, this implies that if our graph contains more than $k(k+1)$ vertices the instance is

a no-instance. Furthermore any vertex can cover at most k edges and hence if the graph contains more than k^2 edges it is a no-instance. It follows that p-Vertex Cover admits a kernel with $k(k+1)$ vertices and k^2 edges. In fact, Chen et al. [CKJ01] proved that by applying the classical theorem of Nemhauser and Trotter one can obtain a kernel with at most $2k$ vertices and k^2 edges. Recently Dell and van Melkebeek [DvM10] proved the bound on edges to be tight, meaning that there is no kernel with $O(k^{2-\epsilon})$ edges for any $\epsilon > 0$ unless $\mathsf{NP} \subseteq \mathsf{coNP/poly}$. For more on the topic of parameterized complexity we refer to the books by Downey and Fellows [DF99], Flum and Grohe [FG06] and Niedermeier [Nie06].

Part II

A fresh perspective

Chapter 4

Adjacencyspan

In this chapter, we introduce a new tree structure called adjacency trees and a graph parameter called adjacencyspan which we prove to be equivalent to treespan. In addition we prove that we can considered a strict subset of the adjacency trees we call branched adjacency trees without losing all optimal solutions with respect to adjacencyspan. This will be one of the main tools for developing algorithm in the rest of the thesis.

4.1 Adjacency trees

Let (X, T, r) be an ordered tree decomposition of a graph $G = (V, E)$. Fomin et al. [FHT05] observed the following: the index set I of X is of size n, so without loss of generality we can assume that $I = V$. In addition we can index the bags such that $X_r = \{r\}$ and $X_v \setminus X_{\rho(v)} = \{v\}$. From now on we will assume the bags of an ordered tree decomposition to be indexed in such a way. They continued this observation by proving the following lemma.

Lemma 4.1 ([FHT05])**.** *Let T be a rooted tree on the vertex set of a graph $G = (V, E)$. There exists an ordered tree decomposition (X, T, r) of G if and only if for every edge $uv \in E$, u and v have an ancestor-descendant relationship in T.*

The reason for every pair of adjacent vertices in G to have an ancestor-descendant relationship in T is that otherwise two adjacent vertices could be introduced in two separate branches of the tree, implying that no bag contains them both, breaking the edge property of tree decompositions. We will now follow up by giving these trees a name.

Definition 4.2. Given a graph G, an *adjacency tree* (T, r) of G is a rooted tree on the vertices of G such that every adjacent pair of vertices in G have an ancestor-descendant relationship in T.

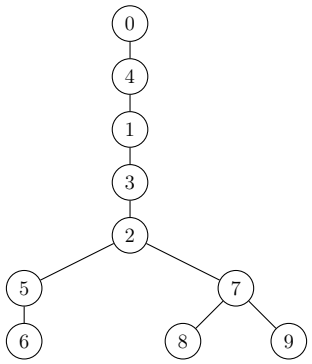
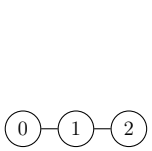


Figure 4.1: The corresponding adjacency tree of the graph in Figure 2.3.

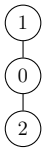
Adjacency trees and elimination trees

A reader with a background in chordal graphs and elimination trees might already have a feeling that adjacency trees and elimination trees somehow relate to each other. Fomin et al. [FHT05] introduced a parameter on elimination trees which they proved to be equivalent to treespan, so in fact there is already an established connection in the literature. In this section we will prove that the set of elimination trees of a graph actually is a proper subset of its set of adjacency trees. We give these results as structural results of independent interest.

We say that a graph G is *chordal* if there is no subset $C \subseteq V(G)$ such that $G[C]$ is a cycle of length at least 4. An *elimination order* of a graph $G = (V, E)$ is a mapping α from V to $\{1, \ldots, n\}$. If we iteratively remove the vertices of G in the order given by α and at each step turn the neighborhood of the vertex to be removed into a clique, the graph we obtain by adding all the edges you created in this process to the original graph G, denoted G^+_α, will be chordal [FG64]. The elimination tree of a certain elimination order is obtained by letting $\alpha^{-1}(n)$ be the root, and for every other vertex v let the parent of v be the vertex $u \in N_{G^+_\alpha}(v)$ with the lowest $\alpha^{-1}(u)$ larger than $\alpha^{-1}(v)$.



(a) A path P_3 on three vertices.



(b) A tree on the vertex set of P_3.

Figure 4.2: The tree on the right is an adjacency tree of the P_3 on the left.

The tree in Figure 4.2 is an adjacency tree of the graph, as every neighbor

in the graph have a ancestor-descendant relationship in the tree, but it is not an elimination tree. The reason being that vertex 1 is the root and hence was removed last, implying that $G_\alpha^+ = G$ and hence vertex 0 and vertex 2 is not connected in G_α^+, contradiction the fact that vertex 0 is the parent of vertex 2.

Proposition 4.3. *For a graph G every elimination tree (T, r) is an adjacency tree of G.*

Proof. For a contradiction, assume otherwise. Then there exists an edge $uv \in E(G)$ such that u and v does not have an ancestor-descendant relationship in (T, r). Without loss of generality, assume that $\alpha(u) < \alpha(v)$ and let $a = \rho(u)$ and $b = \rho(v)$. By definition we know that $\alpha(u) < \alpha(a)$ and $\alpha(v) < \alpha(b)$. If $a = b$ we get that $\alpha(u) < \alpha(v) < \alpha(b) = \alpha(a)$, which is a contradiction as this implies that v is preferred as the parent of u before a in the definition. Hence we can assume that $a \neq b$. We can also assume that a and v do not have an ancestor-descendant relationship, as if v is an ancestor of a it implies that u and v have an ancestor-descendant relationship and if a is an ancestor of v it implies that $\alpha(v) < \alpha(a)$, which contradicts the fact that a is the parent of u. Since u is removed before both a and v in the elimination order we get that $av \in E(G_\alpha^+)$. We iteratively apply this procedure $n + 1$ times, at each step continuing with a and v. As $\alpha(u) + \alpha(v)$ strictly decreases, we are moving up in the tree, but we will never get to the case where one of them is an ancestor of the other or they have the same parent. This gives us our contradiction and completes the proof. □

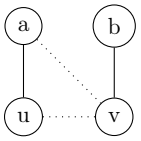


Figure 4.3: Illustration of last part of the proof of Proposition 4.3.

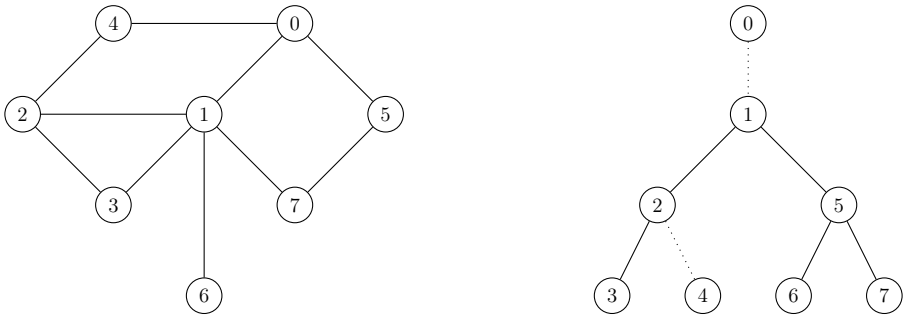
Consequently, we have proved that the set of elimination trees is a strict subset of the adjacency trees. For further reading on elimination trees we refer to Liu [Liu90].

4.2 Adjacencyspan

We will now introduce the parameter *adjacencyspan* and prove it to be equivalent to treespan. This definition will in some sense close the circle, as we went from the normal definition of Bandwidth to minimizing occurrences in ordered path decomposition, lifting this to the generalized problem on

trees, namely TREESPAN, for now to go back to orderings, but this time on trees. But before we do this we need a definition.

Definition 4.4. Let $G = (V, E)$ be a graph and (T, r) an adjacency tree of G. For a vertex $v \in V$ we define the *minimal neighborhood tree*, denoted T_v^m, to be the minimal subtree of T_v such that $N_G[v] \subseteq V(T_v^m)$.



(a) A graph G (b) An adjacency tree T of G

Figure 4.4: The solid edges in the tree display the minimal neighbor tree of vertex 1, T_1^m.

Definition 4.5. Let $G = (V, E)$ be a graph and (T, r) an adjacency tree of G. We then define the *adjacencyspan* of (T, r), denoted $\text{as}(T, r)$, as $\text{as}(T, r) = \max_{v \in V}(|T_v^m| - 1)$ and the adjacencyspan of G as

$$\text{as}(G) = \min \{\text{as}(T, r) \mid (T, r) \text{ is an adjacency tree of } G\}.$$

The adjacency tree in Figure 4.4 have adjacencyspan 5, and is not optimal. In Figure 4.5 we display an optimal adjacency tree with adjacencyspan 3.

We will now spend the rest of this section proving Theorem 4.7, which states that for any graph G, we have that $\text{as}(G) = \text{ts}(G)$. Before we begin the proof we need the following construction.

Construction 4.6. Given an adjacency tree (T, r) of a graph G we can construct X in the following manner such that (X, T, r) is an ordered tree decomposition of G:

1. Let $X_r = \{r\}$ and otherwise let
2. $X_v = \left(N_G\left[V\left(T_v\right)\right] \cap X_{\rho(v)}\right) \cup \{v\}$.

Informally we "drag a vertex of the parent bag $X_{\rho(v)}$ with us down to X_v" if it has a neighbor in the subtree T_v.

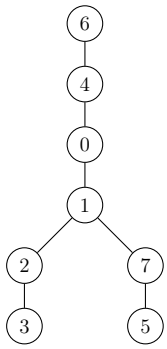


Figure 4.5: An optimal adjacency tree of the graph G in Figure 4.4 with adjacencyspan 3.

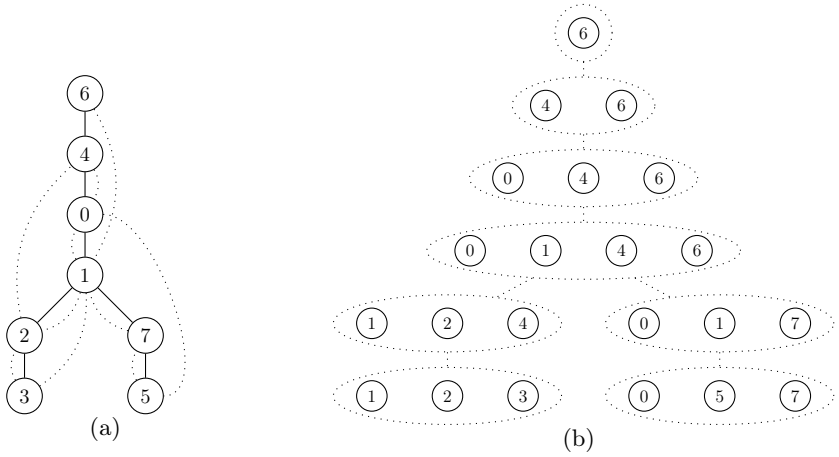


Figure 4.6: (a) is the adjacency tree from Figure 4.5, the dotted edges represent the edges from the graph. (b) is the ordered tree decomposition we get when applying Construction 4.6 to (b).

Theorem 4.7. *For every graph $G = (V, E)$ we have that* $\mathrm{ts}(G) = \mathrm{as}(G)$.

Proof. First we prove that $\mathrm{ts}(G) \leq \mathrm{as}(G)$. Let (T, r) be an adjacency tree of G, and let the bags X be constructed from (T, r) as described in Construction 4.6. Since a vertex v only gets added to a bag X_u if $v \in N_G[V(T_u)]$ and v is an ancestor of u, the set of bags that v is contained in is exactly the set of bags indexed by T_v^m. This implies that for every vertex v we get that $|T_v^m| = \mathrm{span}(v)$, and hence $\mathrm{as}(T, r) = \mathrm{ts}(X, T, r)$. It remains to prove that (X, T, r) is an ordered tree decomposition. Since X_v by construction contains v, we know that the vertex property holds for the decomposition. Let ab be an edge in E and assume without loss of generality that a is an ancestor of b in T. Then, on the path from a to b in T, $a \in \left(N_G\left[V\left(T_v\right)\right] \cap X_{\rho(v)}\right) \cup \{v\}$

for every v by induction and hence $a \in X_b$ and the edge property is maintained. As the bag X_v only contains vertices from $X_{\rho(v)}$ and v, it follows easily that (X, T, r) satisfies the connectivity property of tree decompositions, and hence it is a valid tree decomposition. For it to be an ordered tree decomposition, observe that by definition $|X_r| = 1$ and $|X_v \setminus X_{\rho(v)}| = 1$, and this completes this direction of the proof.

Now we prove that $\text{ts}(G) \geq \text{as}(G)$. Let (X, T, r) be an ordered tree decomposition of G. By Lemma 4.1 we know that (T, r) is an adjacency tree of G. Assume for a contradiction that $\text{ts}(G) < \text{as}(G)$. Then there is some vertex v, such that $|T_v^m| < \text{span}(v)$, hence there is a $u \in T_v^m$ such that $v \notin X_u$. But since $u \in T_v^m$ we know that $V(T_u)$ introduces a neighbor of v, and hence (X, T, r) either fails the edge property or the connectivity property of tree decompositions which gives us our contradiction and completes the proof. □

As a result of Theorem 4.7, for the rest of the thesis our work will be concentrated on ADJACENCYSPAN, as this gives a more manageable perspective both mathematically and algorithmically.

ADJACENCYSPAN	
Input:	A graph G and an integer k.
Question:	Is $\text{as}(G) \leq k$?

4.3 Branched adjacency trees

Definition 4.8. Let $G = (V, E)$ be a graph, and let (T, r) and (T', r') be adjacency trees of G. If there is a vertex $v \in V$ such that $T[V \setminus T_v \cup \{v\}] = T'[V \setminus T'_v \cup \{v\}]$ and $d_{T'}(v) > d_T(v)$ we say that (T', r') has a *higher branching* than (T, r). If there is no adjacency tree with a higher branching than (T, r) we say that (T, r) is a *branched adjacency tree.*

Lemma 4.9. *Let G be a graph and (T, r) an adjacency tree of G. Then (T, r) is a branched adjacency tree if and only if for every vertex $v \in V(G)$, the induced subgraph $G[V(T_v)]$ is connected.*

Proof. ($\Rightarrow$) Let (T, r) be a branched adjacency tree of G. Assume for a contradiction that there is a vertex v in G such that $G[V(T_v)]$ is not connected. Let $C \subseteq V$ be a connected component of $G[V(T_v)]$ not containing v. Let $T_{V \setminus C}$ be the tree obtained from $T[V \setminus C]$ by letting the parent of every vertex $v \in V \setminus C$ such that $\rho_T(v) \in C$ be the vertex closest to v on the simple path from v to r in T which is in $V \setminus C$. Construct T_C in the same way from $T[C]$. Note that T_C will be a tree by this procedure, as otherwise there are two different vertices $u, v \in C$ such that there is no ancestor of u or v in T

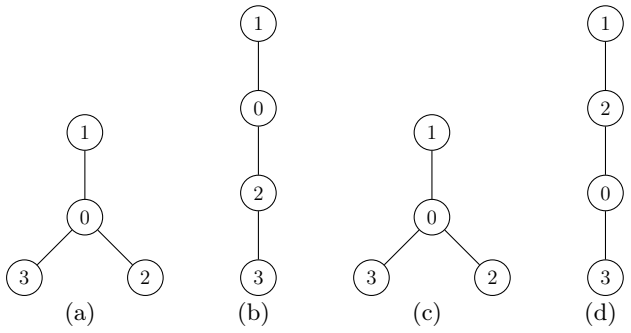


Figure 4.7: Let (a) be our graph and let (b), (c) and (d) be adjacency trees of (a). Observe that (c) has an higher branching than (b), while both (c) and (d) are branched adjacency trees.

contained in C. But then there can be no path between u and v in $G[V(T_v)]$ or otherwise the edge property or the ancestor-descendant property would not hold, hence u and v cannot be part of the same connected component in T_v, which is a contradiction. Let (T', r) be the union of T_C and $T_{V \setminus C}$ when adding an edge between $\rho_T(v)$ and the root in T_C. Note that v cannot be the root, as then G is not a connected graph and hence this construction is well-defined. The only ancestor-descendant relationships that changed from T to T' is the ones between vertices in C and $T_v \setminus C$, and as there are no edges between these to sets (T', r) is an adjacency tree. Let $u = \rho_T(v)$. Note that $d_{T'}(u) = d_T(u) + 1$ and $T[V \setminus T_u \cup \{u\}] = T'[V \setminus T'_u \cup \{u\}]$, which is a contradiction and completes this direction of the proof.

($\Leftarrow$) Suppose now that (T, r) is not a branched adjacency tree. Then there exists an adjacency tree (T', r) such that $T \setminus (V(T_v) \setminus \{v\}) = T' \setminus \{V(T'_v) \setminus \{v\}\}$ and $\mathrm{d}_{T'}(v) > \mathrm{d}_T(v)$ for some vertex v. Then there is two children x and y of v in T' and a child w in of v in T such that $\{x, y\} \subset V(T_w)$ by the pigeonhole principle. Since T' is an adjacency tree there are no edges from vertices in $V(T'_x)$ to vertices in $V(T'_y)$ in G and hence x and y must be connected through v in T'. This implies that T_w is not connected in G, which is a contradiction and completes the proof. □

Lemma 4.10. *Let (T, r) be an adjacency tree of a graph G. Then there exists a branched adjacency tree (T', r) such that* $\mathrm{as}(T', r) \leq \mathrm{as}(T, r)$.

Proof. Let v be a vertex in T such that its degree can be increased in the way described in the definition. Apply the algorithm in the proof of Lemma 4.9 to increase the degree of v by one and call this new tree T'. Observe that for any pair of vertices x, y in G it is not the case that x is an ancestor of y in T and a descendant of y in T'. This implies that $|T'^m_u| \leq |T^m_u|$ for all u, and hence $\mathrm{as}(T', r) \leq \mathrm{as}(T, r)$. Continue this process until T' is a branched adjacency tree. Clearly $\mathrm{as}(T', r) \leq \mathrm{as}(T, r)$. □

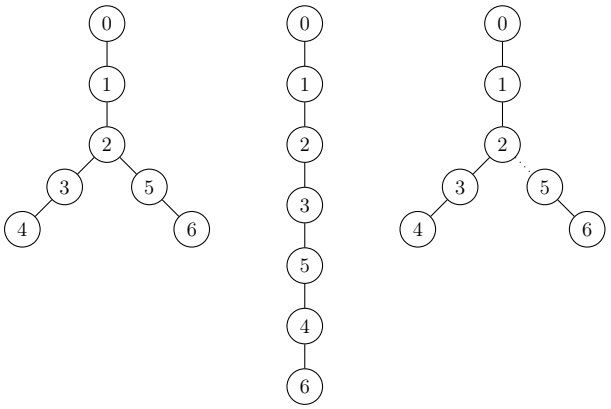


Figure 4.8: Illustration of the right direction of the proof of Lemma 4.9. G is on the left, in the middle is T and on the right is T_C and $T_{V\setminus C}$. The dotted edge illustrates the edge added to get T'.

By an *optimal adjacency tree* we mean an adjacency tree which is optimal with respect to adjacencyspan. The next theorem follows directly from Lemma 4.10.

Theorem 4.11. *For every graph G there exists a branched adjacency tree (T, r) which is optimal.*

In the next part of the thesis, we will use the results of this chapter to give algorithms for Treespan via Adjacencyspan.

Part III

Algorithms

Chapter 5

Adjacencyspan is in XP

As Fomin et al. [FHT05] proved Treespan to be NP-complete, we cannot hope to find a polynomial time algorithm for general input under the assumption that P $\neq$ NP. In this section we will present a dynamic programming algorithm solving Adjacencyspan in polynomial time for every fixed k, proving that p-Adjacencyspan is in XP. Saxe [Sax80] presented a dynamic programming algorithm for solving Bandwidth in $O(f(k)n^{k+1})$ time, building the solutions from left to right using the k previous vertices as the states. Our algorithm will similarly use the previously inserted vertices as states, but it will differ in several key aspects. While Saxe keeps track of not only the k previous vertices, but also which vertices in their neighborhood which are already inserted, we prove that we can manage with only these k vertices and in some cases one additional, specific vertex. This shrinks a factor of the running time from 4^k to $2k + 1$ and keeps the algorithm cleaner. Also, since we are building trees, there is no easy way to know when to branch the tree without the notion of branched adjacency trees, Lemma 4.9 and Theorem 4.11, a problem not appearing in linear orderings. Furthermore, a vertex contributes to the span in a much more involved way in a tree than in an linear ordering. To handle this we divide the available span among the branches in different ways and try to build a combined solution by dynamic programming. It is even non-trivial that the algorithm builds a valid adjacency tree. Before we continue, we give the definition of p-Adjacencyspan:

p-Adjacencyspan	
Input:	A graph G and an integer k.
Parameter:	k.
Question:	Is $\text{as}(G) \leq k$?

Preparation

Definition 5.1. Let G be a graph and (T, r) an adjacency tree of some induced subgraph H of G. We say that an adjacency tree (T', r) of G is a *k-extension* of (T, r) into G if the following holds:

- (T, r) is an induced subtree of (T', r) and
- (T', r) has adjacencyspan at most k.

Furthermore we say that (T, r) is *k-feasible* in G if there exists a k-extension of (T, r) into G.

Definition 5.2. Let (T, r) be an adjacency tree of a graph G. For a vertex v in T let the *k-root path* of v, denoted $R_k(v, T, r) = (v_1 = v, \ldots, v_k)$, be the k first vertices on the simple path from v to r in T. If the distance between v and r is less then k, we take all the vertices on the path as the k-root path. Furthermore, if it is clear from the context which adjacency tree is considered we will use the simpler notation $R_k(v)$.

In the definition of a root path, we considered it as a tuple. During this chapter there will be several occasions where we would like to consider the vertices in the root path as a set. We will then apply the set notation on the tuple and it should be interpreted as the set notation on the set of vertices within the tuple.

Lemma 5.3. *Let G be a graph and (T, r) a branched adjacency tree of G such that* $\mathrm{as}(T, r) \leq k$. *Let v be a vertex in G and*

$$R = \begin{cases} R_{k+1}(v) & \text{if } v \text{ and } v_{k+1} \text{ are adjacent in } G \text{ and} \\ R_k(v) & \text{otherwise.} \end{cases}$$

Let $H_1, \ldots, H_l$ be the connected components of $G \setminus R$ such that $H_i \cap N_G(v)$ is non-empty, then $V(T_v) = V(H_1) \cup \cdots \cup V(H_l) \cup \{v\}$.

Proof. ($\subseteq$) Let u be a vertex in $V(T_v) \setminus \{v\}$ and let H be the connected component in $G \setminus R$ containing u. Assume for a contradiction that $H \neq H_i$ for every i, which implies that $H \cap N_G(v)$ is empty. Then u and v are not connected in $G[V(T_v)]$ and hence $G[V(T_v)]$ is not connected, which is a contradiction to (T, r) being a branched adjacency tree by Lemma 4.9.

($\supseteq$) Let u be a vertex in H_i for some i in $[1, \ldots, l]$ and assume for a contradiction that u is not in T_v. By the definition of H_i there is a vertex $u' \in H_i \cap N_G(v)$. As u' and v are neighbors they must have an ancestor-descendant relationship in (T, r). Assume u' to be an ancestor of v. As u' cannot be in R and u' is a neighbor of v we know that the distance between u' and v in T is at least $k+1$ by the definition of R, which is a contradiction to $\mathrm{as}(T, r) \leq k$.

If u' is a descendant of v we consider a simple path from u' to u within H_i. Start in u' and follow this path as long as it stays inside $V(T_v)$. Since u is not in $V(T_v)$ it must at some point leave this set. Let p_i be the last vertex in $V(T_v)$ and p_{i+1} the next vertex not in $V(T_v)$ on the path. Since p_i and p_{i+1} are neighbors, p_{i+1} must be an ancestor of p_i and hence also v, since $p_{i+1} \notin V(T_v)$. Since p_{i+1} is in H_i it cannot be in R and hence the distance between p_{i+1} and v is at least k. Note that as we considered a simple path, $v \neq p_i$. It follows that p_i is a descendant of v and hence the distance between p_i and p_{i+1} is at least $k+1$, contradicting $\text{as}(T, r) \leq k$ and hence our proof is complete. □

Let $\mathcal{S}_k$ be the set of all functions from $[1, \ldots, k]$ to $[0, \ldots, k+1]$ and let $+$ denote componentwise addition over $\mathcal{S}_k$. Furthermore let $<_0$ be the binary relation over $\mathcal{S}_k$ defined as

$$\{(f, g) \in \mathcal{S}_k^2 \mid \forall i \in [1, \ldots, k], f(i) < g(i) \vee f(i) = g(i) = 0\}.$$

For a vertex v we will use functions in $\mathcal{S}_k$ to indicate how big T_v can be in each of the minimal neighborhood trees of the vertices in R. The reason we also allow $g(i) = f(i) = 0$ is that if there is no size to divide between the subtrees than it should be fine for the children to all consume 0.

Algorithm

We will describe our algorithm in a recursive fashion, and then, to obtain our running time we will apply memoization on the function Feasible. Note that we can find the connected components of G by a depth first search and at the same time check that $\Delta(G) \leq 2k$ in $O(kn)$ time [Sax80]. If G is not connected, Algorithm 5.4 should be applied on each of the connected components, and the answer for G is yes if and only if it is yes for each of the components.

- Feasible: Takes a graph G, an integer k, a root path R and a size function s and returns true if and only if there is a k-extension of the so far constructed adjacency tree T which also contains $V(T_v)$.
- Neighbor-Components: Takes a graph G and a set $R = (v, \ldots)$ as input and returns the connected components of $G \setminus R$ having a non-empty intersection with $N_G(v)$ (as described in Lemma 5.3).
- Valid: Takes a graph G, a root path $R = (v = v_1, \ldots, v_l)$, a size function $s \in \mathcal{S}_k$, a vertex u and a component H such that $u \in H$ and verifies that for every $v_i \in N_G(u) \cap R$ we have that $s(i) \geq 1$ and that $(N_G(v_k) \cap H) \setminus \{u\} = \emptyset$ (ensures that no vertex is inserted which makes the minimal neighborhood tree of a vertex bigger than it should be and that when a vertex is of distance $k+1$ from u and hence has no more span left all of its neighbors should be already inserted).

- ADD: Takes a root path R, a vertex u and a graph G as input. Creates a copy R' of R and adds u to the beginning of R'. If $|R'| = k + 2$, we remove the last element of R'. After this, if $|R'| = k + 1$ and u is not a neighbor of the last element, we remove the last element of R'. Then we return R' (We add u to the front of R, and turns it into the same kind of R as in Lemma 5.3).
- SHIFT(s, c): Takes a function $s \in S_k$ and a constant c. Returns s', where $s'(i) = s(i-1)$ for $i > 1$ and $s'(1) = c$.

Algorithm 5.4 Branch and build.

Input: a graph G and an integer k
Output: **true** if and only if as$(G) \leq k$

```
1: for all r ∈ V(G) do
2:     if FEASIBLE(G, k, (r), (k, 0, ..., 0)) then
3:         return true
4: return false

5: function FEASIBLE(G, k, R, s)
6:     H ← NEIGHBOR-COMPONENTS(G, R)
7:     if H = ∅ then
8:         return true
9:     S_now ← {the zero-function in S_k}
10:    for all H ∈ H do
11:        S_prev ← S_now
12:        S_now ← ∅
13:        for all s_p ∈ S_prev and s_n ∈ S_k such that s_p + s_n <_0 s do
14:            for all u ∈ H such that VALID(G, R, s_n, u, H) do
15:                if FEASIBLE(k, G, ADD(R, u, G), SHIFT(s_n, k)) then
16:                    S_now ← S_now ∪ {s_p + s_n}
17:    return S_now ≠ ∅
```

The algorithm tries all possible roots and inputs them to FEASIBLE one after another. In FEASIBLE we then retrieve the neighbor connected components and solve them separately for each size function and each new vertex to insert, before we try to combine them with a dynamic programming algorithm. Note that the algorithm easily can be changed such that it stores the adjacency tree it constructs and hence becomes a constructive algorithm.

Correctness

We start the program of proving correctness by a lemma stating that the solutions constructed by the algorithm satisfies the same kind of property as

we proved branched adjacency trees to do in Lemma 5.3.

Lemma 5.5. *Let H be a connected component found at some point by* Neighbor-components *in Algorithm 5.4. If $v \in H$ is inserted into the solution by the algorithm and $H_1, \ldots, H_l$ is the connected components found at the next recursion step after inserting v, then $H = H_1 \cup \cdots \cup H_l \cup \{v\}$.*

Proof. Let $(v, v_2, \ldots, r)$ be the vertices on the path from v to r in the so far constructed solution by Algorithm 5.4. Let R be the root path considered when H is found and R' the new root path after v is added. Recall that H is obtained from $G \setminus R$ and $H_1, \ldots, H_l$ from $G \setminus R'$.

Let $H'_1, \ldots, H'_{l'}$ be the connected components intersecting $N_G(v)$ in $G \setminus \{v, v_2, \ldots, r\}$. Since Valid ensures that any vertex v_i for $i \geq k+2$ does not have any neighbors in H we obtain the same connected components if we consider $G \setminus \{v, \ldots, v_{k+1}\}$. Observe that $\{v_2, \ldots, v_{k+1}\} \subseteq R$ and hence we also obtain the same components by considering $G \setminus (R \cup \{v\})$. Let u be a vertex in $H \setminus \{v\}$. We know that there is a simple path $P = (u, \ldots, p, v)$ from u to v in H. When we remove v from $G \setminus R$ to obtain $G \setminus (R \cup \{v\})$, p clearly remains in the same connected component as u. This connected component intersects $N_G(v)$ in p and is hence the component H'_i for some i. This implies that $u \in H'_i$ and it follows that $H \subseteq H'_1 \cup \cdots \cup H'_{l'} \cup \{v\}$. For the other direction let u be a vertex in H'_i some i and let p be a vertex in the intersection between H'_i and $N_G(v)$. Since u is connected to p in $G \setminus (R \cup \{v\})$, u is also connected to p in $G \setminus R$, implying that u is connected to v in $G \setminus R$ as p is a neighbor of v. Hence $u \in H$ and $H \supseteq H'_1 \cup \cdots \cup H'_{l'} \cup \{v\}$. It follows that $H = H'_1 \cup \cdots \cup H'_{l'} \cup \{v\}$.

Finally observe that v_{k+1} only affects the connected components we obtain if it is connected to v, as Valid ensures that it has no neighbors in $H \setminus \{v\}$. In this case $v_{k+1} \in R'$ and hence we obtain the same connected components $H'_1, \ldots, H'_{l'}$ intersecting $N_G(v)$ in $G \setminus R'$ as in $G \setminus (R \cup \{v\})$. Note that this is the definition of $H_1, \ldots, H_l$ and hence these components are equivalent to $H'_1, \ldots, H'_{l'}$ and our proof is complete. □

Lemma 5.6. *Every solution the algorithm constructs is an adjacency tree.*

Proof. Clearly the algorithm constructs a rooted tree if it terminates. It remains to prove that the solution contains every vertex in G exactly once and that is satisfies the ancestor-descendant property. Note that if the algorithm inserts every vertex at most once, it will follow that the algorithm terminates.

When the constructed tree is empty before the algorithm inserts a root, every vertex is in the same component. Let u be a vertex in a component H at some recursion step and let $v \neq u$ be the vertex inserted at this step. It then follows from Lemma 5.5 that there is a connected component H' such that $N_G(v) \cap H' \neq \emptyset$ at the next recursion step such that $u \in H' \subset H$.

Since H' is strictly contained in H it follows by induction that the algorithm inserts u at some step if it manages to construct a solution and hence every vertex will be inserted.

Since $H_1, \ldots, H_l$ are connected components they are trivially pairwise disjoint. Hence the vertices in $H \setminus \{v\}$ are partitioned among the branches, and in branch i only vertices in H_i are inserted. It follows that no vertex is inserted twice.

Let u and v be two adjacent vertices in the same connected component H at some recursion step. If none of the vertices are added to the solution at this step, they will trivially stay in the same component. Otherwise, assume without loss of generality that v is added. Since u also is contained in H it follows from Lemma 5.5 that it will be contained in a strictly smaller component at the next step. It follows by induction that u will be inserted as a descendant of v and satisfy the ancestor-descendant property and our proof is complete. □

Lemma 5.7. *Every adjacency tree the algorithm constructs has adjacencyspan at most k.*

Proof. Let v be any vertex in the solution (T, r). We will prove that for any vertex $u \in T_v^m$ it holds that $|T_u \cap T_v^m| \leq s_u(v)$, where $s_u(v)$ is the allowed size of T_u in T_v^m by the algorithm. If u is a leaf in T_v^m, then we know from VALID that $s_u(v) \geq 1$ and hence $|T_u \cap T_v^m = \{u\}| = 1 \leq s_u(v)$. For our induction step let $w_1, \ldots, w_l$ be the children of u in T_m^v and let $s_{w_1}(v), \ldots, s_{w_l}(v)$ denote how $s_u(v)$ was distributed among u's children. Assume that $|T_{w_i} \cap T_v^m| \leq s_{w_i}(v)$ for every i and remember that $s_{w_1} + \ldots s_{w_l} <_0 s$ from the algorithm. Note that as $w_i \in T_m^v$ for every i we know that $s_{w_i}(v) \neq 0$ and hence $s_{w_1}(v) + \cdots + s_{w_l}(v) < s_u(v)$. It follows that:

$$\begin{aligned} |T_u \cap T_v^m| &= |(T_{w_1} \cup \cdots \cup T_{w_l} \cup \{u\}) \cap T_v^m| \\ &= |T_{w_1} \cap T_v^m| + \cdots + |T_{w_l} \cap T_v^m| + 1 \\ &\leq s_{w_1}(v) + \cdots + s_{w_l}(v) + 1 \\ &\leq s_u(v) \end{aligned}$$

This completes our induction and hence $|T_v \cap T_v^m| = |T_v^m| \leq s_v(v) = k + 1$ and our proof is complete. □

Lemma 5.8. *If a graph G has adjacencyspan at most k, then the algorithm will return true.*

Proof. Assume there is a branched adjacency tree (T, r) with adjacencyspan at most k. Then at some point the algorithm will try r as a root. Assume that for the vertex v the path from v to r is identical in the algorithm and in (T, r). Then FEASIBLE will find the exact same components to build the subtrees from as in (T, r) by Lemma 5.3. And there is a step where the algorithm

will distribute the sizes exactly as in (T, r) as it is a valid distribution and for each of the components it will then try the same vertices as the children of v as in (T, r) and then for the children of v the path to the root equals the on in (T, r) and this completes the induction and our proof. □

Lemma 5.9. *Algorithm 5.4 returns true if and only if the input graph G has adjacencyspan at most k.*

Proof. This follows directly from Lemmas 5.6, 5.7 and 5.8. □

Running time analysis

Theorem 5.10. *The problem* Adjacencyspan *is solvable in $O(k^{3k+3}n^{k+1})$ time.*

Proof. First we find the connected components of G and verify that $\Delta(G) \leq 2k$ in $O(kn)$ time. Note that the worst case scenario for Algorithm 5.4 is if the graph is connected and hence it will be analyzed accordingly. We apply memoization to the function Feasible. This implies that the running time of our algorithm will be $O(nk + f(n,k)g(n,k))$ where $f(n,k)$ is the number of different inputs Feasible can get and $g(n,k)$ is the cost of the work done in Feasible.

Each of the elements in R is a vertex in G and R is of length at most $k+1$. Considering the first k elements of R there are $O(n^k)$ different possibilities. As the $k+1$th element only appears if it is a neighbor of the first element in the root path, there is at most $2k+1$ different options for this last element, either it is one of the $2k$ neighbors or it is not included. Hence there are $O(kn^k)$ different root paths to consider. Since there are $O(k^k)$ different size functions and G and k are invariant through the algorithm we get that $f(n,k) = O(k^{k+1}n^k)$.

The complexity of Neighbor-Components is $O(n+m) = O(kn)$ by a standard depth first search, while both Valid, Add and Shift are of complexity $O(k)$. The dynamic programming procedure combining solutions in the different subtrees takes $O(|\mathcal{H}| \cdot k^k \cdot k^k \cdot n \cdot k)$ time, as there are $|\mathcal{H}|$ many components, $O(k^k \cdot k^k)$ many choices for values of s_p and s_n, at most n elements in H and then we apply Valid, Add and Shift which contributes with a factor $O(k)$. Observe that the size of $\mathcal{H}$ is bounded by the degree of the first vertex in R, $2k$. It follows that $g(n,k) = O(nk+2k\cdot k^k\cdot k^k\cdot n\cdot k) = O(k^{2k+2}n)$. And hence the algorithm runs in $O(nk + f(n,k)g(n,k)) = O(nk + k^{k+1}n^k \cdot k^{2k+2}n) = O(k^{3k+3}n^{k+1})$ time. □

We end this chapter by giving some immediate results of Theorems 4.7 and 5.10.

Theorem 5.11. *The problem* Treespan *is solvable in $O(k^{3k+3}n^{k+1})$ time.*

Corollary 5.12. *The problem* p-Adjacencyspan *is in* XP.

Corollary 5.13. *The problem* p-Treespan *is in* XP.

It follows from the definition of XP and Corollies 5.12 and 5.13 that both Adjacencyspan and Treespan is solvable in polynomial time for every fixed values of k.

Chapter 6

Trees

The goal of this chapter is to solve ADJACENCYSPAN on trees of bounded maximum degree in polynomial time. First we will revise the work of Fomin et al. [FHT05], who proved TREESPAN, and hence ADJACENCYSPAN, to be solvable in polynomial time for trees of maximum degree 3. In their paper they stated the problem of deciding TREESPAN for trees of higher degree as their third and final open problem. Note that BANDWIDTH is NP-complete on trees of maximum degree 3 [GGJK78].

6.1 Trees of degree at most 3

Lemma 6.1 ([Wat05])**.** *A connected graph G has treespan 1 if and only if G is a path.*

If a connected graph has adjacencyspan 1, each vertex v has at most one neighbor as a descendant and this vertex must be a child of v. It follows easily that G must be a path. And clearly if G is a path, this path with one of the degree 1 vertices as a root, will be a valid adjacency tree of adjacencyspan 1. It follows that a connected graph has adjacencyspan, and hence treespan, 1 if and only if the graph is a path.

Lemma 6.2 ([Wat05])**.** *A tree of bounded maximum degree $d \geq 2$ has treespan at most $d - 1$.*

Take any leaf in the tree as a root and let the rest of the tree hang below. The rooted tree we obtain will be a valid adjacency tree. Every vertex has every neighbor either as a child or as its parent. The maximum degree d implies that the adjacencyspan is $d - 1$ for this composition and hence at most $d - 1$. It follows that a tree of bounded maximum degree d has adjacencyspan, and hence treespan, at most $d - 1$.

Lemma 6.3 ([FHT05],[Wat05])**.** *If G is a tree of maximum degree at most 3, then* TREESPAN *is polynomial time solvable.*

We know from Lemma 6.2 that G has treespan at most 2. From Lemma 6.1 we know that G has treespan 1 only if G is a path. Hence we can in linear time decide whether the graph has treespan 1 and otherwise it is 2.

6.2 Trees of bounded degree

Fomin et al. raised the question of whether TREESPAN can be computed in polynomial time for trees of maximum degree higher than 3. In his master thesis, Watnedahl [Wat05] tried to solve the problem for trees of maximum degree 4. He described 8 illegal subgraphs for such trees, but was never able to complete the set and hence determine the adjacencyspan of a tree of maximum degree at most 4. In this section we will prove that for any fixed d, ADJACENCYSPAN is computable in polynomial time for trees of bounded maximum degree d.

Theorem 6.4. *Given a tree G of bounded maximum degree d and an integer k it is decidable in $O(n^{d-1})$ time whether* $\text{as}(G) \leq k$.

Proof. If $k \geq d-1$, we know from Lemma 6.2 that the answer is yes. Assume that $k \leq d-2$. Then we from Theorem 5.10 that ADJACENCYSPAN is decidable in $O(f(k)n^{k+1}) = O(f(d-2)n^{d-2+1}) = O(n^{d-1})$. □

The running time in Theorem 6.4 seems to be optimal, as for the case $d = 2$ it runs in $O(n)$ time. To be able to test if a graph of bounded maximum degree is a path faster than in linear time is unlikely. The next theorem follows directly from Theorems 4.7 and 6.4.

Theorem 6.5. *Given a tree G of bounded maximum degree d and an integer k it is decidable in $O(n^{d-1})$ time if* $\text{ts}(G) \leq k$.

One might wonder if this technique is applicable to graphs of bounded degree as well. It turns out that it is not. Observe that if a bag in an ordered tree decomposition is of size k, then there must be one vertex in the bag which has appeared in k bags since we are only allowed to introduce one new vertex in each bag. This implies that $\text{tw}(G) \leq \text{ts}(G)$ for any graph G. If the graph G is a big grid, it is known to have big treewidth [Die05] and hence big treespan. But if G is a grid, $\Delta(G) = 4$ and hence graphs of bounded degree do not have bounded treespan.

6.3 No locality

For a graph of low adjacencyspan, we know by definition that there exists an adjacency tree such that if two vertices are adjacent in G they are not too far apart in the adjacency tree. But does there exist a similar relation the other way around? Is there always an optimal adjacency tree such that if two

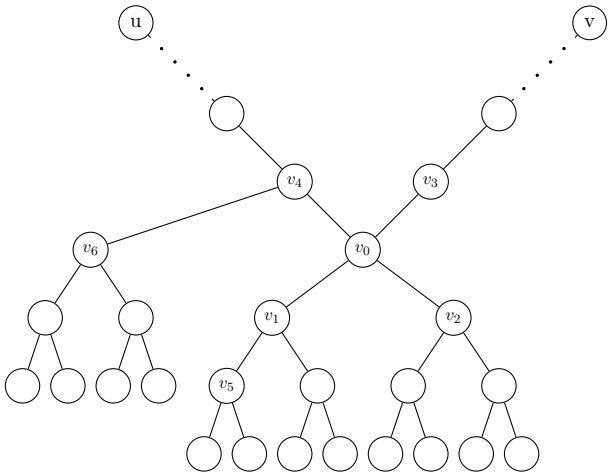


Figure 6.1: Construction in the proof of Proposition 6.6.

vertices are adjacent in the tree they are close in the graph? This might seem reasonable, as two vertices far apart in the graph set no requirement about being close in the adjacency tree in the definition. This would be a great tool in the direction of discovering feasible algorithms for bigger classes of graphs. However, Proposition 6.6 shows that this is not the case even when we restrict our input graph to trees of bounded degree.

Proposition 6.6. *There exist no function $f(d,k)$ that satisfies the following: Given a tree G of maximum degree d, there is an optimal adjacency tree (T,r) of G s.t. if $uv \in E(T)$ then $\text{dist}_G(u,v) \leq f(d,k)$.*

Proof. Let G be the graph in Figure 6.1, where $\text{dist}_G(0,u) = \text{dist}_G(0,v) = c$ for some $c \geq 3$. Figure 6.2 shows that $\text{as}(G) = 2$. We will now prove that u and v have to be neighbors in any optimal adjacency tree. Note that two of the neighbors of v_0 should be under v_0 and two of them should be above in a solution.

Assume v_1 to be the parent of v_0 in some optimal adjacency tree. At least one of the neighbors of v_1 must be an ancestor of v_1. Assume without loss of generality that this is v_5. But now we have used the two positions right above v_0, denying us to put another of the neighbors of v_0 above v_0, as this vertex then will have span more than 2. This gives us a contradiction.

A similar case arise if v_1 is an ancestor of v_0, but not the parent of v_0. Hence the two positions under v_0 must be taken by v_1 and v_2. Hence v_3 and v_4 must be ancestors of v_0. If we let v_3 be the parent of v_0, we run in to the same kind of contradiction as we did when we tried to put v_1 above v_0 because of v_6.

Hence v_1, v_2, v_3 and v_4 has to be positioned relative to v_0 as in Figure 6.2. And after this we have no choice regarding the rest of the tree above v_0, each

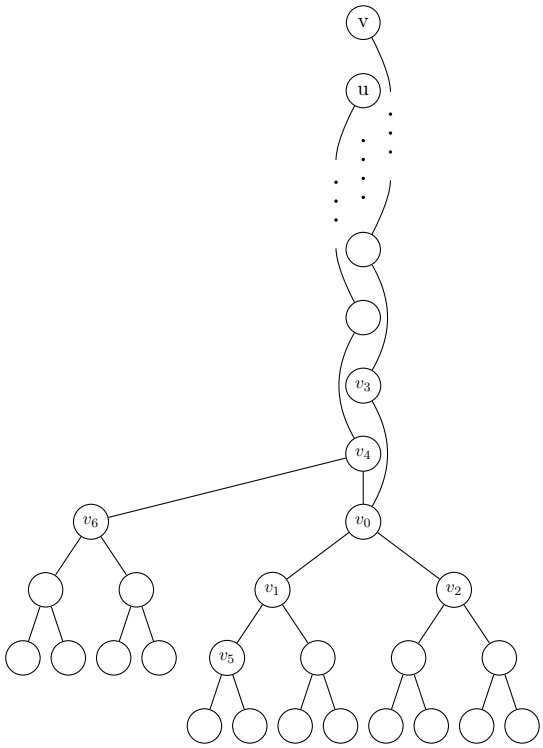


Figure 6.2: Adjacency tree of the graph in Figure 6.1 of adjacencyspan 2. The edges are the edges from the graph, while the structure of the solution is displayed through the positioning of the vertices.

vertex will already have span 2 and must send its last neighbor upwards. Hence we are forced to put u and v as neighbors in any optimal adjacency tree. Since they have distance $2c$ in the graph for some c, and c is not dependent on neither k nor d, there exist no such function f and our proof is complete. □

Chapter 7

Adjacencyspan parameterized

7.1 Double parameterization

In this section we prove that the problem Adjacencyspan admits a polynomial kernel when parameterized by both the vertex cover number and the requested adjacencyspan. The definition of parametrized problems do not permit double parameterizations directly, however both parameters will be bounded by the sum of the two and hence we can use this as a formal parameter to make it fit the definition. But to ease our work, we will consider the problem to be parametrized by two numbers in the rest of this section.

From Chapter 3 we know that a problem is in FPT if and only if it admits a kernel. However, since the kernel extracted from the proof is exponential, one often tend to consider problems which admits polynomial kernels more interesting from a kernelization point of view. After proving the polynomial kernel, we will continue by giving two algorithms one can apply to the kernel. Although a kernel implies the problem to be in FPT, the motivation for presenting the additional algorithms is that they give much better running times than applying brute force on the kernel. In addition, our first algorithm will be a warm-up for the next section. Which of the two algorithms provides the best running time depends on the size of the parameters.

p-Adjacencyspan$/s+k$	
Input:	A graph G, the vertex cover number s of G and k.
Parameter:	s and k.
Question:	Is $\text{as}(G) \leq k$?

In addition, we provide a definition of treespan parameterized by both vertex cover number and required treespan for completeness.

p-TREESPAN$/s+k$	
Input:	A graph G, the vertex cover number s of G and k.
Parameter:	s and k.
Question:	Is $\text{ts}(G) \leq k$?

Polynomial kernel

Theorem 7.1. *p-ADJACENCYSPAN$/s+k$ admits a polynomial kernel with $s(2k+1)$ vertices, $sk(2k+1)$ edges and maximum degree bounded by $2k$. The kernel can be computed in $O(kn)$ or $O(n+m)$ time.*

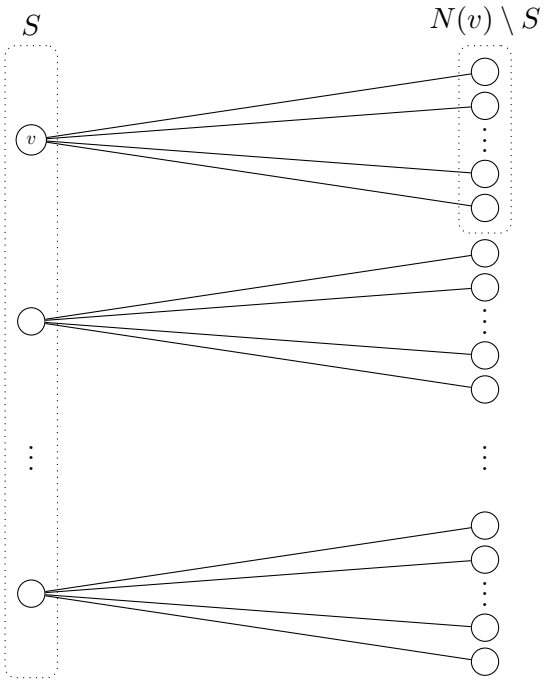


Figure 7.1: Illustration of the proof of Theorem 7.1.

Proof. Let S be a vertex cover of G of size s. Assuming that $\text{as}(G) \leq k$ we know that the degree of any vertex in G is at most $2k$ and it follows that $|N(S)| \leq 2sk$. By the facts that G is connected and S is a vertex cover we get that $V(G) = N(S) \cup S$ and hence $|V(G)|$ is at most $2sk + s = s(2k+1)$. It follows from $\Delta(G) \leq 2k$ and the handshaking lemma [BLW86] that the number of edges is bounded by $sk(2k+1)$. Given G, s and k we return a trivial no-instance if $\Delta(G) > 2k$ or $n > s(2k+1)$ or $m > sk(2k+1)$. Otherwise our instance is a kernel and we return it. Recall from Chapter 5

that we can verify that $\Delta(G) \leq 2k$ in $O(kn)$ time. If this is true $m \leq kn$, and hence the number of vertices and edges can also be counted in $O(kn)$ time. □

Notice that as Vertex Cover has a 2-approximation [PS98] this result can be applied efficiently even if you are not given s as part of the input. From Theorems 4.7 and 7.1 we get the following result.

Theorem 7.2. *p-Treespan$/s+k$ admits a polynomial kernel with $s(2k+1)$ vertices, $sk(2k+1)$ edges and maximum degree bounded by $2k$. The kernel can be computed in $O(kn)$ or $O(n+m)$ time.*

Generating trees

Borchardt [Bor61] proved that there are n^{n-2} trees on n labeled vertices, which has later become known as Cayles's formula in the literature. Furthermore, Prüfer [Prü18] provided a bijection between $\{1,\ldots,n\}^{n-2}$ and the trees on n labeled vertices. In addition, this bijection is known to be linear time computable [MCD06]. Hence we can generate all rooted trees on n labeled vertices in $O(n^n)$ time. This is done by for every element in $\{1,\ldots,n\}^{n-2}$, pick an element $r \in \{1,\ldots,n\}$ and apply the bijection to get the tree and let r be your root. Hence you obtain every rooted tree in $O(n^{n-2} \cdot n \cdot n) = O(n^n)$. In addition, we can guarantee that the trees generated are branched adjacency trees of treespan at most k in $O(n^{n-2} \cdot n \cdot (n+n^2+n^2)) = O(n^{n+1})$. If we apply this algorithm on the kernel from Theorem 7.1 we obtain an $O(nk + (2ks+s)^{2ks+s+1})$ time algorithm for p-Adjacencyspan$/s+k$. This running time will be improved twice by the next algorithms.

First algorithm

The scheme of our first algorithm will be to find an optimal vertex cover S, try all adjacency trees over S and then insert the rest of the vertices into this tree in every possible, valid way. Let $G = (V, E)$ be a graph, S an optimal vertex cover of G and (T, r) an adjacency tree of $G[S]$. We say that a vertex $v \in V \setminus S$ is of *type* N if $N = \text{type}(v) = N_G(v) \cap S$. We define the *neighborhood center* of a vertex $v \in V \setminus S$ of type N as the vertex u furthest away from the root such that $N \cap T_u$ is non-empty and $N \subseteq T_u \cup \{w \mid w \text{ is on the simple path from u to r}\}$.

Lemma 7.3. *Given a graph G, a set $S \subseteq V(G)$, an adjacency tree (T, r) of $G[S]$ and a vertex $v \in V(G) \setminus S$, the neighborhood center of v is computable in time linear in $|S|$.*

Proof. Mark all the vertices in S that are in type(v). Start in the root and execute Algorithm 7.4.

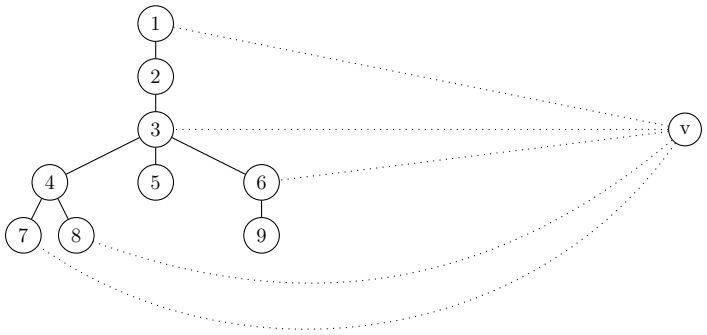


Figure 7.2: Let the tree on the left be an adjacency tree of $G[S]$. The dotted edges illustrate edges in G, hence the type of v is $\{1, 3, 6, 7, 8\}$. Note that the neighborhood center of v is vertex 3.

Algorithm 7.4 Neighborhood center

```
1: function NEIGHBORHOOD-CENTER(v, T, r)
2:     if v is a leaf then
3:         return v if v is marked and null otherwise
4:     Let c_1, ..., c_m be the children of v
5:     For each i, let u_i = NEIGHBORHOOD-CENTER(c_i, T, r)
6:     if u_1 = ··· = u_m = null then
7:         return v if v is marked and null otherwise
8:     if there is exactly one u_i ≠ null then
9:         return u_i
10:    return v
```

Since the algorithm clearly runs in $O(s)$ time, it remains to prove its correctness. Let u be the neighborhood center of v and assume for a contradiction that the algorithm returns $u' \neq u$. In addition we can assume that u and u' have an ancestor-descendant relationship, as otherwise would be a contradiction to the fact that u is the neighborhood center of v. If u is an ancestor of u', the algorithm must have returned u' while processing u. But this is only done if only one child of u returns something else than null and hence exactly one of the subtrees below u contains elements from type(v), which is a contradiction to the fact that u is the neighborhood center. Hence u' must be an ancestor of u. But now the only way the algorithm can return u', as u is a descendant of u', is if several of the branches below return something else than null. But this also contradicts the fact that u is the neighborhood center and hence our proof is complete. □

For a graph G, a set $S \subseteq V(G)$ and an adjacency tree (T, r) of $G[S]$, we

define a *k-embedding* of G into (T, r) to be an adjacency tree (T', r') of G with adjacencyspan at most k, such that for $u, v \in S$ we have that u is an ancestor of v in (T', r') if and only if u is an ancestor of v in (T, r). If there is a k-embedding of G into (T, r) we say that (T, r) is k-embeddable with respect to G.

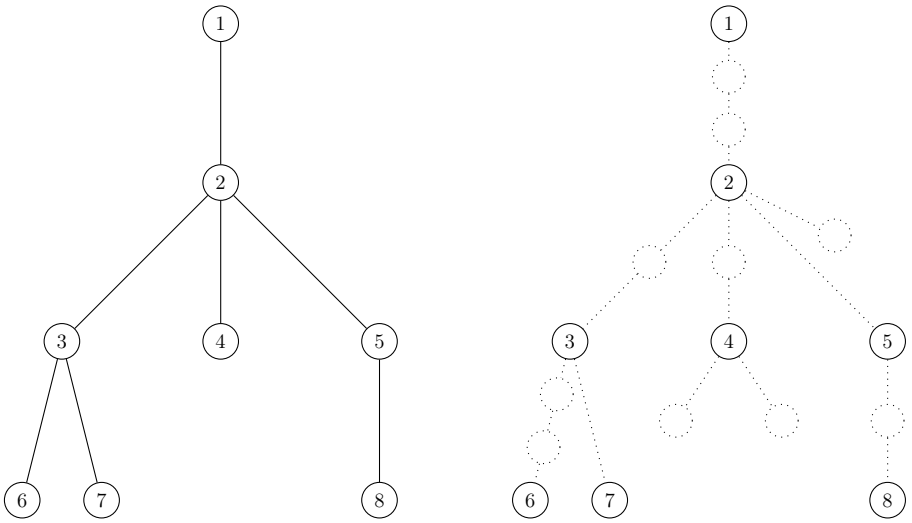


Figure 7.3: An illustration of how a k-embedding (b) might look like compared to the adjacency tree of $G[S]$ (a) for some G, S and k. The dotted vertices are the elements of $V(G) \setminus S$.

Lemma 7.5. *Let G be a graph, S an optimal vertex cover of G and (T, r) an adjacency tree of $G[S]$. If there is a branched k-embedding (T', r') of G into (T, r), then for every vertex $v \in V \setminus S$ with u being the neighborhood center of v, v is either on the $k+1$-root path of u or u is the parent of v in (T', r'). Furthermore, if u is the parent of v then $T_u \cap \text{type}(v) = \{u\}$.*

Proof. Assume that $v \notin T'_u$. Then v must be an ancestor of u for (T', r') to satisfy the ancestor-descendant property, since $T_u \cap N(v)$ and hence $T'_u \cap N(v)$ is non-empty. In addition, since $\text{type}(v) \cap T'_u$ is non-empty, the distance between u and v is at most k since $\text{as}(T', r') \leq k$. It follows directly that v lies on the $k+1$-root path of u.

Assume that $v \in T'_u$. By the definition of u we know that either there are two vertices $a, b \in T'_u \cap \text{type}(v)$ such that u is an ab-separator, or $T \cap \text{type}(v) = \{u\}$. But if such an a and b exist, then v cannot have an ancestor-descendant relationship with both a and b, which is a contradiction and hence $T_u \cap \text{type}(v) = \{u\}$.

Assume for a contradiction that $v \in V(T'_u)$ and that v is not a child of u in (T', r'). It follows that there is a child c of u such that $v \in V(T'_c)$. From

$T_u \cap \text{type}(v) = \{u\}$ and that S is a vertex cover it follows that v has no neighbors in T'_c and hence $G[V(T'_c)]$ is not connected, which contradicts that (T', r') is a branched adjacency tree by Lemma 4.9. Hence v must be a child of u if $v \in V(T'_u)$. □

Algorithm 7.6 Double parameterization

Input: a graph G and natural numbers s and k
Output: true if and only if $\text{as}(G) \leq k$

1: Apply kernelization from Theorem 7.1.
2: Compute a vertex cover S of G of size s.
3: For every adjacency tree (T, r) of $G[S]$:
4: Find the neighborhood center for every vertex in $V(G) \setminus S$.
5: For every vertex $v \in V(G) \setminus S$:
6: Try every position of v above u within distance k.
7: If $T_u \cap \text{type}(v) = \{u\}$ try to set u as the parent of v.
8: If the tree has adjacencyspan at most k, return true.
9: Return false.

Lemma 7.7. *The problem p-*ADJACENCYSPAN*$/s + k$ can be solved in time $O(kn + s^{2sk+s+5}k^3)$ by Algorithm 7.6.*

Proof. We know from Theorem 4.11 that if $\text{as}(G) \leq k$ there is a branched adjacency tree (T', r') of span at most k. Then, at some point we will obtain a tree (T, r) for the vertices in S at step 4, such that (T', r') is a k-extension of G into (T, r). Since $V(G) \setminus S$ is an independent set, it follows from the definition of the neighborhood center that the tree we construct will be a valid adjacency tree. Furthermore, from Lemma 7.5 we will at some point retrieve (T', r') or another k-extension of (T, r) in step 6-8 and hence the algorithm never will always find a solution if one exist. Since the algorithm constructs valid adjacency trees and checks the span of the tree at step 9 it will never return true without actually finding a solution. Correctness follows, and hence it remains to bound the running time of the algorithm.

- From Theorem 7.1 we know that the kernel can be computed in $O(kn)$ time.
- Recall from Chapter 3 that p-VERTEX COVER is solvable in $O(2^s n)$ time[1].
- Recall that the adjacency trees of a graph on n vertices can be generated in $O(n^{n+1})$ time, and hence step 4 can be done in $O(s^{s+1})$ time.

[1] Chen et al. [CKX10] gives an $O(1.2738^s + n)$ algorithm for p-VERTEX COVER, but the $O(2^s n)$ version is more than sufficient for our purposes.

- Step 5 is computable in $O(sn)$ time by Lemma 7.3.
- By Lemma 7.5 there is at most $s+1$ possibilities in steps 7 and 8 in total. Hence the running time of steps 6, 7 and 8 is $O((s+1)^{n-s}) = O(s^{n-s})$.
- Step 9 is clearly computable in $O(n^2)$ time.

This gives a running time of $O(kn + 2^s n + s^{s+1} \cdot sn \cdot s^{n-s} \cdot n^2) = O(kn + 2^s n + s^{n-s+s+2} n^3) = O(kn + s^{n+2} n^3)$. Exploiting the fact that $n \leq s(2k+1)$ from step 3 and onwards we get $O(kn + s^{2sk+s+2}(s(2k+1))^3) = O(kn + s^{2sk+s+5}k^3)$. □

The next corollary follows directly from Theorem 4.7 and Lemma 7.7.

Corollary 7.8. *The problem* p-Treespan$/s+k$ *can be solved in time* $O(kn + s^{2sk+s+5}k^3)$.

Second algorithm

Lemma 7.9. *The problem* p-Adjacencyspan$/(s+k)$ *can be solved in* $O(nk + 2^k k^{4k+4} s^{k+1})$ *time.*

Proof. Given an instance G, s and k apply the kernelization algorithm from Theorem 7.1. From Theorem 5.10 we know that Adjacencyspan is solvable in $O(k^{3k+3} n^{k+1})$, where n is the number of vertices and k is the adjacencyspan requested. After applying the kernelization algorithm, we know that $n \leq s(2k+1)$ and hence $O(k^{3k+3}(s(2k+1))^{k+1}) = O(k^{3k+3} s^{k+1} 2^{k+1} k^{k+1}) = O(2^k k^{4k+4} s^{k+1})$. This results in the running time $O(kn + 2^k k^{4k+4} s^{k+1})$. □

The next corollary follows directly from Theorem 4.7 and Lemma 7.9.

Corollary 7.10. *The problem* p-Treespan$/(s+k)$ *can be solved in time* $O(nk + 2^k k^{4k+4} s^{k+1})$.

7.2 Parameterized by vertex cover number

In the previous section we proved that Adjacencyspan is in FPT when parameterized by both the vertex cover number and the adjacencyspan number. In this section we continue this work by proving a theoretically stronger result, that Adjacencyspan parameterized only by the vertex cover number is in FPT. Note though that $f(s)$ will be doubly exponential in s and hence more of a classification result than one of practical interest. One might ask the question of why it is interesting to parameterize Adjacencyspan by something like the vertex cover number. First of all, the two parameters are incomparable, meaning that one can construct instances with arbitrarily

large adjacencyspan and with a constant size vertex cover and the other way around. In some sense adjacencyspan will be low if the graph has has a large diameter compared to the number of vertices, while this gives a high vertex cover. This makes it possible to create instances where a fixed parameter tractable algorithm parameterized by the vertex cover number will be significantly faster than most other algorithms. Also a fixed parameter tractable algorithm proves that for a specific parameterization one can capture the explosion in the running time within this parameter, which can be an interesting observation by itself.

p-Adjacencyspan/s	
Input:	A graph G, the vertex cover number s of G and an integer k.
Parameter:	s.
Question:	Is $\text{as}(G) \leq k$?

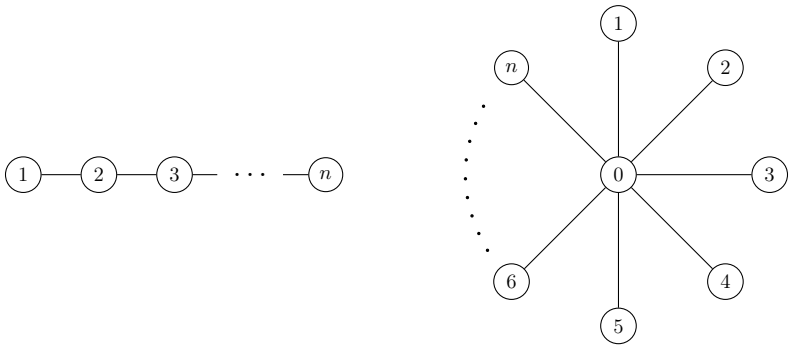


Figure 7.4: P_n has adjacencyspan 1 and vertex cover number $\lceil n/3 \rceil$. While S_n has adjacencyspan $\lceil n/2 \rceil$ and vertex cover number 1.

p-Integer Linear Programming Feasibility (ILP)	
Input:	Matrices $A \in \mathbb{Z}^{m \times p}$ and $b \in \mathbb{Z}^{m \times 1}$.
Parameter:	Number of variables p.
Question:	Does there exist a vector $x \in \mathbb{Z}^{p+1}$ satisfying $A \cdot x \leq b$?

The classical result that ILP parameterized by the number of variables is in FPT was first proven by Lestra [LJ83], a paper for which he received the Fulkerson Prize in 1985. After this, several papers improving the running time of this problem has been published [Kan87, FT87]. Fellows et al. [FLM+08] continued this work by proving Theorem 7.11. After which they used this result to prove that several linear ordering problems admits fixed parameter tractable algorithms parameterized by the vertex cover number.

p-Integer Linear Programming Optimization (Opt-ILP)	
Input:	Matrices $A \in \mathbb{Z}^{m \times p}$, $b \in \mathbb{Z}^{m \times 1}$ and $c \in \mathbb{Z}^{1 \times p}$.
Parameter:	Number of variables p.
Question:	Find a vector $x \in \mathbb{Z}^{p+1}$ that minimizes the objective function $c \cdot x$ and satisfies $A \cdot x \geq b$.

Theorem 7.11 ([FLM+08]). *The problem* Opt-ILP *can be solved using* $O(p^{2.5p+o(p)} \cdot L \cdot \log MN)$ *arithmetic operations and space polynomial in* L. *Here,* L *is the number of bits in the input,* N *is the maximum of the absolute values any variable can take and* M *is an upper bound on the absolute value of the minimum taken by the objective function.*

We will start in the same manner as in Algorithm 7.6 in the previous section, by computing a vertex cover S of size s, build all trees on these vertices and try to embed the rest of the vertices into this tree. The obstacle is that there is no way to bound the number of vertices outside the vertex cover any more. Hence we focus our work on proving more structure on the solution tree itself. We will prove that between two vertices in the tree, there is an embedding with a structure bounded by a function of s. After this we will do brute force on how this structure looks like, before using Theorem 7.11 to embed the rest of the vertices into the tree.

Preparation

Let G be a graph, S a corresponding vertex cover of size s, (T, r) a branched adjacency tree of $G[S]$ and (T', r') a branched k-embedding of G into (T, r). Let $u, v \in S$ be two vertices such that u is the parent of v in (T, r). Since (T', r') is branched and S is a vertex cover we know that every internal vertex on the simple path from u to v in (T', r') is of degree 2 in (T', r'). We define the *uv-interval* with respect to (T', r') as $P \setminus \{u, v\}$, and denote it I_{uv}. We will always let u be the vertex of the two closest to the root. Furthermore we define the *root interval* I_r as $\{v \mid v \text{ is on the path from } r \text{ to } r' \text{ in } (T', r')\} \setminus \{r\}$. For an interval I we define a *sub interval* as a connected sub path of I.

When we traverse an interval I or speak of an ordering of the vertices in I, we will apply the natural ordering from one of the endpoints to the other. For a type N for which a vertex of this type appears in I, let the *start vertex* of N be the first vertex appearing in I of this type and the *end vertex* be the last vertex of this type appearing in I. And let the sub interval from the start vertex to the end vertex of a type N be called the *safe interval* of N. Observe that moving vertices, as long as each vertex still lives in its original safe interval, does not increase the adjacencyspan.

We adopt the notion of zones and zonal dimension of embeddings from Fellows et al. [FLM+08]. We say that a sub interval is *uniform* if all vertices in the sub interval is of the same type. A *zone* is then defined as

a maximal uniform sub interval. For an interval I in (T', r') we define the *zonal dimension* of I, denoted $\zeta_I(T', r')$, as the number of zones in I. And the zonal dimension of (T', r'), denoted $\zeta(T', r')$, as $\max\{\zeta_I(T', r') \mid I \text{ is an interval in } (T', r')\}$.

Lemma 7.12. *Let G be a graph, S a vertex cover of size at most s and (T, r) an adjacency tree of $G[S]$ such that there is a branched k-embedding of G into (T, r). Then there exists a branched k-embedding (T', r') of G into (T, r) such that $\zeta(T', r') \leq 2^{s+1}$.*

Proof. Let (T', r') be any branched k-embedding of G into (T, r). We will prove that for any interval I in (T', r') we can rearrange I such that $\zeta_I(T', r') \leq 2^{s+1}$, while all vertices stay within their safe zone.

Let I be an interval in (T', r'). Take the type N for which the safe interval ends first. Rearrange the vertices in the safe interval such that the vertices of type N comes first and keep the ordering internally between the other elements. Clearly the ones of type N stay inside their safe interval and as N was the type to end first all the others stay within their safe intervals as well. Hence no element will be moved outside of its safe interval by this operation. Remove the vertices of type N and connect the two vertices who had neighbors of type N. This reduces the zonal dimension of the interval by 2. Continue this process until the interval is empty. By going backwards and inserting the types in the positions where they were removed we obtain an interval who respects the old safe intervals. This interval has two times the number of removals as zonal dimension. Notice that as $|S| \leq s$ there is at most 2^s different types and hence at most 2^s removals. Which implies that $\zeta_I(T', r') \leq 2^{s+1}$ and by applying this procedure on all intervals we get $\zeta(T', r') \leq 2^{s+1}$ and hence our proof is complete. □

Note that for a vertex v of type N, if u is a vertex strictly inside the smallest subtree containing N, it does not matter if u is in N or not with respect to the span of v. In Figure 7.2 it would not matter if the type of v is $\{1, 3, 6, 7, 8\}$ or $\{1, 6, 7, 8\}$. Hence one might believe that the bound in Lemma 7.12 could be improved. But if (T, r) is a star with r as its center, we have $2^s + 1$ different types, proving that 2^{s+1} is optimal within a constant factor.

Definition 7.13. Let G be a graph, s and k integers, S a vertex cover of size s and (T, r) an adjacency tree of $G[S]$. Assume that we know the type of each zone of every interval of a k-embedding (T', r') of G into (T, r). Let the variable x_I^i represent the size of zone j in interval I in (T', r'). And for each type N such that v is the neighborhood center of N and $N \cap T_v = \{v\}$ let the variable x_v^N be the number of variables of type N with v as a parent. Let x_{span} be the span variable, representing the adjacencyspan of the solution we find. Let (H, r_H) be the tree we obtain when setting every variable to one

and inserting the new vertices in (T, r). For a vertex u in H let x_u be 1 if u is an element of S and the value of the corresponding variable otherwise.

Span constraint
For a vertex $v \in H$ we define the *span constraint* of v as:

$$\sum H_v^m \leq x_{\text{span}} + 1.$$

Type constraint
For a type N such that there is n_N vertices of type N in $V(G) \setminus S$ we define the *type constraint* as:

$$\sum Z(N) = n_N, \text{ where}$$

$$Z(N) = \{x \mid \text{the vertices } x \text{ represents is of type } N\}.$$

Span-ILP
We define *span-ILP* as the following integer linear program:

$$\begin{aligned} \min \quad & x_{\text{span}} \\ \text{such that} \quad & \\ & \text{every vertex in } H \text{ satisfies the span constraint,} \\ & \text{every type satisfies the type contraint and} \\ & \text{every variable takes values in the interval } [0, k+1]. \end{aligned}$$

The span constraint ensures that no vertex, in S or outside, has a span of more than x_{span} and the type constraint ensures that we embed all vertices in G into the tree.

Algorithm

We are now ready to prove that p-AdjacencySpan$/s$ is fixed-parameter tractable when parameterized by the vertex cover number. The scheme of our algorithm will start of as Algorithm 7.6. First we will find a vertex cover S and then we exhaustively try tree structures of S. Then we try every possible zone structure for every interval and derive an integer linear program where each variable represents the size of a zone and solves it.

Theorem 7.15. *The problem p-AdjacencySpan$/s$ is in* FPT *since it can be solved in time $O(s^{s+2}4^s n + (s2^s)^{2.5(s+1)2^s + o(s2^s)} \log k)$.*

Proof. From Theorem 4.11 we know that if $\text{as}(G) \leq k$ there is a branched adjacency tree (T', r') of G with adjacencyspan at most k. If such an (T', r') exist, then it is clear that at some point at step 2, we will find an adjacency tree (T, r) of $G[S]$ such that (T', r') is a k-extension of G into (T, r). By Lemma 7.12 step 4 and 5 will eventually find the zonal information for some branched adjacency tree (T'', r'') such that $\text{as}(T'', r'') \leq \text{as}(T', r')$.

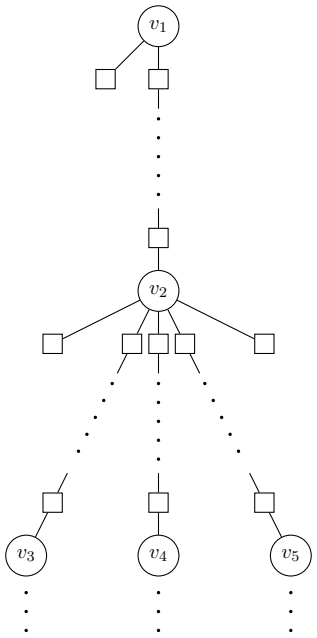


Figure 7.5: An example of how (H, r_H) could look like. Note that there are two types, $\{v_2\}$ and $\{v_1, v_2\}$, that has v_2 as its neighborhood center such that intersection between the type and T_2 is v_2. Hence v_2 has 2 variables not in an interval as children while v_1 only has 1.

Algorithm 7.14 Parameterized by the vertex cover number

Input: a graph G and integers s and k
Output: true if and only if $\text{as}(G) \leq k$

1: Compute a vertex cover S of G of size s.
2: For every adjacency tree (T, r) of $G[S]$:
3: For every number of zones in each interval
4: For every type of each zone:
5: Verify that (H, r_H) respects the ancestor-descendant property.
6: Create the span-ILP.
7: Solve the span-ILP, return true if x_{span} is at most k.
8: Return false.

Then span-ILP will embed $V(G) \setminus S$ into (T, r) in such a way that the resulting branched adjacency tree has optimal adjacencyspan and hence has adjacencyspan at most k. It follows that the algorithm will always return true if the answer is true. From step 5 we know that every solution will satisfy the ancestor-descendant property, from the type constraints we know that every vertex will be embedded in the tree, and from the span constraints

we know that every vertex will have span at most x_{span}. Hence the algorithm will only return true if it found a valid solution and the correctness of the algorithm follows.

Observe that there is $s2^s$ variables representing zones and 2^s variables representing vertices with the neighborhood center as their parent. Hence there are $(s+1)2^s$ variables in span-ILP and the size of (H, r_H) is $(s+1)2^s+s$. It remains to bound the running time of the algorithm.

- From Chapter 3 we know that step 1 is solvable in time $O(2^s n)$.
- Recall that we can find all adjacency trees in $O(s^{s+1})$ time.
- The number of zones in each interval is bounded by 2^{s+1} by Lemma 7.12 and there is at most 2^s different zones. Observe that there is s intervals and hence we get a running time of $O(s4^s)$ for step 3 and 4.
- Let h be the size of (H, r_h). Observe that (H, r_h) can be generated in $O(h)$ time and verify that (H, r_H) satisfies the ancestor-descendant property in $O(h^2)$ time. And each span constraint can also be generated in $O(h)$ time and as there is $O(h)$ many span constraints, hence all span constraints can be generated in $O(h^2)$ time. For the type constraints we first count the number of variables of each type in $O(n)$ time. Then we generate each type constraint in $O(h)$ time and hence generating all type constraints can be done in $O(h^2)$ time. This gives a running time for steps 5 and 6 of $O(h + h^2 + h^2 + n + h^2) = O(h^2 + n)$. Recall that $h = (s + 1)2^s + s$ and hence the span-ILP can be created in $O(h^2 + n) = O(((s + 1)2^s + s)^2 + n) = O(s^2 4^s + n)$ time.
- Solving span-ILP can be done in $O(p^{2.5p+o(p)} \cdot L \cdot \log(MN))$ by Theorem 7.11. We know that $L = O(s^2 4^s)$, $M, N \leq k$ and that $p = (s+1)2^s$. Hence span-ILP is solvable in $O(((s + 1)2^s)^{2.5(s+1)2^s+o((s+1)2^s)} \cdot s^2 4^s \cdot \log k) = O((s2^s)^{2.5s2^s+o(s2^s)} \cdot \log k)$ time. The $s + 1$ goes to a s by the following observation:

$$
\begin{aligned}
&O(((s + 1)2^s)^{f(s)}) = \\
&O((s2^s + 2^s)^{f(s)}) = \\
&O((s2^s)^{f(s)} + (s2^s)^{f(s)-1} \cdot 2^s + \cdots + (2^s)^{f(s)}) = \\
&O((s2^s)^{f(s)} + s^{f(s)-1}(2^s)^{f(s)} + \cdots + (2^s)^{f(s)}) = \\
&O((s2^s)^{f(s)}).
\end{aligned}
$$

In total we get the running time:

$$
\begin{aligned}
&O(2^s n + s^{s+1} \cdot s4^s \cdot (s^2 4^s + n + (s2^s)^{2.5(s+1)2^s+o(s2^s)} \log k)) = \\
&O(2^s n + s^{s+4} 16^s + s^{s+2} 4^s n + (s2^s)^{2.5(s+1)2^s+o(s2^3)} \log k = \\
&O(s^{s+2} 4^s n + (s2^s)^{2.5(s+1)2^s+o(s2^s)} \log k).
\end{aligned}
$$

□

Hence we have reached the goal of this section, to prove that ADJACENCYSPAN admits fixed parameter tractable algorithms when parameterized by the vertex cover number. Notice that by increasing the $\log k$ to $\log n$, we can solve the optimization version of the same problem. To complete this section we provide the corresponding result for TREESPAN. It follows from Theorems 4.7 and 7.15.

p-TREESPAN/s	
Input:	A graph G, the vertex cover number s of G and an integer k.
Parameter:	s.
Question:	Is $\text{ts}(G) \leq k$?

Theorem 7.16. *The problem* p-TREESPAN/s *is in* FPT *since it can be solved in time* $O(s^{s+2}4^s n + (s2^s)^{2.5(s+1)2^s + o(s2^s)} \log k)$.

Part IV

Discussion and conclusion

Chapter 8

Concluding remarks and open questions

8.1 Theoretical and practical implications of this work

In this thesis we have provided a new perspective to Treespan and a seemingly powerful tool for designing algorithms, in form of branched adjacency trees. We have provided an XP-algorithm, a polynomial time algorithm for trees of bounded degree, a polynomial kernel when parameterized by both the vertex cover number and the required treespan, and a fixed parameter tractable algorithm when parameterized by the vertex cover number only. The last result applies a tool of which Niedermeier asked of more applications [Nie06]. We have hence started the work of deciding the limits of the tractability of Treespan. However, several open questions remain, and we list some of them in the next sections in this chapter.

Watnedal implemented an algorithm for solving Treespan in his master thesis [Wat05]. He used dynamic programming to generate non-isomorphic trees and test them as solutions. He also made a specialized version for deciding if a graph has treespan 2. He reported that for a graph on 43 vertices, when labeling the vertices in the graph in a good way for the algorithm, it terminated in about 13 hours. This is the best known implemented algorithm solving Treespan. By Theorem 5.11 this problem is solvable in $O(n^2(n+m))$ time. As the running time $O(k^{2k+5}n^k(n+m)) = O(k^{2k+5}n^{k+2})$ hides no big constants, an implementation of Algorithm 5.4 should run in about $2^9 \cdot 43^4/10^7 \approx 175$ seconds, or almost 3 minutes, assuming that the computer is able to perform 10^7 not too heavy operations per second, which is highly likely for even most 5 year old laptops. This a drastic decrease in computation time compared to the implementation and hence this thesis can be considered a practical contribution to the computation of treespan. It is our hope that the algorithms in the thesis are given in a precise enough way

so that they can easily be implemented and turned into working code.

8.2 Caterpillars with long hair

A *caterpillar* is a tree which contains a path such that every vertex in the tree is at distance at most one from a vertex on this path. A *caterpillar of hair length l* is the natural generalization where each vertex should be at distance at most l from a vertex on the path. We adopt the notion of the *density* of a graph G from Chinn et al. [CCDG82] as $\text{dens}(G) = (n-1)/\text{diam}(G)$. Fomin et al. [FHT05] proved that if G is a caterpillar, then $\text{ts}(G) = \text{bw}(G) = \max\{\lceil \text{dens}(H) \rceil \mid H \subseteq G\}$. This is computable in $O(n \log n)$ time by Assmann et al. [APSZ81]. In fact they prove that $\text{bw}(G) = \max\{\lceil \text{dens}(H) \rceil \mid H \subseteq G\}$ also holds for caterpillars of hair length 2 and that these also are solvable in $O(n \log n)$ time. Furthermore, Monien [Mon86] proved Bandwidth to be NP-complete on caterpillars of hair length 3.

We know that for a star G, $\text{bw}(G) = \text{ts}(G)$. We can generalize this quite easily to *big stars*, meaning trees with at most one vertex of degree more than 2. One can prove this by the same kind of merging above the center as in Figure 6.2 and Figure 8.1. The fact that bandwidth equals treespan for both caterpillars and big stars might suggest that bandwidth is equal to treespan for caterpillars with longer hairs. However, we provide a caterpillar of hair length 2 below where this is not true. Let G be the caterpillar in

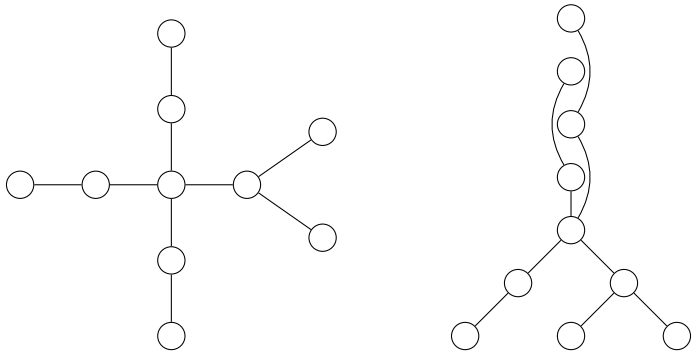


Figure 8.1: On the left, a caterpillar of hair length 2 where treespan is not equal to bandwidth. On the right, an adjacency tree proving that the treespan of the graph equals 2.

Figure 8.1. As

$$\text{bw}(G) \geq \text{dens}(G) = \lceil (n-1)/\text{diam}(G) \rceil = \left\lceil \frac{9}{4} \right\rceil = 3$$

we know that $\text{ts}(G) \neq \text{bw}(G)$. This motivates our first open problem.

Open problem 1. Given a caterpillar G of hair length l and an integer k, for which values of l can it be determined in polynomial time whether $\text{ts}(G) \leq k$, and for which values is it NP-complete?

8.3 Interval graphs

A graph $G = (V, E)$ is an *interval graph* if there exists a function f from V to intervals in $[0, \ldots, n]$ such that $uv \in E$ if and only if $f(u) \cap f(v) \neq \emptyset$. The function f is called the *interval representation* of G. If there is an interval representation of G such that no interval is a proper subset of another we say that the graph is a *proper interval graph*. Kleitman and Vohra [KV90] provided a polynomial time algorithm solving Bandwidth on interval graphs. Fomin et al. [FHT05] proved that for any proper interval graph G it holds that $\text{ts}(G) = \text{bw}(G)$ and hence it follows that Treespan is polynomial time solvable on proper interval graphs.

Open problem 2. Is $\text{bw}(G) = \text{ts}(G)$ for every interval graph G? If not, is there a polynomial time algorithm that decides whether an interval graph G has treespan at most k given G and k.

We have failed to provide a counter example to this question and believe that it is might be true that bandwidth and treespan in fact are equal also on interval graphs. It is known that every interval graph has an optimal tree decomposition which is a path such that every bag is a clique [GH03]. It follows that every minimal separator in an interval graph is a clique. Note that for a graph, an adjacency tree without any branching, meaning that the adjacency tree is a path, is also a valid solution to the bandwidth problem. Furthermore, observe that if an adjacency tree is to branch at some point, the path from this point to the root must be a separator in the graph, separating the branches. Hence if one proves that there exists an optimal adjacency tree such that for every separator in an interval graph, one can assume that one of the separated parts in the graph lives above the lowest vertex of the separator, equality follows. We believe that this is a direction that might lead to a complete proof.

8.4 General trees

While Bandwidth is proven to be NP-complete on trees of maximum degree 3 [GGJK78] we have provided a polynomial time algorithm in this thesis solving Treespan on trees of bounded degree. This raises the interesting question of whether Treespan is solvable in polynomial time for general trees. Proposition 6.6 proves that one cannot assume vertices close in the solution to be close in the input graph. As the construction from the proof of the lemma can be generalized to any degree on the center vertex, this

appears to motivate a construction for an NP-hardness reduction. But it appears to be difficult to introduce choices in the solution while still forcing the solution to maintain structure to express other problems. It is not clear how to solve Treespan even on trees with all vertices except one of bounded degree. This seems to be a case worthwhile to study as it is not known to be polynomial time solvable and also captures some of the ideas we have for an NP-reduction.

Open problem 3. Given a tree G and an integer k, is it polynomial time decidable whether the treespan of G is at most k?

Note that such polynomial-time algorithms on trees often yield fixed parameter tractable algorithms for general input parameterized by treewidth. But as treewidth is bounded by treespan, this seems unlikely. This might be an indication of non-existence of such an algorithm. Another interesting problem is whether Treespan admits a fixed parameter tractable algorithm on trees.

Open problem 4. Given a tree G and an integer k, is it decidable in $O(f(k) \cdot n^{O(1)})$ time whether the treespan of G is at most k?

8.5 Exponential algorithm

We adopt the notion of O^* from the book on exponential algorithms by Fomin and Kratsch [FK10]. We say that $f(n) = O^*(g(n))$ if there exists a polynomial such that $f(n) = O(g(n) \cdot \text{poly}(n))$. From the equivalence between some measure on elimination trees and treespan by Fomin et al. [FHT05] it follows that Treespan is solvable in $O^*(2^{O(n \log n)})$ time. An $O^*(10^n)$ algorithm was provided for Bandwidth by Feige and Kilian [FK00], which was later improved by Cygan and Pilipczuk [CP08] to $O^*(5^n)$. This raises the question of whether Treespan admits such an algorithm.

Open problem 5. Given a graph G and an integer k, is it possible to decide if G has treespan at most k in $O^*(2^{O(n)})$ time?

The techniques used for Bandwidth do not easily translate to Treespan. The algorithms are based on dividing the linear ordering into several intervals of size depending on k and then handling the intervals more or less separately. The first problem with applying this technique is that an adjacency tree might contain half of the vertices as leafs. Which implies that if we divide our tree into blocks by the distance from the root, the size of our blocks are not bounded by k. The other problem is that the vertices interact in a much more complex way in an adjacency tree. It is exploit in the algorithms for Bandwidth that one only cares about the neighbor furthest away to the left in the ordering. A similar statement cannot be given for

TREESPAN as they might be in different branches and both contribute to the span. We believe that these are two problems one will have to tackle for successfully solving this problem.

8.6 FPT vs W-hardness

The problem p-BANDWIDTH was proven to be W[t]-hard for every t by Bodlander et al. [BFH94]. The same question would be very interesting to answer for p-TREESPAN. This was discussed at a Dagstuhl seminar in 2011 [FFKT11]. We leave it here as an open problem.

Open problem 6. Is p-TREESPAN parameterized by k W[t]-hard for some $t \geq 1$ or does it admit a fixed parameter tractable algorithm?

8.7 Further work parameterized by vertex cover

We proved in this thesis that TREESPAN parameterized by the vertex cover number admits a fixed parameter tractable algorithm. In addition we provided a polynomial kernel for TREESPAN parameterized by both the vertex cover number and the treespan. The celebrated tool of OR-decompositions [BDFH09], designed to prove that polynomial kernels are not likely to exist for certain problems, is conjectured to hold also for AND-decompositions in the same paper. And a proof of this conjecture will be presented at a Dagstuhl seminar this year [FGMS12]. If it holds, it implies that TREESPAN parameterized by k is not likely to admit a polynomial kernel. This follows from the fact that the disjoint union of instances of TREESPAN has treespan at most k if and only if each of the original instances have treespan at most k. Hence it would be very interesting to investigate if TREESPAN admits a polynomial kernel parameterized by the vertex cover number.

Open problem 7. Is there a polynomial kernel for TREESPAN parameterized by the vertex cover number of the input graph?

Our running time for TREESPAN parameterized by the vertex cover number is doubly exponential. It would be interesting to improve this.

Open problem 8. Is is possible to improve our running time solving TREESPAN parameterized by the vertex cover number?

8.8 Treespan by other parameterizations

Another interesting path would be to investigate TREESPAN parameterized by other parameterizations. Normally, a natural next step after the vertex cover number would be the feedback vertex set number of the input graph.

But as this leaves the rest of the graph as a tree and there is no known algorithm solving TREESPAN on trees this seems highly non-trivial.

Open problem 9. Does TREESPAN admit a fixed parameter tractable algorithm when parameterized by other interesting parameters?

By this we feel the most interesting, unresolved problems regarding treespan have been stated.

Bibliography

[AG02] S. Alpern and S. Gal. *The theory of search games and rendezvous*, volume 55. Springer, 2002.

[APSZ81] SF Assmann, GW Peck, MM Sysło, and J. Zak. The bandwidth of caterpillars with hairs of length 1 and 2. *SIAM Journal on Algebraic and Discrete Methods*, 2:387, 1981.

[BDFH09] H.L. Bodlaender, R.G. Downey, M.R. Fellows, and D. Hermelin. On problems without polynomial kernels. *Journal of Computer and System Sciences*, 75(8):423–434, 2009.

[BFH94] H.L. Bodlaender, M.R. Fellows, and M.T. Hallett. Beyond np-completeness for problems of bounded width (extended abstract): hardness for the w hierarchy. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 449–458. ACM, 1994.

[BKW97] G. Blache, M. Karpiński, and J. Wirtgen. *On approximation intractability of the bandwidth problem.* Citeseer, 1997.

[BLW86] N. Biggs, E.K. Lloyd, and R.J. Wilson. *Graph Theory, 1736-1936.* Clarendon Press, 1986.

[Bón06] M. Bóna. *A walk through combinatorics: an introduction to enumeration and graph theory.* World Scientific Pub Co Inc, 2006.

[Bor61] C.W. Borchardt. *Über eine Interpolationsformel für eine art symmetrischer Functionen und über deren Anwendung.* Dr. d. Königl. Akad. d. Wiss., 1861.

[BS91] D. Bienstock and P. Seymour. Monotonicity in graph searching. *Journal of Algorithms*, 12(2):239–245, 1991.

[CCDG82] P.Z. Chinn, J. Chvátalová, A.K. Dewdney, and N.E. Gibbs. The bandwidth problem for graphs and matrices-a survey. *Journal of Graph Theory*, 6(3):223–254, 1982.

[CKJ01] J. Chen, I.A. Kanj, and W. Jia. Vertex cover: further observations and further improvements. *Journal of Algorithms*, 41(2):280–301, 2001.

[CKX10] J. Chen, I.A. Kanj, and G. Xia. Improved upper bounds for vertex cover. *Theoretical Computer Science*, 411(40-42):3736–3756, 2010.

[Coo71] S.A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[CP08] M. Cygan and M. Pilipczuk. Faster exact bandwidth. In *Graph-Theoretic Concepts in Computer Science*, pages 101–109. Springer, 2008.

[Der09] D. Dereniowski. Maximum vertex occupation time and inert fugitive: Recontamination does help. *Information Processing Letters*, 109(9):422–426, 2009.

[DF95] Rod G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness ii: On completeness for w[1]. *Theoretical Computer Science*, 141(1âĂŞ2):109 – 131, 1995.

[DF99] Rod G. Downey and R. Fellows. *Parameterized complexity*. Monographs in computer science. Springer, 1999.

[Die05] R. Diestel. Graph theory. 2005. *Grad. Texts in Math*, 2005.

[DKT97] N.D. Dendris, L.M. Kirousis, and D.M. Thilikos. Fugitive-search games on graphs and related parameters. *Theoretical Computer Science*, 172(1-2):233–254, 1997.

[DvM10] Holger Dell and Dieter van Melkebeek. Satisfiability allows no nontrivial sparsification unless the polynomial-time hierarchy collapses. In *Proceedings of the 42nd ACM symposium on Theory of computing*, STOC '10, pages 251–260, New York, NY, USA, 2010. ACM.

[FFKT11] F.V. Fomin, P. Fraigniaud, S. Kreutzer, and D.M. Thilikos. Theory and applications of graph searching problems. In *Dagstuhl Seminar*, number 11071 in Dagstuhl, 2011.

[FG64] D.R. Fulkerson and O.A. Gross. Incidence matrices and interval graphs. Technical report, DTIC Document, 1964.

[FG00] F.V. Fomin and P.A. Golovach. Graph searching and interval completion. *SIAM Journal on Discrete Mathematics*, 13(4):454–464, 2000.

[FG06] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. Springer, 2006.

[FGMS12] M.R. Fellows, J. Guo, D. Marx, and S. Saurabh. Data reductions and problem kernels. In *Dahstuhl Seminar*, number 12241 in Dagstuhl, 2012.

[FHT05] F.V. Fomin, P. Heggernes, and J.A. Telle. Graph searching, elimination trees, and a generalization of bandwidth. *Algorithmica*, 41(2):73–87, 2005.

[FK00] U. Feige and J. Kilian. Exponential time algorithms for computing the bandwidth of a graph. *Manuscript in preparation*, 2000.

[FK10] F.V. Fomin and D. Kratsch. *Exact exponential algorithms*. Springer Verlag, 2010.

[FLM+08] M. Fellows, D. Lokshtanov, N. Misra, F. Rosamond, and S. Saurabh. Graph layout problems parameterized by vertex cover. *Algorithms and Computation*, pages 294–305, 2008.

[FT87] A. Frank and É. Tardos. An application of simultaneous diophantine approximation in combinatorial optimization. *Combinatorica*, 7(1):49–65, 1987.

[GGJK78] M.R. Garey, R.L. Graham, D.S. Johnson, and D.E. Knuth. Complexity results for bandwidth minimization. *SIAM Journal on Applied Mathematics*, pages 477–495, 1978.

[GH03] PC Gilmore and AJ Hoffman. A characterization of comparability graphs and of interval graphs. *Selected papers of Alan Hoffman with commentary*, 16:65, 2003.

[GHK+09] P. Golovach, P. Heggernes, D. Kratsch, D. Lokshtanov, D. Meister, and S. Saurabh. Bandwidth on at-free graphs. *Algorithms and Computation*, pages 573–582, 2009.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and intractability*, volume 174. Freeman San Francisco, CA, 1979.

[KAM30] FP KAMSEY. On a pboblem of fokmal logic. 1930.

[Kan87] R. Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of operations research*, pages 415–440, 1987.

[Kar10] R.M. Karp. Reducibility among combinatorial problems. *50 Years of Integer Programming 1958-2008*, pages 219–241, 2010.

[KP86] L.M. Kirousis and C.H. Papadimitriou. Searching and pebbling. *Theoretical Computer Science*, 47:205–218, 1986.

[Kra12] S. Kratsch. Co-nondeterminism in compositions: A kernelization lower bound for a ramsey-type problem. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 114–122. SIAM, 2012.

[KV90] D.J. Kleitman and R. Vohra. Computing the bandwidth of interval graphs. *SIAM Journal on Discrete Mathematics*, 3(3):373–375, 1990.

[Liu90] J.W.H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134, 1990.

[LJ83] H.W. Lenstra Jr. Integer programming with a fixed number of variables. *Mathematics of operations research*, pages 538–548, 1983.

[MCD06] P. Micikevicius, S. Caminiti, and N. Deo. Linear-time algorithms for encoding trees as sequences of node labels. *Congressus Numerantium*, 183:65, 2006.

[MHG+88] N. Megiddo, S.L. Hakimi, M.R. Garey, D.S. Johnson, and C.H. Papadimitriou. The complexity of searching a graph. *Journal of the ACM (JACM)*, 35(1):18–44, 1988.

[Mih10] R. Mihai. *Games on graphs: searching and online coloring.* PhD thesis, University of Bergen, 2010.

[Mon86] B. Monien. The bandwidth minimization problem for caterpillars with hair length 3 is np-complete. *SIAM Journal on Algebraic and Discrete Methods*, 7:505, 1986.

[Nie06] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms.* Oxford Lecture Series in Mathematics And Its Applications. Oxford University Press, 2006.

[Par78] T. Parsons. Pursuit-evasion in a graph. *Theory and applications of graphs*, pages 426–441, 1978.

[Pet82] NN Petrov. A problem of pursuit in the absence of information on the pursued. *Differentsial'nye Uravneniya*, 18(8):1345–1352, 1982.

[Prü18] H. Prüfer. Neuer beweis eines satzes über permutationen. *Arch. Math. Phys*, 27:742–744, 1918.

[PS98] C.H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Dover Pubns, 1998.

[Rad09] S.P. Radziszowski. Small ramsey numbers. *Electronic Journal of Combinatorics*, 1, 2009.

[Rau05] D. Rautenbach. Lower bounds on treespan. *Information processing letters*, 96(2):67–70, 2005.

[RS83] A.L. Rosenberg and I.H. Sudborough. Bandwidth and pebbling. *Computing*, 31(2):115–139, 1983.

[Sax80] J.B. Saxe. Dynamic-programming algorithms for recognizing small-bandwidth graphs in polynomial time. *SIAM Journal on Algebraic and Discrete Methods*, 1:363, 1980.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[Spe94] J.H. Spencer. *Ten lectures on the probabilistic method*, volume 64. Society for Industrial Mathematics, 1994.

[Sto73] L. Stockmeyer. Planar 3-colorability is polynomial complete. *ACM Sigact News*, 5(3):19–25, 1973.

[Tur38] A.M. Turing. On computable numbers, with an application to the entscheidungsproblem. a correction. *Proceedings of the London Mathematical Society*, 2(1):544, 1938.

[Ung98] W. Unger. The complexity of the approximation of the bandwidth problem. In *Foundations of Computer Science, 1998. Proceedings. 39th Annual Symposium on*, pages 82–91. IEEE, 1998.

[Wat05] K. Watnedal. Avgjøre små treespenn. Master's thesis, University of Bergen, 2005.

[Woe03] Gerhard Woeginger. Exact algorithms for np-hard problems: A survey. In Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi, editors, *Combinatorial Optimization - Eureka, You Shrink!*, volume 2570 of *Lecture Notes in Computer Science*, pages 185–207. Springer Berlin / Heidelberg, 2003.