Master's Thesis

Spanheight, A Natural Extension of Bandwidth and Treedepth

Author:
N. van Roden

Supervisor:
Prof. dr. Hans. L. Bodlaender

A thesis submitted in fulfilment of the requirements for the degree of Master of Science

in the

Faculty of Science
Department of Information and Computing Sciences

ICA-4206754

October, 2015

Universiteit Utrecht

Abstract

In graph theory, the bandwidth problem has a long history, and a number of practical applications. Fomin, Heggernes, and Telle [20] introduced treespan, an extension from bandwidth to depth first search spanning trees in the context of the occupancy measure in search games. It is the equivalent of a tree-decomposition, where adjacent bags differentiate with at most 1 vertex, and each vertex is only allowed to appear in at most k bags. Dregi [16] explored parameterized algorithms for treespan under the name adjacencyspan.

In this thesis, we define spanheight as a different extension from bandwidth to depth first search spanning trees. A spanheight-decomposition is a DFS spanning tree, in which every pair of adjacent vertices are connected with a path of length at most $k + 2$. It is the equivalent of a tree-decomposition, where adjacent bags differentiate with at most 1 vertex, and the sub-tree induced by bags containing a vertex v has height at most $k + 2$.

We proof spanheight to be NP-Complete. We introduce a single exponential algorithm for k-spanheight using $O(n^2 9^n)$ time and $O(n^2 2^n)$ space. This algorithm is based on the bucket-assignment technique from Feige [17]. Attempting to find an FPT algorithm, we argue that the graph minor theorem and Courcelle's theorem cannot be used to proof the FPT membership of spanheight. Instead we proof it to be FPT when restricted to graphs of bounded treedepth. Finally we present an $O(nt^4 \cdot 2^{7t^3 \cdot 2^t \cdot \log t})$ time and $O(n \cdot 2^{3t^3 \cdot 2^t \cdot \log t})$ space FPT algorithm by parameterizing on the treedepth t of a graph.

A second problem we study is restricted-spanheight, which is a special case of spanheight where the DFS spanning tree is restricted to edges from the input graph. For restricted-spanheight we provide similar results to spanheight. As a side result we look at reconfiguration of DFS spanning trees. The results in this thesis are mainly of theoretical significance, because the presented algorithms are very slow.

Acknowledgements

I would like to thank Hans Bodlaender for introducing me to the field of exact and parameterized algorithms. For suggesting spanheight as a topic for this thesis, and helping me find great literature. For the countless discussions and comments on early versions of algorithms. I want to thank all members of the treewidth study group for the TACO day, and the weekly talks on complexity proofs, FPT algorithms, and other topics like measure and conquer. Thank you for listening to my presentation on an early version of this thesis, and giving valuable comments.

Finally, I would like to thank my parents Eugène and Miranda, for everything.

Contents

Chapter 1

Introduction

Fomin, Heggernes, and Telle [20] introduce the graph property treespan. Their motivation comes from graph based search games, which strongly resemble well known graph problems. A search game consist of a single fugitive, and multiple cops. The fugitive is invisible, and is caught when a cop stands on the same vertex of the graph, and the fugitive has no vertex to escape to. In a standard search the cops move much slower than the fugitive. The problem of finding the minimum number of cops for which there exist a guaranteed strategy to catch the fugitive is the equivalent of finding a minimal path-decomposition of the graph. This is trivially true when the cops start searching at a vertex in the root bag, and walk to a vertex in the child bag, such that every vertex in the child bag is occupied by a cop. They continue this process until all bags are searched. The fugitive cannot reach the upper side of the path-decomposition without walking on the same vertex as a cop, by the definition of a path-decomposition.

In a special search version called Inert or Lazy search, the fugitive is only allowed to move just before the cops are going to walk to his vertex. This version enables the graph to be decomposed into a tree structure, because branches can be searched independently. The minimum number of cops for this type of problem is equal to the treewidth of a graph.

A different optimization objective, minimizes the number of turns a cop spends on each individual vertex. This is called the occupation time of a vertex, and the problem is equivalent to the bandwidth problem. The bandwidth problem is equivalent to finding an optimal path-decomposition with the following properties:

- The number of bags that contain a vertex v, is bounded by the bandwidth b
- For each pair of adjacent bags, exactly one vertex is introduced and/or forgotten, such that $|X_1 \setminus X_2| = 1$.

The main interest of Fomin et al was to find the graph property that is equivalent to a Lazy Search version of the bandwidth problem, or equivalently, a Lazy Search minimizing the Occupation time. This problem is an extension of bandwidth to trees, and equivalent to finding an optimal tree-decomposition with the following properties:

- The number of bags that contain a vertex v, is bounded by the treespan k
- For each pair of adjacent bags, exactly one vertex is introduced and/or forgotten, such that $|X_1 \setminus X_2| = 1$.

They call this search problem treespan.

	Number of Searchers	Occupation Time
Standard Search	pathwidth	bandwidth
Lazy Search	treewidth	treespan

Fomin et al. also proof certain properties for treespan of specific graph classes, and show that treespan is NP-Complete on cobipartite graphs. Rautenbach [25] presents lower bounds for treespan based on the chromatic number, connectivity number and the ratio between edges and vertices of a graph.

In [16], Dregi defines adjacencyspan, as the structural graph problem equivalent to the graph search problem treespan. The equivalence follows from the fact that both problems are equivalent to the same restricted type of tree-decomposition of a graph.

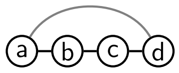


Figure 1.1: A mapping of a graph G to the interval [1..4] with bandwidth 3. The grey edge denotes that $\{a, d\}$ are adjacent in G.

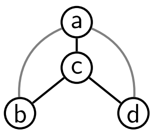


Figure 1.2: A DFS spanning tree of Figure 1.1, where the vertex b is placed in its own branch, reducing the distances between adjacent vertices. The grey edges are back-edges.

In this thesis, we have a different viewpoint on bandwidth, and a different extension to trees. A solution to the bandwidth problem is a mapping of the vertices $v \in V$ to the interval $[1..n]$. Such that the distance between any pair of adjacent vertices, in this interval, is at most b (see Figure 1.1). This interval can be seen as a DFS spanning tree consisting of a path, and we formulate the following question. Can we create branches on this path, to decrease the bandwidth? We formally define this as spanheight:

Spanheight
Instance: An undirected connected graph $G = (V, E)$, and a positive integer k.
Question: Can we add edges to E and get the graph $H = (V, E')$, where H is called a supergraph of G. Does there exist a DFS spanning tree $T = (V, F, r)$ of H in which every back-edge in $E' \setminus F$ spans at most k vertices?

For the bandwidth solution in Figure 1.1, it is shown in Figure 1.2, that allowing branches will decrease the maximum distance between the adjacent vertices. Therefore, we believe that spanheight is a natural extension of bandwidth to trees. The spanheight problem has a relation to tree-decompositions. In fact the spanheight problem is equivalent to finding an optimal tree-decomposition with the following properties:

- The sub-tree, induced by the bags containing v, has height bounded by the spanheight k.
- For each pair of adjacent bags, exactly one vertex is introduced and/or forgotten, such that $|X_1 \setminus X_2| = 1$.

These definitions allows us to compare treespan and spanheight. It is trivial that a solution for treespan is also a solution for spanheight, and therefore treespan upper bounds spanheight.

Spanheight strongly resembles the treedepth of a graph, which also searches DFS spanning trees. The objective function of treedepth is to minimize the distance between the root and every leaf vertex. Therefore, both problems minimize the distance between pairs of vertices on depth first search spanning trees. However, spanheight is much harder because its objective function is local to every vertex.

Our motivation to research spanheight comes from the observation that spanheight seems to be an extension from treedepth to something closer to treewidth. And since both treedepth and treewidth have interesting practical applications, maybe spanheight will as well. The research objective of this thesis is to find fast algorithms for spanheight. In this search we will be using the bandwidth, treewidth and treedepth relations to spanheight.

A second problem we study is restricted-spanheight, which is a special case of spanheight where the depth first search spanning tree is restricted to edges from the input graph. For restricted-spanheight we provide similar results to spanheight. A good reason for studying both problems is to study the effect of this restriction on the complexity.

Restricted-spanheight
Instance: An undirected connected graph $G = (V, E)$, and a positive integer k.
Question: Does there exist a DFS spanning tree $T = (V, F)$ of G in which every back-edge in $E \setminus F$ spans at most k vertices?

The complexity class Fixed Parameter Tractability (abbreviated FPT) is concerned with finding faster algorithms for NP-Complete problems by solving the problem for a restricted set of instances. This restricted set of instances must admit a certain property, which is called the parameter. The concept is that we design an algorithm, that is exponential in the size of the parameter, and polynomial in the length of the input. This gives a running time of $O(poly(n) \cdot f(k))$, where $f(k)$ is allowed to be an exponential function in the parameter k. The function $f(k)$ is often in the form of 2^k, but larger exponential functions are allowed. Transferring the exponential dependency from 2^n to 2^k is beneficial when k is much smaller. A requirement of FPT algorithms is that k is a constant and independent of n. Because k is a small constant, the running time of an FPT algorithm is polynomial with a hidden constant (exponential) factor.

Kernelization is a method to reduce the size of the input from n to $O(poly(k))$ for some parameter k. This effectively creates an FPT algorithm if this reduction takes polynomial time (See [15], chapter 4-5). Assuming the AND-distillation conjecture holds, there does not exist a polynomial kernel for the graph property treewidth. The similarity between treewidth and spanheight makes us assume that same type of argument is applicable to spanheight, and we will not consider Kernelization in this thesis. However, in [6] treewidth is shown to have a kernel on graphs of bounded vertex cover number, and this approach may be used to create a kernel for spanheight as well.

Dynamic programming on a tree-decomposition of the input graph is another well researched method to create FPT algorithms (See [15], chapter 10-12,15). The benefit of this approach is that it works for a lot of problems. For example, the problems Independent set and Hamiltonian cycle are under strong assumptions not in FPT, but the versions restricted to graphs of bounded treewidth are in FPT. For most local graph problems there does not seem to exist a trivial single exponential time 2^k FPT algorithm on treedecompositions. Cygan et al. [14] show that a small group of locality problems do admit single exponential probabilistic algorithms. In [9, 7] Bodleander et al. show that these problems also admit single exponential deterministic algorithms using representative sets. Under the Strong Exponential Time Hypothesis assumption, we can find lower bounds on the minimum complexity of FPT algorithms parameterized by treewidth (see [23]). These lower bounds are useful to determine of known algorithms are optimal.

Dynamic programming on a tree-decompositions requires computing a tree-decomposition beforehand. Computing a tree-decomposition of bounded treewidth can be done in linear time Bodlaender [3]. The tutorial [4] by the same author summarizes upper bounds and lower bounds between treewidth and related problems, and lists results on special graph classes. There are a number of FPT algorithms on tree-decompositions using depth first search. In [5] the depth first search algorithm is used to detect circus graph minors, creating an FPT algorithm for a special case of graph minor tests. [15, chapter 15] expands on this, and shows how to use depth first search as a parameterized algorithm design tool using the Plehn-Voigt Theorem. The DFS spanning tree problem treedepth admits an $O^*(2^{t^2})$ time algorithm on a tree-decomposition [26].

There are many other graph properties that can be used as a FPT algorithm design tools. For the bandwidth problem, the only known FPT algorithm parameterizes by the vertex cover number s of the graph (Fellows et al. [18]). When s is small, than all other $O(n)$ vertices of the graph can be categorized into at 2^s categories. This is than used to create an algorithm that focuses on the number of vertices per category, instead of each vertex individually.

In [16, chapter 7.2], Dregi introduces an FPT algorithm for treespan with a running time of $O(s^{2+s}4^s n + (s2^s)^{2.5(s+1)2^s + o(s2^s)})$, where s denotes the size of a vertex cover of the graph. This algorithm can be trivially changed to output a solution for the spanheight problem, and therefore we informally claim that spanheight is FPT for graphs of bounded vertex cover number. Their algorithm is based on the similar algorithm for bandwidth from [18]. It enumerates all structures on the vertex cover, and inserts the other categorized vertices using an FPT version of the Integer Linear Programming problem, bounded by the number of categories 2^s. The completeness and correctness of their algorithm is questionable, because they only give a rough sketch of the ILP model. However, being familiar with the original FPT algorithm for bandwidth, we are able to recognize that a more refined ILP model does exist. The complexity analyses for this algorithm contains small mistakes. Most notably, the algorithm defines $O(s \cdot 2^{s+1})$ zones, and every zone is assigned one of the $O(2^s)$ vertex categories. The algorithm must exhaustively enumerate all possible combinations, giving a total of $O((2^s)^{s\cdot 2^{s+1}})$ iterations. However, their analyses incorrectly states that they enumerate at most $O(s4^s)$ different combinations, possibly because they thought the number of combinations was multiplicative.

In chapter 5 of the same paper, Dregi present an XP algorithm, by observing that graphs of bounded treespan, have bounded degree of $2k$. Note that this degree bound does not translate to spanheight. In chapter 7.1, a correct $O^*(s^n)$ algorithm is given for treespan. Then incorrectly, they obfuscate this into a FPT algorithm for treespan restricted to graphs of bounded treespan k and bounded vertex cover number s. Their method consist of using the degree bound to rewrite s^n to the equivalent running time s^{s2k} because $s2k \geq n$.

1.1 Organization of this thesis

In Chapter 3 we look at the reconfiguration on DFS spanning trees, which seems to be useful for heuristic purposes and not for exact algorithms for spanheight. In Chapter 4 the relationship between spanheight and treedepth, treewidth and bandwidth is explored.

In Chapter 5 we proof NP-Completeness for both versions of spanheight with a reduction from treedepth. Then we focus on finding fast exact algorithms. First a single exponential algorithm for graphs of bounded spanheight is given in Chapter 6. This algorithm is strongly based on the bucked-assignment algorithm for bandwidth. The same algorithm is then adapted to solve restricted-spanheight.

In order to get an exact sub-exponential algorithm we proof that our problems are Fixed Parameter Tractable. Chapter 7 and 8 look at meta-theorems for proving membership in FPT. First we consider the graph minor theorem and proof that it does not hold for spanheight. Then in Chapter 8 we use Courcelle's theorem to find FPT instances of both problems. Chapter 9 presents a linear time FPT algorithm for restricted-spanheight on graphs of bounded restricted-spanheight. Chapter 10 is about a linear time FPT algorithm for spanheight on graphs of bounded treedepth.

Chapter 2

Preliminaries

2.1 Definitions

(Graphs). We will use $G = (V, E)$ to define an undirected graph. We will use n to denote the size of the set of vertices $|V| = n$. The graph G is allowed to be disconnected, meaning that it consist of multiple disconnected components, unless it is specifically stated that G is connected. The set V contains all vertices of G. The set E contains all undirected edges $\{v, w\} \in E$ of G between any pair of vertices $v \in V$ and $w \in V$ such that $v \neq w$.

(Edge and vertex sets). We will often use $E(G)$ and $V(G)$ to reference to the edge and vertex set of a graph $G = (V, E)$. The notation is also used for paths and a trees. This limits the number of variables we have to define and the reader has to remember.

(Path). A path P between a vertex v and w is an ordered sequence of vertices including the endpoints. The length of a path can be expressed as $|P|$ and equals the number of vertices on this path including the endpoints. When the path between two points is exclusive, the endpoints are not on the path.

(Trees and rooted trees). Unrooted trees are acyclic graphs and are defined by the tuple $T = (V, F)$, where V is a set of vertices, F is a set of tree-edges. In this thesis we will mostly use rooted trees unless specified otherwise. Let $T = (V, F, r)$ be a rooted tree, where $r \in V$ is the root vertex.

(Tree-Path). Let $T = (V, F)$ be a tree, a tree-path P is a path between using vertices $V(P) \subseteq V(T)$ that uses only tree-edges $E(P) \subseteq E(T)$.

(Height of a rooted tree). The height of a vertex v in a rooted tree is the maximum length of a path from v to a leaf vertex. Therefore, the height of a leaf equals 1. The height of a rooted tree equals the height of the root vertex.

([property-] decomposition). We use "decomposition" to denote a structure created from a graph G, such this structure admits to the specifications of the given graph property. For example:

- A tree-decomposition is a decomposition of a graph into a tree structure of treewidth at most tw.
- A path-decomposition is a decomposition of a graph into a path of pathwidth at most tw.
- A bandwidth-decomposition is a decomposition of a graph into an interval graph of bandwidth at most b.
- A spanheight-decomposition is a decomposition of a supergraph into a depth first search spanning tree of a spanheight at most k.
- A restricted-spanheight-decomposition is a decomposition of a graph into a depth first search spanning tree of a spanheight at most k.

2.2 Depth first search spanning tree

Let $G = (V, E)$ be an undirected and connected graph where V is a set of n vertices and E is a set of undirected edges. An example of such a graph G is given below in Figure 2.1. A rooted spanning tree of a graph G is a rooted tree that contains all the vertices from the graph and a subset of the edges. Let $T = (V, F)$ be a rooted spanning tree of the graph G using all vertices in V and let $F \subseteq E$ be a subset of the edges of the original graph G as displayed in Figure 2.2.

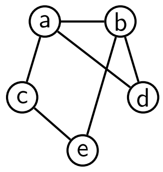


Figure 2.1: An example of a graph $G = (V, E)$.

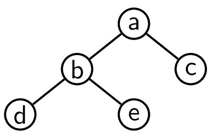


Figure 2.2: An example of a rooted spanning tree of G.

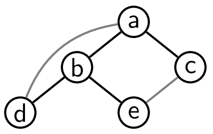


Figure 2.3: The same spanning tree, with the non-tree edges added in a grey color.

A DFS spanning tree is a special type of spanning tree in which we classify the non-tree edges, which are edges from the original graph G, that are not included in the spanning tree itself. We classify each non-tree edge as either a back-edge or a cross-edge. In Figure 2.2 we have not displayed the non-tree edges, but in Figure 2.2 we added all the non-tree edges of the graph G using grey colored edges.

For the non-tree edge $\{a, d\}$, the vertices a is a descendant of d, we call this non-tree edge a back-edge going back up in the tree. The non-tree edge $\{c, e\}$ is different, the vertex c is not an ancestor of the vertex of e because it is contained in a different branch of the spanning tree. This type of non-tree edge is called a cross-edge because it crosses between tree-branches.

Of these two types of non-tree edges, a DFS spanning tree forbids the existence of cross-edges. As a consequence all non-tree edges must be back-edges. The spanning tree from Figure 2.2 and 2.3 is therefore not a DFS spanning tree. A DFS spanning tree of the graph G is displayed in Figure 2.4. Every graph of n vertices admits at least n different DFS spanning trees by picking different roots. Computing a DFS spanning tree of a graph can be done in $O(|V| + |E|)$ time by the famous depth first search algorithm that enumerates all vertices in a pre-fix manner.

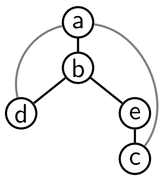


Figure 2.4: A DFS spanning tree of the graph from Figure 2.1.

For a back-edge between two vertices a and c. Let P path between a and c. We say that vertices $P \setminus \{a, c\}$ are spanned by the back-edge. We will also reference to these vertices as below the back-edge.

DFS spanning trees have many applications, e.g. solving puzzles, finding the connected components of a graph, and finding a path between two vertices in a graph. There are also many algorithms that use the depth first search algorithm to solve hard problems heuristically, e.g. heuristics for the constraint satisfaction problem, and for job scheduling problems.

2.3 Treewidth

The graph property called treewidth measures how much a connected undirected graph resembles a tree-structure. It defines a tree-decomposition as a mapping of an arbitrary connected (cyclic) graph $G = (V, E)$ onto a special type of unrooted tree $(\{X_i | i \in I\}, \mathcal{T} = (I, F))$. The tree $\mathcal{T}$ is build with numbered bags, with numbers $i \in I$. The set of edges between the numbered bags are represented as the set F. Each bag $i \in I$ of the $\mathcal{T}$ contains a subset of vertices $X_i \subseteq V$. A tree-decomposition has the following properties:

- $\bigcup_{i \in I} X_i = V$, every vertex is contained in the tree-decomposition.
- For every edge $\{v, w\} \in E$ of the original graph, there exist a bag $i \in I$ containing both v and w, or equivalently $(\forall v, w \in V)((\{v, w\} \in E) \to (\exists i \in I)(v, w \in X_i))$.
- For every vertex $v \in V$, the set of bags containing it $\{X_i |\ v \in X_i,\ i \in I\}$, form a connected sub-tree of $\mathcal{T}$.

An example of a tree-decomposion is given in Figure 2.5. The treewidth of a tree-decomposition is the size of the biggest bag minus 1 or equivalently $\max_{i \in I} |X_i| - 1$. The treewidth of a graph depends on the density/sparsity, a single vertex has treewidth 0, and a tree has treewidth 1.

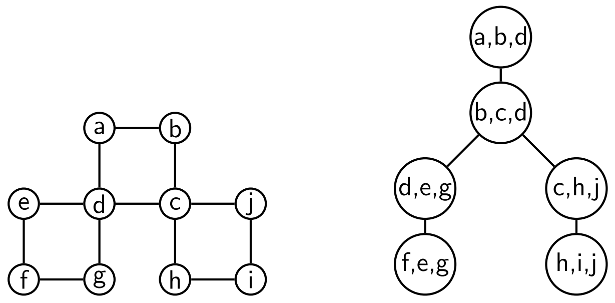


Figure 2.5: A graph and its tree-decomposition of treewidth 2.

Deciding if there exist a tree decomposition with tree-width at most k of a graph is NP-Complete [2], but the problem is shown to be in FPT and solvable in linear time and linear space but with a constant exponential factor.

A lot of NP-Hard problems have a lower order of complexity when the input graph G is a tree structure. However, an algorithm that only solves a problem for trees is not very useful in practice. Tree-decompositions are more practical than general trees because it allows us to exploit treelike properties of cyclic graphs.

A path-decomposition is a special case of a tree-decomposition, where no join nodes are allowed. As a consequence, it will always be a path.

2.3.1 Nice tree-decomposition

The operations on a tree-decomposition naturally divide into four separate operations: leaf, introduce, forget, and join. Instead of detecting which operations apply to each bag of the tree-decomposition we transform the tree-decomposition in what is called a nice tree-decomposition. In a nice tree-decomposition the bags have the following properties.

- Every bag has zero, one or two children.
- Leaf bags contain only a single vertex.
- A join bag X, and its two children X' and X'' contain the exact same vertices $X = X' = X''$.
- An introduce bag X introducing the vertex v has a single child X', where $X \setminus \{v\} = X'$.
- A forget bag X forgetting the vertex v has a single child X', where $X = X' \setminus \{v\}$.

2.4 Treedepth

The graph property called treedepth measures for a graph G what the smallest height of any DFS spanning tree is when we are allowed to create new edges. The decision problem is NP-Complete on bipartite and cobipartite graphs [8].

TREEDEPTH
INSTANCE: An undirected connected graph $G = (V, E)$ and a positive integer k.
QUESTION: Can we add edges to E and get the graph $H = (V, E')$. Does there exist a DFS spanning tree $T = (V, F)$ of H with height at most k?

Graphs of low treedepth look like stars, with the root vertex as the center. It has a number of practical applications that have generated interest in the problem. It has been introduced multiple times using different names and definitions:

- as minimum elimination tree by Pothen in 1988;
- as ordered coloring by Katchalksi et al. in 1995;
- as vertex ranking by Bodlaender et al. in 1998;
- as treedepth by Nešetřil and Ossona de Mendez in 2008.

It has some interesting relations to the other problems in this thesis, it upper bounds both the spanheight and the treewidth of a graph (Chapter 4). We will create an FPT algorithm for spanheight by parameterizing on the treedepth of a graph. The version of treedepth without edge creation does not have any published research to our knowledge. Therefore, we introduce it in Chapter 5 as part of an NP-Completeness proof.

2.5 Bandwidth

BANDWIDTH
INSTANCE: An undirected connected graph $G = (V, E)$ and a positive integer b.
QUESTION: Does there exist a function f that maps vertices $v \in V$ to the interval $[1..n]$, such that for every edge $\{v, w\} \in E$ the distance $|f(v) - f(w)|$ between the vertices is at most b?

Chapter 3

DFS Spanning Tree Reconfiguration

In our search for an algorithm for restricted-spanheight we discovered an DFS spanning tree reconfiguration operator. While not used in any of our algorithms, it is an interesting side result. The transformation operator has the ability to make small local changes to a DFS spanning trees. Furthermore, the operation can be used to turn any tree T into a DFS spanning tree with small local changes to remove cross-edges, such that the main structure of T can be maintained. We will show that this operator can be used to create any DFS spanning tree of a graph in a polynomial number of steps. Applications of our reconfiguration operator are heuristic and local search algorithms for problems on DFS spanning trees.

Graph transformation A graph transformation transforms the structure of a graph G into a graph H. There exists a good summary on the topic by W. Goddard and H. Smart in "Distances between graphs under edge operations", 1997 [22]. While we are interested DFS spanning trees instead of graphs we will use similar notations and proofs.

Suppose $\xrightarrow{\varepsilon}$ denotes a symmetric nonreflexive binary relation on the graphs. Then we say that graph G can be transformed into graph H in k steps by ε if there exists a sequence $G = G_0, G_1, G_2, ..., G_k = H$ of graphs such that $G_i \xrightarrow{\varepsilon} G_{i+1}$ for $0 \leq i \leq k-1$. The distance $\delta_\varepsilon(G, H)$ between G and H is the minimum value of k such that G can be transformed into H in k steps by ε, if such k exists; otherwise the distance is defined to be ∞. Not all pairs of graphs have finite distance k for every function ε. For example, G cannot be transformed into H when they have a different number of vertices/edges and ε does not have the ability to introduce new vertices/edges.

Geometric non-crossing spanning tree transformation In geometric graphs there exists the problem of transforming a spanning tree T with non-crossing edges (planar) to T'. Non-crossing denotes that no pair of edges of the spanning tree cross each other given the euler coordinates of the vertices. The transformation under the planer restriction allows any edge to be created between any pair of vertices, which again is a little different from our goals, but the analyses of their planar restriction is interesting. An upper bound is given in "A quadratic distance bound on sliding between crossing-free spanning trees" (2006) by O. Aichholzer and K. Reinhardt [1]. The proof uses the edge operation called "slide triangle" $\xrightarrow{ST}$ which moves the edges along a triangle on 3 points. This satisfies the non-crossing property when the triangle has no interior vertices. They then show that every non-crossing spanning tree can be transformed to a x-monotone non-crossing spanning tree in $O(N^2)$ steps using $\xrightarrow{ST}$, and the symmetrical property of $\xrightarrow{ST}$ allows for transformations in the general case. For $\xrightarrow{ST}$ they prove that this upper bound is tight by giving an example case that requires $O(N^2)$ steps.

The previous transformations do not apply to DFS spanning trees because they require that an edge can be created between any pair of vertices. And in DFS spanning trees we can only use edges from the original graph. Furthermore, the previous transformations only affect the new and old endpoints of a single edge; while an operation on a DFS spanning tree can effect all other vertices of the graph as well.

Definition (Tree-edge introduction $\xrightarrow{\sigma}$). Let $\xrightarrow{\sigma}$ denote a symmetric nonreflexive binary relation on DFS spanning trees. The function σ takes an back-edge $\{x, y\}$ that spans the path P in the DFS spanning tree T and introduces it as a tree-edge in the new DFS spanning tree T'. This creates a cycle in T', which we break by setting the parent of every internal vertex of P to their child in P in the new tree T'. As a result of the tree-edge introduce, the vertices from P are subject to cross-edges with other vertices in T'. Iteratively, pick the highest cross-edge from T', and introduce it using $\xrightarrow{\sigma}$, until all cross-edges are removed from the tree.

This process is illustrated in Figure 3.1. The proposed method preserves the old sub-tree as much as possible.

Lemma 3.1. *A tree-edge introduction $\xrightarrow{\sigma}$ can be computed in $O(n^3)$ time.*

Proof. Let $T = (V, E, r)$ be a tree with n vertices. Let a tree-edge introduction $\xrightarrow{\sigma}$ be performed on the cross-edge $\{v, w\}$ where $\text{depth}(v) = i$ and $\text{depth}(w) = j$. Let a be their lowest common ancestor vertex. Adding $\{v, w\}$ as a tree-edge of T creates a cycle through a. Assume w.l.o.g. that the vertex v has a greater depth than w. The cycle is broken by making every vertex on the path P between v and a an descendant of w. We make two observations:

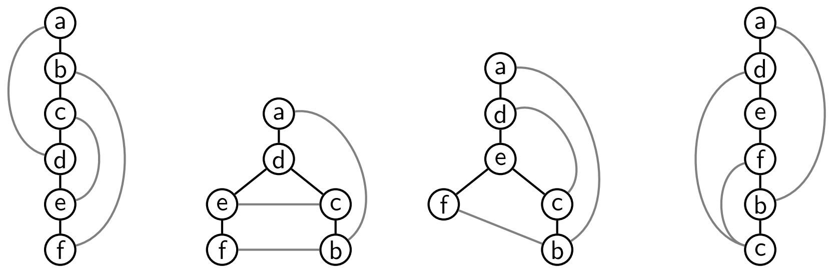
- There exist a vertex u on the path P with the same depth as w, and as a consequence of the tree-edge introduction, its depth increases. We conclude that there are at most $O(n)$ vertices at depth j in T, and the number of vertices at depth j decreases by 1, after each cross-edge removal incident to a vertex at depth j.
- Only the vertices in the path P are moved at each cross-edge removal. Any new cross-edges created as a side effect, must be incident to P. And the vertices incident to these new cross-edges must have been descendants v. We conclude that new cross-edges are created between only vertices of greater depth than j

The algorithm iteratively removes cross-edges starting at depth 1, until the lowest leaf node with depth at most $O(n)$ is reached. At each depth i, the algorithm introduces cross-edges using the $\xrightarrow{\sigma}$ operation, until one of the following cases is reached:

- There is only 1 vertex at depth i left, and it cannot have cross-edges to vertices at depth i or lower.
- There are multiple vertices at depth i left, and they have no cross-edges to vertices at depth i or lower.

Every depth takes at most $O(n)$ steps, and it takes $O(n)$ time to process a step, giving a $O(n^3)$ algorithm. □

Our definition of the transformation $\xrightarrow{\sigma}$ uses a top-down ordering of the cross-edges to avoid re-introducing cross-edges after they have been removed. This allows us to bound the number of steps by $O(n^2)$ with $O(n)$ work per step. Does this upper bound also holds for an arbitrary



(a) before (b) iteration 1 (c) iteration 2 (d) end

Figure 3.1: Introducing $\{a, d\}$ as a tree-edge with $\xrightarrow{\sigma}$. After the first step, this creates crossedges, which are removed in a top down approach.

ordering of the cross-edges? We believe this to be true, because for every arbitrary tree T we tested, even an arbitrary ordering seems to remove all cross-edges in at most $O(n^2)$ steps. We have unable to find a formal proof to answer this question.

A graph transformation is of limited usefulness when it can only search a subset of the search space of DFS spanning trees. The following theorem shows that our transformation can reach the complete search space.

Theorem 3.2. *The distance between a pair of arbitrary DFS spanning trees is bounded by $O(n)$ steps of the tree-edge introduce $\xrightarrow{\sigma}$ operation.*

Proof. Let $T = (V, E, r)$ be the DFS spanning tree that we transform into the DFS spanning tree $T' = (V, E', r')$. Iterate the vertices of T' in prefix ordering. For every vertex $v \in V$, let $w \in V$ be the parent, and introduce the edge $\{v, w\}$ in T as tree-edge using $\xrightarrow{\sigma}$. Once w is a parent of v in T, then this remains this way, because new cross-edges are created only below v. After all vertices are iterated, T is transformed into T', because the parent-child relations are equivalent. □

Chapter 4

Properties of Spanheight

The spanheight problem allows edges to be added to the graph in order to get a new graph with lower spanheight. In Figure 4.1 we illustrate a DFS spanning tree with spanheight 2, which trivially is optimal. Once we allow the edge $\{a, c\}$ to be created, the spanheight of the new graph is reduced to 1 as illustrated in Figure 4.2.

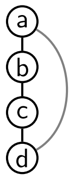


Figure 4.1: An optimal DFS spanning tree without edge creation.

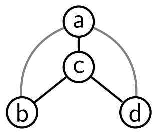


Figure 4.2: A DFS spanning tree with the created edge between a and c of the graph from Figure 4.1.

Any solution for restricted-spanheight is also a solution to spanheight, and therefore the spanheight of a graph upper bounds a restricted-spanheight of the same graph. In the introduction it was also shown that treespan upper bounds the spanheight. A notable trivial case for spanheight is trees, because they do not contain back-edges, and are already depth first search spanning trees. They have a spanheight zero.

4.1 Relation to bandwidth

Spanheight directly resembles the bandwidth problem, because they both restrict the distance between adjacent vertices in their respective output structures. The difference is that the output structure of bandwidth is a path, and a DFS spanning tree in the case of spanheight. The difference in output structure makes us to believe that there is not a strong relation between the two problems. Clearly bandwidth is an upper bound on the spanheight of a graph, because every bandwidth decomposition of bandwidth b is also a DFS spanning tree. The NP-Hardness cases are also different; bandwidth is NP-Complete on tree structures, while spanheight is easy on tree structures. The relation to bandwidth that we exploit in this thesis are the methods used to design algorithms for bandwidth.

4.2 Relation to treewidth

Lemma 4.1. *Let $T = (V, H, r)$ be a DFS spanning tree rooted at r of G with spanheight k, then there exist a tree-decomposition $(\{X_i | i \in I\}, \mathcal{T} = (I, F))$ of G with treewidth $k + 1$.*

Proof. We transform T into a tree-decomposition and prove that the three properties of a tree-decomposition are satisfied. The first step is to create a bag $i_v \in I$ for every vertex $v \in V$. Then we copy the tree structure of T by creating an edge $\{i_v, i_w\} \in F$ if and only if the corresponding edge $\{v, w\} \in H$ exist in T. And for every vertex bag i_v, let X_{i_v} be the set containing the first $k + 2$ vertices on the path from v to the root r, inclusive.

Cleary every vertex from G is contained in T and therefore has an associated bag i_v in the tree-decomposition. The bags that can contain a vertex v form a connected sub-graph rooted at the bag i_v. We have left to show that every pair $v, w \in V$ of adjacent vertices in G, are contained together in a bag of $\mathcal{T}$. The vertices v, w must form an ancestor/descendant relationship in T and the path P connecting them has length at most $k + 2$ by definition. Let v be the lowest vertex of the two, then the bag X_{i_v} contains every vertex on P including w by construction. This proofs the last property. $\square$

A consequence of this lemma is that the spanheight of a graph upper-bounds the treewidth of a graph.

4.3 Relation to treedepth

The maximum distance between any pair of vertices in a treedepth-decomposition, is t. This is also a spanheight-decomposition of spanheight t, and it follows that the treedepth is an upper bound on spanheight of a graph.

Later in this thesis we will use the following property to create an FPT algorithm for spanheight.

Lemma 4.2. *Let $T = (V, F, r)$ be DFS spanning tree of a connected undirected graph $G = (V, E)$ with spanheight at most k and with edge creation $F \nsubseteq E$. Let q be the maximum length of any path in G. The height of T is at most $(q - 1) \cdot k + q$.*

Proof. Let $v \in V$ be a leaf of T. Let P be a path from v to r in G. Observe that any pair of adjacent vertices on P, have distance at most $k + 1$ between each other in T, or the spanheight is larger than k. Using structural induction, starting at the leaf v in T. We travel on T to reach the next vertex u on the path P, which has distance at most $k + 1$ from v. Continue this procedure until the root r of T is reached. At every step of this procedure, we move only up or down in the tree because there do not exist cross-edges in T. If we move only up, then we visit at most $(|P| - 1) \cdot k + |P|$ vertices. This proves that the maximum distance between any leaf v and the root r is a function of the spanheight in combination with the maximum length of any path in G. $\square$

This upper bound can also be expressed in terms of the treedepth of a graph.

Corollary 4.3. *Let $T = (V, F, r)$ be DFS spanning tree of a connected undirected graph $G = (V, E)$ with spanheight at most k and with edge creation $F \nsubseteq E$. Let t be the treedepth of G. The height of T is at most $(2^t - 1) \cdot k + 2^t \leq t2^t$.*

Proof. Nešetřil and Mendez [24, formula 6.2] show that for graphs of bounded treedepth t, the maximum length q of any path in G, is upper bounded by 2^t. In the formula from Lemma 4.2 we substitute q with 2^t. Recall that treedepth upper bounds the spanheight and substitute k with t. Observe that $t \geq k + 1$, and we simplify $(2^t - 1) \cdot k + 2^t < 2^t \cdot (k + 1) \leq 2^t \cdot t$. $\square$

Chapter 5

NP-Completeness

In this section we will present a NP-completeness proof for the spanheight problem with a reduction from treedepth. But first we show that treedepth when restricted to edges from the original graph remains NP-hard. This proof is analog to the NP-hardness proof for the normal version of treedepth (Or equivalently vertex ranking) in [8].

Restricted-treedepth
Instance: An undirected connected graph $G = (V, E)$, and a positive integer k.
Question: Does there exist a DFS spanning tree $F = (V', E', r)$ of G with height lower or equal to k, such that $V' = V$ and $E' \subseteq E$?

We will show that this problem reduces to the balanced complete bipartite supgraph problem (abbreviated BCBS). The BCBS problem is the NP-complete [21, GT24] and defined as follows:

Balanced complete bipartite supgraph
Instance: A bipartite graph $G = (V, E)$, and a positive integer k.
Question: Are there two disjoint subset $W_1, W_2 \subseteq V$ such that $|W_1| = |W_2| = k$ and such that $u \in W_1, v \in W_2$ implies that $\{u, v\} \in E$?

Theorem 5.1. Restricted-treedepth *on cobipartite graphs is reducible to* BCBS *on bipartite graphs.*

Proof. Let $G = (V_1, V_2, E)$ be a bipartite graph of n vertices and let k be a positive integer. Together they form an instance to the BCBS problem. We claim that G contains a balanced complete bipartite graph using $2 \cdot k$ vertices, if and only if the cobipartite graph G' created from G admits a DFS spanning tree restricted to edges from G' of height $(n - k + 2)$.

Let $W_1 \subseteq V_1$, $W_2 \subseteq V_2$ with $|W_1| = |W_2| = k$ be a solution to the BCBS problem on G. Let $G' = (V_1', V_2', E')$ be the cobipartite graph created from G such that $V_1' = V_1$, $V_2' = V_2$, and create edges $e \in E \leftrightarrow e \notin E'$. We create a DFS spanning tree for G' as illustrated in Figure 5.1: Create an universal vertex a with two branches/subtrees below it. One branch is a path using the vertices from W_1 and the other branch is a path using the vertices from W_2. These paths can be created without creating new edges because by definition of G' the vertices contained in W_1 and W_2 form cliques in G'. This also does not create cross-edges because by definition of G' there does not exist an edge from any vertex $u \in W_1$ to $v \in W_2$ in G'. By definition the vertices from $V_1' \setminus W_1$ from a clique and we create a path using these vertices above the vertex a. Then we add a new universal vertex b as root, and finally add a path using the vertices $V_2' \setminus W_2$ above b. The tree is a long path ending the vertex a, with two branches below it of length k, giving a tree height of $(n - k + 2)$.

Let F be any DFS spanning tree of G', then F resembles the structure described above. Let b be the root of F with a as a child of b, and below a two branches. One branch is a path using the vertices from V_1 and the other a path using the vertices from V_2. Let this be the left tree from Figure 5.2. Observe that vertices contained in V_1 and V_2 form a clique in G', and therefore they can never create new sub-branches in F without creating cross-edges. Therefore, vertices can only move up and down while re-configuring the tree. As a consequence, any cross-edges

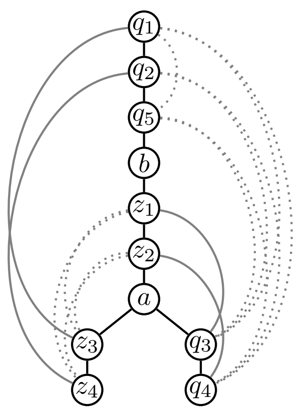


Figure 5.1: A DFS spanning tree of a cobipartite graph with $V_1 = \{z_1, z_2, z_3, z_4\}$, and $V_2 = \{q_1, q_2, q_3, q_4, q_5\}$. The edges of the cliques V_1 and V_2 are represented by dotted edges, the edges with endpoint in V_1 and V_2 are represented by a solid grey edge, and the tree-edges are drawn using a solid black color. The edges of the two universal vertices a and b are not drawn for increased clarity.

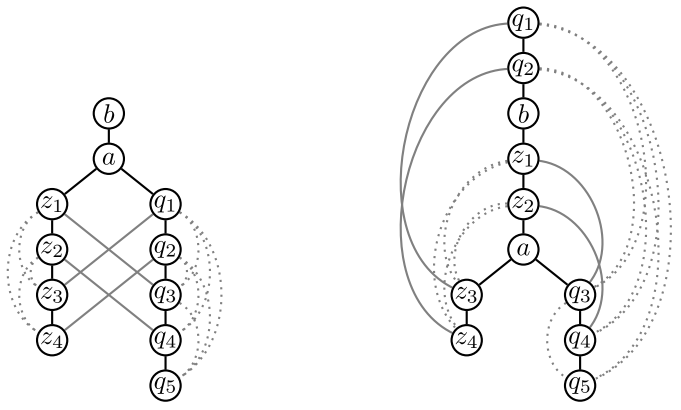


Figure 5.2: Two spanning trees of a cobipartite graph G' as defined in Figure 5.1. The vertices forming cross-edges in the left tree are moved up into the tree on the right.

between the branches below a must be removed by moving a vertex from below a to above a. Let the right tree of Figure 5.2 be an example where all cross-edges are removed. We conclude that every DFS spanning tree of G' consist of a path ending in some vertex x, with two branches below x.

Let F' be any arbitrary DFS spanning tree of G' with treedepth $(n - k + 2)$ with $k \geq 1$, then as proven its structure consist of a path ending in some vertex x with two branches below it. As proven one side of the branch can only use vertices W_1' from V_1 and the other side uses the vertices W_2' from V_2. Then W_1' and W_2' form a complete bipartite sub-graph of G by definition of G'. If $|W_1'| > |W_2'|$, then the treedepth is $(n - |W_1'| + 2)$, and we can take any subset of $W_1'' \subseteq W_1'$ such that $|W_1''| = |W_2'| = k$, and they form a balanced complete bipartite sub-graph in G. □

Corollary 5.2. Restricted-treedepth *is NP-Complete on cobipartite graphs.*

Proof. The NP-hardness on cobipartite graphs follows directly from the reduction in Theorem 5.1 to the BCBS problem which is known to be NP-Complete from [21]. To prove NP-Completeness we have left to show that there exist a polynomial time certificate testing the validity of a solution. Let $T = (V', F, r)$ be DFS spanning tree of trepdepth k of the undirected connected graph $G = (V, E)$. First compute for each vertex in T the height by iterating them in post-fix order. If the height of the root vertex is lower or higher than k then the treedepth is satisfied. To test if T is a valid DFS tree we must test whether T contains all vertices from G, is acyclic and connected, but also that is does not contain cross-edges. These tests are trivially done in polynomial time. We conclude that there trivially exist such an certificate proving the corollary. □

Theorem 5.3. RESTRICTED-SPANHEIGHT *is reducible to* RESTRICTED-TREEDEPTH *on arbitrary graphs.*

Proof. Let $G = (V, E)$ be a an undirected connected graph of n vertices and let k be a positive integer. Together they form an instance to the treedepth problem. We claim that G admits a DFS spanning tree of height k, if and only if the graph G', constructed from G by adding a gadget, admits a DFS spanning tree of spanheight $k - 1$.

Let $T = (V, F, r)$ be a DFS spanning tree of G with treedepth k, for an example see the left tree in Figure 5.3. Add a set Z containing $(2 \cdot k + 1)$ new vertices to G to get the new graph G'. In addition add the edge-set $(\{z_i, z_j\} | z_i, z_j \in Z, 1 \leq i < j \leq |Z|, j - i < k)$ and make z_{k+1} an universal vertex in G' as illustrated in Figure 5.3. We claim that any DFS spanning tree T' of G' with spanheight $k - 1$ must contain a DFS spanning tree T of G with treedepth k as a sub-tree, with as a consequence the correctness of the theorem.

This claim follows from the fact that for every vertex $z_i \in Z$, the parent in T' is z_{i+1}, except the highest two $z_{2 \cdot k}$ and $z_{2 \cdot k+1}$. The vertices from G are required to have distance at most $k + 1$ from the universal vertex z_{k+1}. Since z_{k+1} already has a parent, the vertices from G are either ancestors or descendants of z_{k+1}. When we do not allow edge creation, the vertices from G have no edges to the ancestors of z_{k+1}, and are therefore descendants. Even if we allow the creation of edges, placing any vertex between any pair of z_i and z_{i+1} is impossible, because them being a clique of size k requires them to form a path 7.1. Placing a vertex from G above the vertex $z_{2 \cdot k}$ will create a back-edge with spanheight at least k, which is higher than the allowed $k - 1$. We conclude that the vertices from G form a sub-tree of T' below z_{k+1} with height at most k, such that the leafs have distance at most $k + 1$ to the universal vertex z_{k+1}. By definition this subtree is a DFS spanning tree T with treedepth k of G.

We have left to show that for each vertex $z_i \in Z$ except $z_{2 \cdot k}$ and $z_{2 \cdot k+1}$, the parent in T' is required to be z_{i+1}. The reverse ordering is also possible w.l.o.g. The parent relation follows from the definition of the cliques, there are a cliques on the vertices with indices (1 to $k + 1$), (2 to $k + 2$), . . . , and (k+1 to $2 \cdot k + 1$). Each clique has size $k + 1$, and must form a path in the in the tree T'. Let i be the index of vertex z_i, and $j = i + 1$ of z_j. Then z_i and z_j share a clique and must have an ancestor-descendant relationship. Furthermore, at least 1 of the following holds:

- z_i is part of a clique ($i - k$ to i) with only vertices with index $< j$, every vertex of this clique must be on one side of j to be able to form path.
- z_j is part of a clique (j to $j + k$) using only vertices with index $> i$, every vertex of this clique must be on one side of i to be able to form path.

Therefore, there are only two orderings of the vertices with spanheight at most k. Ordered by decreasing index and by increasing index. The highest vertex $z_{2 \cdot k+1}$ is the only exception, because it is only part of one clique, it can move below z_{k+1} but this does not affect the placement and spanheight of any vertices. □

Corollary 5.4. RESTRICTED-SPANHEIGHT *is NP-Complete.*

Proof. The NP-hardness follows directly from the reduction in Theorem 5.3 to the RESTRICTED-TREEDEPTH problem which is proven to be NP-Complete on cobipartite graphs in Corollary 5.2. To prove NP-Completeness we have left to show that there exist a polynomial time certificate testing the validity a solution. Let $T = (V', F, r)$ be DFS spanning tree of spanheight k of the undirected connected graph $G = (V, E)$. First compute for each vertex in T the height by iterating them in post-fix order, and then test for each edge if the height difference is lower or equal to $k + 1$. The DFS property is trivially tested. We conclude that there trivially exist a certificate proving the corollary. □

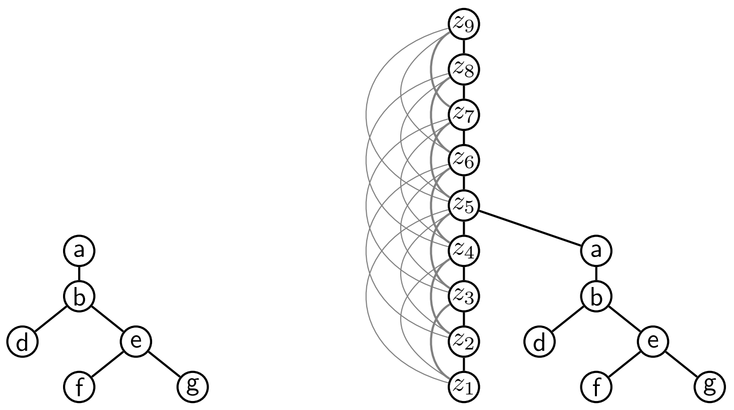


Figure 5.3: Left the DFS spanning tree of treedepth $k = 4$ of a graph G. On the right the DFS spanning tree of spanheight $k - 1$ the graph of G' created from G by adding the vertices z_i for $1 \leq i \leq 2 \cdot k + 1$, and the universal vertex z_{k+1}. The lighter grey edges are back-edges, not all back-edges of the universal vertex are not shown for clarity.

Corollary 5.5. SPANHEIGHT *is reducible to* treedepth *on arbitrary graphs.*

Proof. This follows directly from Theorem 5.3. Let T be a DFS spanning tree of G with treedepth k. The reduction creates a graph G' such that the DFS spanning tree T' of G' with spanheight $k - 1$ contains the tree T as a sub-tree. Allowing the tree T to use edges that do not exist in G will still allow us to use it as sub-graph of T' for the spanheight problem. Furthermore, Theorem 5.3 shows that even with edge creation the gadget still works as placing vertices above the universal vertex is impossible with respect to the spanheight. □

Corollary 5.6. Spanheight *is NP-Complete.*

Proof. The NP-hardness follows directly from the reduction in Corollary 5.5 to the TREEDEPTH problem which is proven to be NP-Complete on bipartite and cobipartite graphs in [8]. To prove NP-Completeness we have left to show that there exist a polynomial time certificate testing the validity a solution. Let $T = (V', F, r)$ be a DFS spanning tree with spanheight k of the undirected connected graph $G = (V, E)$. First compute for each vertex in T the height by iterating them in post-fix order, and then test for each edge if the height difference is lower or equal to $k + 1$. The other DFS properties are trivially tested. We conclude that there trivially exist a certificate proving the corollary. □

Chapter 6

Exact Algorithms

In this chapter we introduce exact algorithms for the decision problems spanheight and restricted-spanheight. Both problems can trivially be solved be performing an exhaustive search in the solution space. The size of the solution space for NP-Complete problems is so large that such an enumeration is intractable. For a lot of combinatorial problems there exist faster algorithms using dynamic programming techniques. These algorithms still require exponential time, but are much more efficient than an exhaustive search. The running time of these "efficient" algorithms is in the order of $O^*(c^n)$, where c is a small constant, preferably as low as possible.

The fastest known exact algorithm for treedepth uses $O(1.9602^n)$ time due to Fomin, Giannopoulou, and Pilipczuk [19]. Another variant called treedepth with capacity admits an $O(2.5875^n)$ algorithm [27].

In this chapter we define an single exponential time algorithm for both spanheight and restricted-spanheight. We will draw inspiration from the existing exact algorithms for bandwidth. In [17] Feige and Killian introduced a bucket-assignment scheme to solve bandwidth in $O^*(10^n)$ time and polynomial space. This algorithm has been improved by Cygan and Pilipczuk in [13] to get a $O^*(5^n)$ time and $O^*(2^n)$ space algorithm. Finally using measure and conquer the same authors found a $O^*(4.83^n)$ time and $O^*(4^n)$ space algorithm in [12].

6.1 Spanheight algorithm

Our algorithm will be an adaptation of the second fastest algorithm by Cygan and Pilipczuk. It consists of two phases. First we assign vertices to segments.

Definition (Segments and layers). Let $T = (V, E, r)$ be rooted tree. Vertices in this tree have a fixed height. A layer at height i contains all vertices of height i. The height of a layer is counted from the bottom of the tree to the top. The height of layer is used to reference to adjacent layers. In Figure 6.2 the layers are drawn using horizontal lines, and height is displayed at the right side.

A segment represents a group of $k + 2$ layers. In Figure 6.2 the tree is partitioned in 3 segments covering all layers of the tree. The size of a segment is chosen as $k + 2$ such that any back-edge in the tree lies within a segment, or between two neighboring segments. If not, then this back-edge must span at least $k + 2$ vertices, which is not allowed in our solution space.

The main concept of the algorithm is to first partition all vertices into segments. And then find the position of these vertices within the segment in a second phase of the algorithm.

- Partition the interval $[1, 2n]$ in $\lceil \frac{2n}{k+2} \rceil$ segments. The segments form a line, with a parent-child relations ship between every pair of consecutive segments.
- Assign a vertex $v \in V$ to the middle segment.
- Iteratively, take a vertex $v \in V$ that is not assigned to a segment, but has at least one neighbor u that is assigned to a segment. The placement of v must be in the in the same segment as u, or one of the two connecting segments. If not then the back-edge between them spans at least 1 segment with a path using $k + 1$ vertices, which violates the spanheight constraint. If there is no valid placement for v, then this partial segment assignment will never lead to a solution. For every vertex there are 3 options which will be enumerated in an exhaustive search.

After the first phase there are at most $O(3^{n-1})$ assignments of vertices to segments. We observe that any assignment that uses more than $\lceil \frac{n}{k+2} \rceil$ segments is redundant and can be deleted, since any path on n vertices must fit in $\lceil \frac{n}{k+2} \rceil$ segments. The total computation time of the first phase is $O((n + |E|) \cdot 3^{n-1})$.

Note that our first phase avoids trying multiple initial positions for the first vertex, reducing the running of states generated by the first phase with a polynomial factor. This approach can possibly be used to reduce the hidden polynomial for the previously mentioned bandwidth algorithms.

Segment	Assigned vertices
1	a, b, c, d, f
2	w, j, k, p, g, r, z, l, s
3	m, u, t, n, o, v

Figure 6.1: An example of an assignment of n vertices to at most 3 segments

6.1.1 Depth first search over assignments

For each assignment generated in the first phase we apply the second phase in order to find a solution to the decision problem. If no assignment from phase 1 can be refined into a solution, then answer to the decision problem is negative.

We will first define how an assignment is refined. Recall that each segment is representative of a forest of height $k + 1$. Phase 1 tells us what vertices are in this forest, but not at which heights these vertices are positioned in this forest. This second phase of the algorithm is designed to assign the height to vertices. For the assignment in Figure 6.1, we have assigned a height to each of the vertices in Figure 6.2. In this figure, segment 1 contains the first $k + 2$ layers of the tree, then comes segment 2 with layers of height 4 to 6. After assigning vertices to layers we compute the DFS spanning tree using the following lemmas. First we show how this tree can be computed.

Lemma 6.1. *Let π be an assignment of vertices to layers. Let X be the set of vertices assigned to a layer at height i. Let W be the set of all vertices below this layer (including all vertices in lower segments). Let C be the set of connected components in $X \cup W$. This layer is not suspect to cross-edges if $(\forall C' \in C)|C' \cap X| \leq 1$.*

Proof. Let $|C' \cap X| > 1$ for a connected component $C' \in C$ and assume that there does not exist a cross-edge below any pair of vertices in $|C' \cap X|$. Observe that $C' \cap X$ contains at least two vertices (u, v), and they have a common ancestor $a \in V$. There exist a path between the

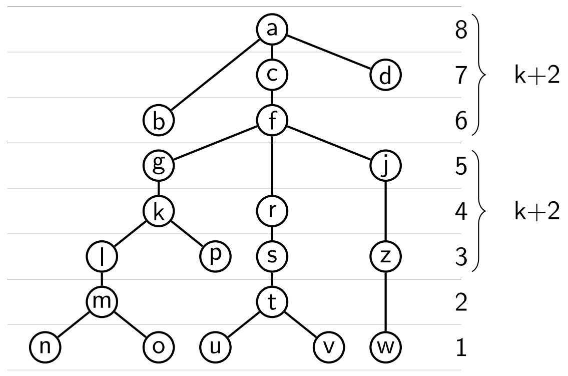


Figure 6.2: An assigment of vertices to layers within each segment, for the segment assignment in Figure 6.1. Back-edges are not drawn for clarity.

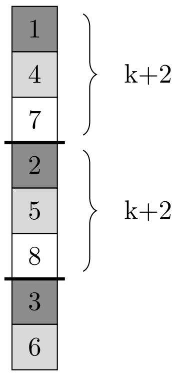


Figure 6.3: A colored ordering of the layers in all segments. The number gives the index of the layer in the color ordering.

vertices through a. There also exists a path between these vertices in the connected component C'. Combining these two paths creates a cycle. The cycle is broken by placing a part of the cycle below u, and possibly a part of the cycle below v. These two parts, including v and w admit at least one pair of vertices connecting into a cycle. These two vertices have a cross-edge between them. This contradicts the assumption. □

We call such a layer **suspect to cross-edges**. We have left to show that it is possible to create a DFS spanning tree when no layer in the assignment is suspect to cross-edges.

Lemma 6.2. *Let π be an assignment of vertices to layers that avoids layers suspect to cross-edges. Then a DFS spanning tree $T = (V, E, r)$ is computed by connecting every vertex $x \in V$ to the highest vertex in every connected component below x.*

Proof. Assume that there is a cross-edge in T between two vertices (u, v) and let a be their lowest common ancestor. As a consequence there exist a path P_1 from u to a and a path P_2 from v to a. Remove the vertex a from these paths, then there are two different cases in which a cross-edge can exist.

- No pair of vertices on P_1 and P_2 share a layer. Assume w.l.o.g. that the vertex v is placed on a lower layer than vertex u. Observe that u is the lowest vertex on P_1. Let W be set of vertices on P_2 that are on a lower layer than u. Let w_2 be the highest vertex in W. Clearly w_2 is in a connected component below u, and therefore w_2 is a child of u according steps defined in the lemma. Furthermore, u is an ancestor of v and this contradicts that the cross-edge exists.
- At least one pair of vertices (w_1, w_2) in P_1 and P_2 share a layer. If $w_1 = u$ and $w_2 = v$ then they form a connected component. Otherwise there is a connected component C below and including w_1 that contains u. And there is a connected component C' below and including w_2 that contains v. Because u and v are connected, the connected components must be the same $C \cup \{w_2\} = C' \cup \{w_1\}$. Lemma 6.1 shows that there exist a cross-edge below them as a consequence of this shared connected component. Our algorithm and the lemma will explicitly forbid states that contains a such layers that are suspect to cross-edges, contradicting that such an cross-edge can exist.

□

The key concept of the dynamic programming algorithm is the order in which layers are assigned vertices. In [13] the notion of a color ordering is introduced. Every layer in each segment is given a color index based on the position within the segment, from top to bottom. There are $k + 2$ different colors. In Figure 6.3 the pattern of assigning colors is visualized. The color ordering sorts all layers based on color first, and segment number second. The algorithm will assign vertices to layers in the color ordering.

Lemma 6.3. *Let π be an assignment of vertices to layers with spanheight at most k. If there is an edge $\{u, v\}$ between two segments, where u belongs to the segment of greater number, then $color(\pi(v)) > color(\pi(u))$.*

Proof. By definition the number of colors between a color i in segment 1 and i in segment 2 is $k + 1$, and this back-edge would have spanheight $k + 1$. Such an back-edge cannot exist if the structure created by π has spanheight at most k. □

Definition The state of the dynamic programming algorithm is a tuple (i, A), where i is the number of layers we have assigned a set of vertices, and $A \subseteq V$ is the set of vertices assigned to these layers. Every state must admit to the following properties:

- The vertices of A can be assigned to the first i layers of the color ordering. This layer assignment
  - contains no edge $\{u, v\}$ for $u \in A, v \notin A$ and v is assigned to a segment with a greater number than u;

- contains no layers suspect to cross-edges as per Lemma 6.1;
- is compatible with the segment assignment. Which means that all vertices assigned to a layer are also assigned to the segment containing that layer.

Lemma 6.4. *Let $(0, A_0 = \emptyset), (1, A_1), \cdots, (n, A_n = V)$ be a sequence of states with $Q_i = A_i \setminus A_{i-1}$. The function π assigns the set of vertices Q_i to the i^{th} layer of the color ordering, this gives a layer assignment admitting a DFS spanning tree of a supergraph with spanheight at most k.*

Proof. Assume (i, A_i) is a state for every $0 \leq i \leq n$. Then according to Lemma 6.2 we must be able to compute a DFS spanning tree. Lemma 6.3 shows that the edge restriction in the state is correct, because the unassigned endpoint in a lower segment can only be assigned at distance at least $k + 2$. □

The dynamic programming algorithm consist of computing every possible sequence of states. A depth first search approach is taken starting from the state $(0, \emptyset)$ until the final state is reached. Within the depth first search, let (i, A_i) be the current state. We extend the state with every possible set of vertices W such that $(i + 1, A_i \cup W)$ is a state. Once the final state is reached, it is possible to find the corresponding sequence of states by back-tracking on the depth first search stack. Lemma 6.4 guarantees that we can compute a DFS spanning tree with spanheight at most k from this sequence.

Clearly there are $O(n \cdot 2^n)$ states. All states are reached in the format $(i+1, A_i \cup W)$, giving at most $O(\sum_{i=1}^{n} \binom{n}{i} \sum_{j=1}^{i} \binom{n-i}{j}) = O(3^n)$ combinations of A_i and W. There are at most $O(n3^n)$ state transitions, for each transition the dynamic programming table is accessed in $O(n)$ time. Testing if a state can be extended with W is trivially be done in $O(n + |E|)$ time, this is only tested once for each state, giving $O(n^3 2^n)$ steps. Therefore the algorithm visits at most $O(n \cdot 2^n)$ states in $O(n^2 \cdot 3^n)$ time.

Theorem 6.5. *Let G be an arbitrary graph, deciding if G has spanheight at most k takes at most $O(n^2 \cdot 9^n)$ time and $O(n^2 \cdot 2^n)$ space.*

Proof. Phase 1 uses $O(n^2 3^n)$ time and $O(n + |E|)$ space. Phase 2 is executed for every segment assignment of phase 1, giving a $O(3^n)O(n^2 \cdot 3^n)$ algorithm. If phase 2 finds a valid layer assignment, then the corresponding DFS spanning is computed in $O(n + |E|)$ time. □

As mentioned in the prequel, the papers [13] and [12] use a deeper analyses and measure and conquer to get faster running times for bandwidth. Can the same techniques be used to improve Theorem 6.5 as well?

6.2 Restricted-spanheight algorithm

With a few adjustment the algorithm will work for restricted spanheight as well. For every layer assignment in phase 2. We need to guarantee that the computed tree uses only edges between two connected layers. Two layers are connected when they form a parent-child relationship. We also need to ensure that these edges exist in the original graph.

Definition The state of the dynamic programming algorithm is a quad (i, A, F, L), where

- i is the number of layers we have assigned a set of vertices using the color ordering;
- $A \subseteq V$ is the set of vertices assigned to layers;
- $F \subseteq A$ is the set of vertices assigned to the first layer of each segment;
- $L \subseteq A$ is the set of vertices assigned to the last assigned layer of each segment.

It is possible that the currently last assigned layer of a segment is the first layer in the segment, and that F and L overlap. Every state must admit to the following properties:

- The vertices of A can be assigned to the first i layers of the color ordering. This layer assignment
  - contains no edge $\{u, v\}$ for $u \in A, v \notin A$ and v is assigned to a segment with a greater number than u;
  - contains no layers suspect to cross-edges as per Lemma 6.1;
  - is compatible with the segment assignment. Which means that all vertices assigned to a layer are also assigned to the segment containing that layer.
  - For every vertex $v \in A$ assigned to a layer of height i, there must exist exactly one $w \in A$ with $\{v, w\} \in E(G)$ where w is assigned to the layer of height $i-1$, iff the layer of height $i-1$ is already in the layer assignment. (height references to the height in the final computed tree, and not the color ordering)
  - For a segment with vertices V_s, must assign the set $V_s \cap F$ to the first layer this segment.
  - For a segment with vertices V_s, must assign the set $V_s \cap L$ to the currently last assigned layer in this segment.

Lemma 6.6. *Let* $(0, \emptyset, \emptyset, \emptyset), (1, A_1, F_1, L_1), \cdots, (n, A_n = V, F_n, L_n)$ *be a sequence of states with* $Q_i = A_i \setminus A_{i-1}$. *The function* π *assigning the set of vertices* Q_i *to* i^{th} *layer of the color ordering, this gives a layer assignment admitting a DFS spanning tree with spanheight at most* k.

Proof. Assume (i, A_i, F_i, L_i) is a state for every $0 \leq i \leq n$. Then according to Lemma 6.2 we must be able to compute a DFS spanning tree. Lemma 6.3 shows for each i, any vertex $v \in Q_i$ has edges to vertices with distance at most $k+1$, if not than this contradicts that (i, A_i, F_i, L_i) is a state. As a consequence this computed DFS spanning tree has spanheight at most k.

We have left to show that the computed DFS spanning tree uses only edges from the original graph. When the vertices Q_i are assigned to layer i of the color ordering at height q in the tree, they have an edge to the layer with height $q-1$ and index j in the color ordering. These edges are introduced at the $\max(i, j)^{th}$ state of the sequence, if not then this contradicts that they are both states. To proof that these edges can actually be computed at the corresponding state, we give the expression for both cases. In the i^{th} state the edges are computed with $(\forall v \in Q_i, \exists w \in (L \cap V(\mathit{segment}(\mathit{layer}(i)))))(\{v, w\} \in E(G))$, and at the j^{th} state the edge are computed with $(\forall w \in (F \cap V(\mathit{segment}(\mathit{layer}(j)) + 1)), \exists v \in Q_j)(\{v, w\} \in E(G))$. □

The only difference to the algorithm in the previous section is the way layers connect, and therefore the way states are extended in the depth first search algorithm. Let (i, A_i, F_i, L_i) be the current state. Let $i+1$ be index of the next layer in the color ordering. This layer is contained in a segment with number s. Let V_s be the set of vertices in this segment. Let $L = V_s \cap L_i$ be the set of vertices assigned to the previous layer in this segment s. Take a subset of $W = V_s \setminus A_i$, and try to assign it to the layer $i+1$. Explicitly check if this will create a valid state. The new state will be $(i+1, A_i \cup W, F_{i+1}, (L_i \setminus L) \cup W)$ where $F_{i+1} = F_i \cup W$ for the first layer in each segment, and $F_{i+1} = F_i$ otherwise.

For this algorithm there are $O(\sum_{i=1}^{n} \binom{n}{i} \cdot 2^i \cdot 2^i) = O(5^n)$ combinations of A, F and L. The number of states is bounded by $O(n \cdot 5^n)$. The running time of the algorithm depends on the number of ways each state can be extended, which are all tried in an exhaustive search of the depth first search algorithm.

- For the first layer in each segment the state (i, A, F, L) where $A = F = L$ is reached from any state $(i-1, A_{i-1} \cup W, F_{i-1} \cup W, L_{i-1} \cup W)$, there are $O(3^n)$ ways we can create such combinations of A and W.

- Otherwise the state (i, A, F, L) with $A \neq F \neq L$ is reached from a state $(i-1, A_{i-1} \cup W, F_{i-1}, (L_{i-1} \setminus Z) \cup W)$, where $A_{i-1} \cup W$ admits $O(\sum_{i=1}^{n} \binom{n}{i} \sum_{j=1}^{n-i} \binom{n-i}{j})$ combinations, F_{i-1} admits $O(2^i)$ combinations, and $(L_{i-1} \setminus Z)$ admits $O(\sum_{q=1}^{i} \binom{i}{q} \sum_{r=1}^{i-q} \binom{i-q}{r})$ combinations. Combining all these factors gives $O(\sum_{i=1}^{n} \binom{n}{i} \sum_{j=1}^{n-i} \binom{n-i}{j} \cdot 2^i \cdot \sum_{q=1}^{i} \binom{i}{q} \sum_{r=1}^{i-q} \binom{i-q}{r}) = O(7^n)$.

The combined running time is $O(7^n + 3^n) = O(7^n)$.

Theorem 6.7. *Let G be an arbitrary graph, deciding if G has restricted-spanheight at most k takes at most $O(n^2 \cdot 21^n)$ time and $O(n^2 \cdot 5^n)$ space.*

Proof. Phase 1 uses $O(3^n)$ time and $O(n + |E|)$ space. Phase 2 is executed for every segment assignment of phase 1, giving a $O(3^n)O(n^2 \cdot 7^n)$ algorithm. If phase 2 finds a valid layer assignment, then the corresponding DFS spanning tree is computed in $O(n + |E|)$ time. □

Chapter 7

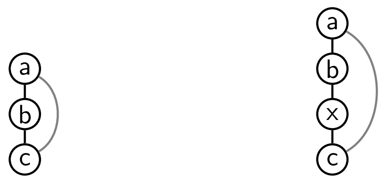
The Graph Minor Theorem

A graph minor relationship is shared between two undirected graphs G and H when it is possible to transform G into H using the following operations. An edge deletion deletes an edge from the graph. A vertex deletion deletes a vertex and all incident edges from the graph. An edge contraction with incident vertices (v, w) deletes both vertices from the graph including all edges incident to v or w. A new vertex u is added with an edge to all vertices that shared an edge with either v or w. Informally an edge contraction can be seen as the merging of two vertices.

A graph property is closed under the taking of minors, if for every graph G that admits this graph property, all minors of G also admit this graph property. The graph minor theorem by Roberson and Seymour is summarized in [15]. The theorem proofs that if a graph property is closed under the taking of minors, then there exist a finite set F of graphs called forbidden minors. If a graph H contains any member of the forbidden minors as a minor, it cannot be part of the graph family that admits the graph property. A consequence of this theorem is that if F can be computed, then there exist an FPT algorithm on graphs of bounded treewidth based on detecting the forbidden minors. In this chapter we show that both versions of spanheight are not closed under the taking of minors.

Lemma 7.1. *Let G be a graph containing a clique of 3 vertices $\{a, b, c\}$. Let T be a DFS spanning tree of G with spanheight 1. The vertices $\{a, b, c\}$ form a path in T.*

Proof. Assume that the clique $\{a, b, c\}$ does not form a path in T. Without loss of generality, let a be the highest vertex of $\{a, b, c\}$ in T, c the lowest, and b between them. The path between a and c contains b, our assumption that $\{a, b, c\}$ does not form a path requires that there is at least one other vertex x on the path between a and c. We observe that the back-edge $\{a, c\}$ spans over two vertices x and b, this contradicts that the spanheight of T is 1. □



Observe that the path using the vertices $\{a, b, c\}$ uses exactly 2 of the 3 edges of the clique.

Lemma 7.2. *Let G be a graph containing the cliques $\{a, b, c\}$ and $\{a, b, d\}$. Let T be a DFS tree of G with spanheight 1. The edge $\{a, b\}$ that is shared between the cliques must be a tree-edge of T.*

Proof. From Lemma 7.1 we know that T must contain paths using the vertices $\{a, b, c\}$ and $\{a, b, d\}$. Assume that the common edge $\{a, b\}$ is not used then for the first clique c must be on the path between a and b. The second path using only the vertices $\{a, b, d\}$ cannot exist because c lies on the path between a and b. This shows that it is impossible that T does not contain the shared edge. □

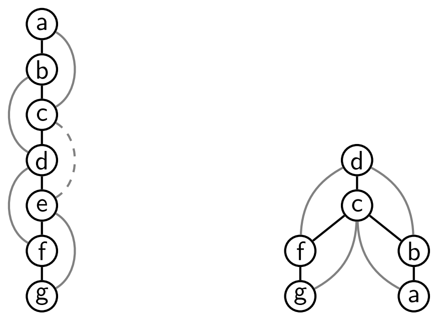
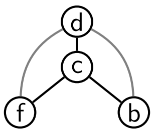


Figure 7.1: Contracting the edge $\{c, e\}$ and the unique resulting tree with spanheight 1.

Lemma 7.3. *The the graph G, obtained from the edge contraction of $\{c, e\}$ in the graph from Figure 7.1, admits exactly one DFS spanning tree of spanheight 1.*

Proof. We perform a case analyses of all DFS spanning trees for the graph G. Instead of focusing on vertices we enumerate all trees using the cliques after the contraction. The cliques are $\{a, b, c\}$, $\{b, c, d\}$, $\{c, f, d\}$, and $\{c, f, g\}$. From Lemma 7.2 we learn that the shared edges $\{b, c\}$, $\{c, d\}$, and $\{c, f\}$ must be tree-edges. Using the 3 edges we have either c as root with 3 children or another vertex as root, with c as child and 2 leaf nodes below c. Since d has edges to both f and b he cannot be a leaf node or it would create cross-edges to the other leaf(s), therefore we can only choose d as root:

Figure 7.2: The only valid sub-tree using tree-edges $\{b, c\}$, $\{c, d\}$, and $\{c, f\}$.



Notice that this sub-structure already contains the cliques $\{b, c, d\}$ and $\{c, f, d\}$. Next we introduce the clique $\{a, b, c\}$ intro the tree. The edge $\{b, c\}$ is already part of the tree and therefore we must either add $\{a, b\}$ or $\{a, c\}$ as an tree-edge. We cannot add $\{a, c\}$ because it would create a cross-edge between a and b. Adding $\{a, b\}$ does not create cross-edges nor cycles so we can add it.

The argument for introducing the clique $\{c, f, g\}$ is the same and must add the edge $\{g, f\}$. The resulting tree is the right tree from Figure 7.1. Note that this tree uses the 3 initial edges that we are required to use, and we have no choice which edges we added for the 2 last cliques. Which means that the position of every vertex in the tree is fixed and cannot be changed.

We must also consider if adding more edges can create alternative trees. There are 3 edges not present in the graph $\{g, d\}$, $\{a, d\}$, and $\{a, g\}$. The first two would create cliques of size 4, which cannot admit a tree with spanheight 1 and therefore we cannot add these edges to the graph. The edge $\{a, g\}$ would create a cross-edge and require us to move one or more vertices in the tree structure, but their position is fixed and therefore this is edge cannot be added to the graph as well. We conclude that we can only create one unique tree of spanheight 1 even if we allow the creation of additional edges in G. □

Theorem 7.4. *The graph family of spanheight 1 with or without edge creation is not closed under the taking of minors.*

Proof. From Lemma 7.3 we know that for the graph G from Figure 7.1, only one unique DFS spanning tree T with spanheight 1 exists. This tree requires that the vertex below the contracted edge becomes the new root of the tree. Take 2 of such graphs G and G' and glue them together with the edge $\{g, a'\}$ to get the graph G'' from Figure 7.3 . Obviously G'' has spanheight 1.

Contract the edges $\{c, e\}$ and $\{c', e'\}$ in G''. We assume G'' still has spanheight 1 and we build the tree T'' that witnesses this. Observe that T'' must contain the unique solutions T and T' for the subgraphs G and G'. There are two cases:

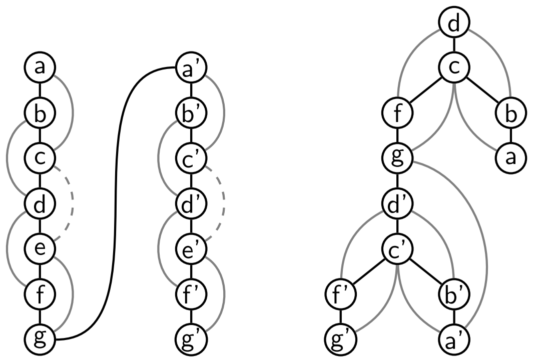


Figure 7.3: Two gadgets glued together, the contracted edges $\{c, e\}$ and $\{c', e'\}$, and the failed attempt to create a tree T'' with spanheight 1 of the contracted graph G''.

- We are not allowed to create edges. Assume T will have a higher position in T'' than T' w.l.o.g. The root of T' must become a child of one of the vertices of T, for this an edge must be created. This contradicts that T'' can be constructed.
- We are allowed to create edges. Assume T will have a higher position in T'' than T' w.l.o.g. The root d' of T' must become a child of the vertex g in T as illustrated in Figure 7.3. If not then there would exist a cross-edge between g and a'. The back-edge $\{g, a'\}$ spans 3 vertices, and this cannot be changed since all vertices have a fixed position as shown in Lemma 7.3. This contradicts that T'' with spanheight 1 can be constructed.

□

Chapter 8

Courcelle's Theorem

Let G be a graph, we want to decide if G belongs to the graph family with spanheight at most k. Courcelle's theorem as summarized in [15] states that this can be decided in linear time on graphs of bounded treewidth, if the graph family can be expressed in Monadic Second Order Logic (abbreviated MSOL). This result was independently rediscovered by Borie, Parker and Tovey in [10]. In this chapter we attempt to define spanheight in MSOL, in order to find a linear time algorithm.

The MSOL expression is a predicate calculus description of the graph family. In this chapter we will use a variation of MSOL called MSO_2. In MSO_2 we are allowed to describe the graph family of a graph $G = (V, E)$ using the following rules:

- Quantifying over universal subsets of vertices $\forall V' \subseteq V$ and subsets of edges $\forall E' \subseteq E$.
- Quantifying over existential subsets of vertices $\exists V' \subseteq V$ and subsets of edges $\exists E' \subseteq E$.
- Quantifying over universal and existential elements of sets $\forall v \in V$, $\exists e \in E$.
- Logic operators: negation $\neg$, equivalence $\leftrightarrow$ or $=$, disjunction $\vee$, conjunction $\wedge$, implication $\rightarrow$.
- Testing if a vertex $v \in V$ is incident to an edge $e \in E$, using the expression $inc(e, v)$.

It is forbidden to quantify over other sets $V' \not\subseteq V$ and $E' \not\subseteq E$. The length of each expression is required to be constant. For a graph family that admits a property bounded by an integer constant k, any expression of length $f(k)$ is considered constant and allowed. Expressions can be split into multiple parts called functions. The paper [10] provides example functions and expression to build upon.

8.1 Restricted-treedepth

There are multiple algorithms witnessing the membership in FPT of treedepth, but not for restricted-treedepth.

$$\exists F \subseteq E, r \in V$$
$$(\text{Rooted-DFS-Tree}(V, E, F, r) \wedge v \in V \rightarrow \text{Bounded-Distance}(v, r, V, F, t))$$

Where clearly every leaf has depth at most t, implied by the tree-path of bounded length t to the root vertex r. We shall now define the sub-definitions.

Rooted-DFS-Spanning-Tree(V, E, F, r) :=
$$(\text{Spanning-Tree}(V, F) \wedge \{v, w\} \in E \rightarrow \text{Ancestor}(V, F, r, v, w) \vee \text{Ancestor}(V, F, r, w, v))$$

Spanning-Tree(V,E) :=
$$\forall v \in V, \exists w \in V(\text{Acyclic}(V, E) \wedge \text{Connected}(V, E))$$

A cycle is a connected sub-graph where every vertex has degree two.

$$\text{Acyclic(V,E)} :=$$
$$\forall W \subseteq V((\forall v \in W)(\mathit{Degree}(V,E,v,2)) \rightarrow \neg \mathit{Connected}(W,E))$$

$$\text{Connected(V,E)} :=$$
$$\forall W_1, W_2 \subseteq V(\text{Partitioning}(W_1,W_2,V) \rightarrow (\,W_1 = \emptyset \vee W_2 = \emptyset \vee$$
$$(\exists v \in W_1, \exists w \in W_2)(\{v,w\} \in F)))$$

$$\text{Partitioning}(W_1,W_2,V) := (W_1 \cap W_2 = \emptyset \wedge W_1 \cup W_2 = V)$$

The set E_1 containing only edges incident to v, denotes the degree of this vertex. A degree of k implies that size of the set E_1 must be equal to k.

$$\text{Degree}(V,E,v,k) :=$$
$$\forall E_1 \subseteq E((\forall e \in E_1)(\mathit{Incident}(e,v)) \rightarrow \mathit{SizeEquals}(E,k))$$

If a path from a vertex v to a vertex r contains a vertex w, then this is an successor of v on this path. If r is the root of a tree, this successor is also an ancestor.

$$\text{Ancestor}(V,E,r,v,w) :=$$
$$\forall W \subseteq V(v \in W \wedge r \in W \wedge \mathit{Connected}(W,E) \rightarrow w \in W)$$

A set has size k, if and only if it contains exactly k unique elements.

$$\text{SizeEquals}(E,k) :=$$
$$(\exists q_1 \in E_1, \cdots, \exists q_k \in E_1)((\bigwedge_{j=2}^{k} \bigwedge_{i=1}^{j-1} q_i \neq q_j) \wedge \neg(\exists e \in E)(\bigvee_{i=1}^{k} e \neq q_i))$$

$$\text{SizeBounded}(E,k) := (\bigvee_{i=1}^{k} \mathit{SizeEquals}(E,i))$$

Two vertices have bounded distance, if at least one path of length k exist between them.

$$\text{Bounded-Distance}(v,r,V,E,k) :=$$
$$\exists W \subseteq V(v \in W \wedge r \in W \wedge \mathit{Connected}(W,E) \wedge \mathit{SizeBounded}(W,k))$$

8.2 Restricted-spanheight

$$\exists F \subseteq E, r \in V$$
$$(\text{Rooted-DFS-Tree}(V,E,F,r) \wedge \{v,w\} \in E \rightarrow \text{Bounded-Distance}(v,w,V,F,k+2))$$

8.3 Treedepth

It is well known that treedepth is in FPT, therefore the following expression is just an exercise. In the MSOL expression language we are not allowed to iterate edges that do not exist in the original graph. Recall that the treedepth problem is equivalent to the vertex ranking of a graph. Our MSOL expression maps each vertex v to a color i by placing $v \in H_i$.

$$\begin{aligned}
&\forall H_i \subseteq V for\ 1 \leq i \leq t,\ \forall u, v \in V\\
&(Partitioning(H_1, ..., H_t, V) \wedge (\neg EqualRank(u, v) \vee\\
&(\forall W \subseteq V)(v \in W \wedge u \in W \wedge Connected(W, E) \rightarrow\\
&(\exists q \in W)(HigherRank(q, u) \wedge HigherRank(q, v))))
\end{aligned}$$

Where

$$\begin{aligned}
EqualRank(u, v) &:= \bigwedge_{i=1}^{t} (u \in H_i \leftrightarrow v \in H_i)\\
HigherRank(u, v) &:= \bigwedge_{i=1}^{t} (u \in H_i \rightarrow \bigvee_{j=1}^{i-1} v \in H_j)
\end{aligned}$$

8.4 Spanheight

On a tree-decomposition, the path connecting a pair of bag-vertices needs to be represented in the state, in order to add new vertices on this path. If not, then the spanheight of these back-edges cannot be tested. The length of this path can be linear $O(n)$ in the size of the graph, and contains at most $n \cdot k$ back-edges. Every subset $O(2^{n \cdot k})$ of back-edges is a possible state of the algorithm. This exponential tendency in n makes it unlikely that there exist an FPT algorithm for spanheight on graphs of bounded treewidth.

Therefore, we will introduce an FPT algorithm on graphs of bounded treedepth instead. In Corollary 4.3 it was shown that treedepth can be used to upper bound the height of an spanheight-decomposition by $\sigma = t2^t$, where t is an upper bound on the treedepth of G. Our MSOL definition will build a vertex ranking of ranking at most σ, which is equivalent to depth first search spanning tree of height at most σ. The color i of every vertex directly denotes to the height i. Adjacent vertices have spanheight at most k if the difference between their heights is at most $k + 1$.

$$\begin{aligned}
&\forall H_i \subseteq V for\ 1 \leq i \leq \sigma,\ \forall u, v \in V(Partitioning(H_1, ..., H_\sigma, V) \wedge\\
&(\forall v, w \in V)(\{v, w\} \in E \rightarrow BoundedRankDifference(v, w, k + 1)) \wedge\\
&(\neg EqualRank(u, v) \vee (\forall W \subseteq V)(v \in W \wedge u \in W \wedge Connected(W, E) \rightarrow\\
&(\exists q \in W)(HigherRank(q, u) \wedge HigherRank(q, v))))
\end{aligned}$$

Where

$$\begin{aligned}
EqualRank(u, v) &:= \bigwedge_{i=1}^{\sigma} (u \in H_i \leftrightarrow v \in H_i)\\
HigherRank(u, v) &:= \bigwedge_{i=1}^{\sigma} (u \in H_i \rightarrow \bigvee_{j=1}^{i-1} v \in H_j)\\
BoundedRankDifference(u, v, k) &:= \bigwedge_{i=1}^{\sigma} (u \in H_i \rightarrow \bigvee_{j=i-k,\ s.t.\ 1 \leq j \leq \sigma}^{i+k} v \in H_j)
\end{aligned}$$

Chapter 9

Restricted-spanheight FPT Algorithm

We introduce an FPT algorithm using tree-decompositions solving to solve the decision version of restricted-spanheight in an exact manner. One of the most important concepts of this algorithm is to temporary relax the edge creation restriction, allowing the algorithm to build structures independent of vertex introduction order. The output of the algorithm is an restricted-spanheight decomposition of spanheight at most k, but sub-states will resemble spanheight decompositions of spanheight at most k. Note that for both the restricted and unrestricted version we call the number of vertices below a back-edge the spanheight. This follows from the fact that the restriction is on the structure, not on how the number vertices are computed below a back-edge.

The dynamic programming algorithm takes the triple $(G, \mathcal{T}, k)$ as input, where G is an undirected connected graph $G = (V, E)$, $\mathcal{T}$ a nice tree-decomposition of G with treewidth tw, and let k be an integer. The algorithm decides whether **restricted-spanheight**$(G) \leq k$ in $O(n \cdot 2^{(4tw^2+5tw) \cdot \log(k+2)} \cdot tw^2 \cdot k)$ time and $O(2^{(3tw+2tw^2) \log(k+2)})$ space. For a yes-instance the algorithm can be modified to output a witnessing restricted-spanheight-decomposition using back-tracking in the dynamic programming tables.

The algorithm is explained in three steps. The first step defines a way to solve the problem using partial decomposition on the bags of the nice tree-decomposition. The second step defines restricted partial decompositions, which uses properties of a tree-decomposition to limit the state to the bag vertices, with as consequence a reduction on the number of unique states in our dynamic programming tables. We then proceed to define and proof the operations off the algorithm.

9.1 Main algorithm

The algorithm works by creating tables of partial decompositions for each bag of the nice tree-decomposition using bottom-up dynamic programming techniques. Every step will consider a bag of the tree-decomposition, and for every bag the algorithm will maintain a state in a dynamic programming table. The state for each bag is computed using the state of child bag(s) in the tree-decomposition. A state consist of a set of partial decompositions.

A partial decomposition resembles a restricted-spanheight decomposition of the graph G, however it only covers a subgraph of G. On a tree-decomposition, the state of every bag $i \in I$ only covers the subgraph induced by the vertices in this bag, and all descendant bags in the tree-decomposition. Let $\mathcal{T}_i$ denote a sub-tree of the rooted tree-decomposition $\mathcal{T}$, rooted at the bag $i \in I$. Then $V(\mathcal{T}_i)$ is the set of vertices contained in the bag i and all descending bags, and $G[V(\mathcal{T}_i)]$ is the induced subgraph covered by partial decompositions in the bag i.

Definition (Partial decomposition). A partial decomposition is a rooted forest F with $V(F) \subseteq V(G)$ and $E(F) \nsubseteq E(G)$, where every tree $T \in F$ is a depth first search spanning tree with restricted-spanheight at most k.

Definition (Edge creation relaxation). For every partial decomposition, we allow tree-edges that do not exist in the original graph. This allows the algorithm to guess for pairs of non-adjacent vertices, that there exist a path in G connecting them. However, this path is not present in the subgraph covered in the current state of the algorithm. The concept is that when we create an edge between this pair, it will possibly be replaced with this path from G, in a later state of the algorithm. When an guessed edge is not replaced at the end of the algorithm then this is called an erroneous guess. Any partial decomposition that contains an erroneous guessed edge can never become restricted-spanheight decomposition and can be deleted from the state tables.

Definition (Spanheight of a Partial decomposition). The spanheight of a partial decomposition F is the maximum distance between any two vertices with a back-edge to each other.

As we start working on tree decomposition, we will be forgetting vertices from the partial decompositions, restricting the partial decomposition to a subset $X \subseteq V(\mathcal{T}_i)$ of vertices that it covers, while still maintaining all necessary information about the covered subgraph $G[V(\mathcal{T}_i)]$. Therefore, we need a to define a structure that removes these forgotten vertices but remembers their effect on the placement of new vertices.

Definition (Restriction of a partial decomposition). A restriction of a partial decomposition is a penta (F, B, l, s, Z) where,

- F is a forest of rooted trees with $V(F) \subseteq V(G)$ and $E(F) \subseteq V(F) \times V(F)$ and $E(F) \not\subseteq E(G)$.
- $B \subseteq V(F) \times V(F)$ is the set of edges between vertices in F that are either back-edges or can become back-edges in a future state. Therefore, B contains all tree-edges and back-edges from G with both endpoints in F. The set B also contains representative back-edges in F that do not exist in G.
- $l : E(F) \to \mathbb{N}^+$ is a function denoting the length of tree-edges. Normally every tree-edge has length 0. When a tree-edge represents a forgotten path of vertices, it has length $l(e)$ denoting that this forgotten path used $l(e)$ vertices. The length function $l(e)$ is clamped to the range $0 \leq l(e) \leq k+1$, because forgotten paths longer than k cannot have spanning back-edges.
- $s : V(F) \times V(F) \to \mathbb{N}^+$ is a function denoting the forgotten spanheight of back-edges in B. The forgotten spanheight is clamped to the range $0 \leq s(e) \leq k$ because any higher would not be a valid state.
- $Z \subseteq V(F)$ contains the of vertices in the forest F that have a forgotten parent, and no ancestor in F.

A restricted partial decomposition can be abbreviated by and referred to with "restriction", "equivalency class", and "class".

Definition (Forgetting a vertex). Let x be the forget vertex in the rooted forest F of the graph G. If there exist a created/guessed edge incident to x, then this must be an erroneous edge. Because by definition of a tree-decomposition there cannot exist a path in G that is not covered by F that connects to x. If such an erroneous edge exist, then the corresponding restriction must be deleted from the state. Note that if one of the neighbors of x is a forgotten vertex, then this forgotten vertex is a neighbor of x in G, otherwise it would not have been forgotten next to x.

Definition (Tree-edge length). After a forget operation we replace the forgotten vertex by creating an edge between its parent p and each one of its children $v \in V(F)$ as illustrated in Figure 9.1. Each new edge e has a length $l(e)$, denoting that the edge represents a path using $l(e)$ forgotten vertices. $l(e)$ equals to the length of the path between p and v before the forget operation $l(\{p, v\}) = l(\{p, x\}) + 1 + l(\{x, v\})$. Initially every tree-edge has length 0. Note that tree-edges with length higher than 0 represent a path in G, and are allowed to exist in the final partial decomposition even if the edge itself is not present in G.

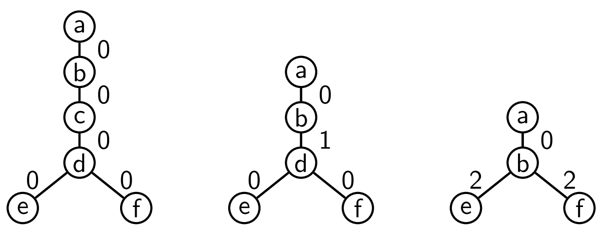


Figure 9.1: Tree-edge lengths before and after a forget operation of c, and d.

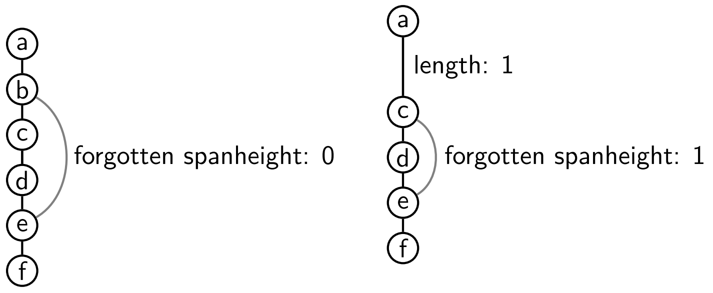


Figure 9.2: Computing a representative back-edge with forgotten spanheight.

Definition (Back-edge forgotten spanheight). Let b and e be two vertices with a back-edge of spanheight 2 as illustrated in Figure 9.2. Observe that on path P between them, we can insert at most $k-2$ new vertices. Let b now be forgotten and c the vertex next to b on the path P. We cannot insert a vertex between b and c after the forget operation. Therefore, we can insert at most $k-2$ new vertices on the path $P \setminus \{b\}$ between c and e. To enforce this we create the back-edge $\{c, e\}$ with spanheight 1. Notice that the new back-edge has a lower spanheight than the original and as a result we can insert more vertices to P than the original back-edge allowed. To counter this we remember how much we shortened P by removing endpoints as the forgotten spanheight of a back-edge. The spanheight of an back-edge now becomes the sum of the length of the path below it and its forgotten spanheight. In our example the forgotten spanheight is 1 to account for the vertex b that was forgotten above the endpoint c, and the spanheight of the new edge $\{c, e\}$ becomes $1 + 1 = 2$. If new back-edge already $\{c, e\}$ exists, then it may have a different forgotten spanheight can be different. We store the highest forgotten spanheight of the two and discard the lower one.

Definition (Restricted partial decomposition spanheight). The spanheight of a back-edge $e \in B$ is the length of the tree-path P spanned by the back-edge plus the forgotten spanheight $s(e)$. The spanheight of a restricted partial decomposition is the maximum spanheight of all back-edges in this restriction.

Definition (Restricted partial decomposition equivalency). Two restricted partial decompositions (F, B, l, s, Z) and (F', B', l', s', Z') are equivalent when $F = F'$, $B = B'$, $l = l'$, $s = s'$, and $Z = Z'$.

Definition (Forgetting a tree-root). Let (F, B, l, s, Z) be a partial decomposition. In order to iterate every possible tree the algorithm allows there to exist multiple trees in the forest F. After forgetting a tree root, the tree can split into multiple trees. Each of these sub-trees has a new root $v \in V(F)$. It is impossible to add new vertices above the new tree roots because they are only allowed one parent which is forgotten. The set Z stores all tree-roots in F of which the parent is forgotten. This allows the algorithm to respect the single parent rule.

9.2 Operations

The algorithm exists of computing the state for each bag, using a specific function for each bag type of the nice tree-decomposition.

```
Algorithm 1: Restricted-spanheight on graphs of bounded treedwidth
input  : The triple $(G, \mathcal{T}, k)$, where $G = (V, E)$ is a connected undirected graph,
         $\mathcal{T} = (X, F)$ is a nice tree-decomposition of $G$, and $k$ is an integer upper bound of
         the restricted-spanheight of $G$.
output: True if there exist a restricted-spanheight-decomposition for graph $G$ of
         spanheight at most $k$, and False otherwise.
1 Let $X_r$ be the root bag of $\mathcal{T}$
2 Let $R$ be the set containing all restricted partial decompositions of the root bag computed
  with spanheight-recursive$(G, \mathcal{T}, X_r, k)$.
3 To enforce that the edge creation relaxation is lifted, delete from $R$ every restriction
  $(F, B, l, s, Z)$ that contains two adjacent vertices in the forest $F$ that are not adjacent in
  $G$ if and only if the tree-edge between them has length zero.
4 if $R \neq \emptyset$ then
5     return True
6 else
7     return False
```

```
Algorithm 2: Restricted-spanheight-recursive
input  : The quad $(G, \mathcal{T}, X_i, k)$, where $G = (V, E)$ is a connected undirected graph,
         $\mathcal{T} = (X, F)$ is a nice tree-decomposition of $G$, $X_i$ is a bag of the
         tree-decomposition $\mathcal{T}$, and $k$ is an integer upper bound on the
         restricted-spanheight of $G$.
output: A set $R$ containing all restricted partial decompositions on the sub-graph
         $G[V(\mathcal{T}_i)]$.
1 $R := \emptyset$
2 if $X$ is a leaf then
3     $R$ = Leaf$(X)$
4 else if $X$ is a forget bag then
5     Let $v$ be the forget vertex
6     Let $X'$ be the child bag of $X_i$.
7     $R'$ = spanheight-recursive$(\mathcal{T}, G, X', k)$
8     $R$ = Forget$(R', v)$
9 else if $X$ is an introduce bag then
10    Let $v$ be the introduce vertex
11    Let $X'$ be the child bag of $X_i$.
12    $R'$ = spanheight-recursive$(\mathcal{T}, G, X', k)$
13    $R$ = Introduce$(R', v, k)$
14 else if $X$ is a join bag then
15    Let $X_1$, and $X_2$ be the child bags of $X_i$.
16    $R_1$ = spanheight-recursive$(\mathcal{T}, G, X_1, k)$
17    $R_2$ = spanheight-recursive$(\mathcal{T}, G, X_2, k)$
18    $R$ = Join$(R_1, R_2, k)$
19 return $R$
```

We now define the operations on the bags of the nice tree-decomposition. We use a different operation depending on the type of bag. But first we define how we compute the spanheight of a restricted partial decomposition.

9.2.1 Computing the spanheight

Let (F, B, l, s, Z) be a restricted partial decomposition, we compute the spanheight of each back-edge $\{v, w\} \in B$ and take the maximum $\max_{e \in B} sp(e)$. Where we define the spanheight of an edge $sp(e)$ to be the expression:

$$\sum_{e \in E(P)} l(e) + |V(P)| - 2 + s(\{v, w\}) \tag{9.1}$$

Where:

- P is the tree-path from v to w in the forest F.
- $\sum_{e \in E(P)} l(e)$ is the sum of the length of the edges of P.
- $|V(P)| - 2$ be the number of vertices on the path P minus the endpoints.
- $s(\{v, w\})$ represents the forgotten spanheight.

9.2.2 Leaf bag

Create a class (F, B, l, s, Z) with $F = \{(V = \{x\}, E = \emptyset)\}$, $B = \emptyset$, $\forall e \in B(l(e) = 0 \wedge s(e) = 0)$, and $Z = \emptyset$. And put it in the set of classes for the current bag $R_i = \{(F, B, l, s, Z)\}$.

9.2.3 Introduce operation

Let $R \leftarrow \emptyset$ be the set of restricted partial decompositions of current bag, and let R' be set of the child bag. Let x be the introduce vertex. For each class $(F', B', l', s', Z') \in R'$ we take the forest F' and generate new forests by inserting x in locations that do not create cross-edges or edges between the disjoint trees of the forest. Places we consider are:

- As child of any vertex $p \in V(F')$ and inheriting none or a subset of its children. Let C be the set of children of p in F', for whom the tree-edge to their parent does not represent a forgotten path $\forall c \in C \rightarrow l(\{p, c\}) = 0$. The vertex x can take any subset of $C' \subseteq C$ as children such that x lies on the path from p to $c \in C'$. The vertex x can also have the tree-roots of other trees as its children. Let Q be the set of tree-roots in F whose tree does not contain p and have no forgotten parent $q \in Q \rightarrow q \notin Z$. The vertex x can take any subset $Q' \subseteq Q$ as children, including the empty subset.
- As the root of a tree, with tree-roots of other trees as its children. Let Q be the set of tree-roots in F that have no forgotten parent $q \in Q \rightarrow q \notin Z$. The vertex x can take any subset $Q' \subseteq Q$ as children, including the empty subset.

The number of places we can insert the new vertex is exponential $O(tw2^{tw})$ in the number of vertices in F' because we can pick any subset of children for any parent. To finish the operation we add all edges from G incident to x and a vertex $v \in V(F)$ to B' to get B. The variables l', s' and Z' remain unchanged. For each placement j we compute the class (F_j, B, l', s', Z') and put it in R if the spanheight of the computed class is lower than or equal to k. If the class (F_j, B, l', s', Z') is already in R then we don't have to add it again.

9.2.4 Forget operation

Let $R \leftarrow \emptyset$ be the set of restricted partial decompositions of the current bag, and let R' be set of the child bag. Let x be the forget vertex. For each class $(F', B', l', s', Z') \in R'$ we take the forest F' and delete the vertex x and all back-/tree-edges incident to x and get the class (F, B, l, s, Z) using the following procedure:

If x has a neighbor in the tree F' that is not adjacent to x in the original graph G, and the edge between them is not a forgotten path, then this is a erroneous guessed edge. In this case we will not create a restriction of this class for the the current bag. Otherwise compute the functions l and s of the restriction using the following sub-algorithms:

Compute the replacing tree-edges For every child v of x we create and edge to the parent w of x and set its length to match the length of the forgotten path using x. The length of the forgotten path $l(\{w, v\})$ equals $l'(\{w, x\}) + l'(\{x, v\}) + 1$.

Compute the replacing back-edges As a consequence of deleting x, its back-edges will also be deleted from B. For each vertex from the current bag $v \in W$ that has an back-edge to x create a new back-edge representing the old back-edge as shown in Figure 9.2:

- Let w be the first vertex on the tree-path P from x to v in F', exclusive. If there is no such vertex, or $v = w$, then stop because it is not necessary to represent the old back-edge since it is impossible to insert new vertices below it.
- Otherwise we compute the forgotten spanheight:
  - The vertex x and the edge $\{x, w\}$ is removed from P after the forget operation. Therefore the path P has reduced length $l(\{x, w\}) + 1$.
  - The forgotten spanheight of the new representing back-edge equals the forgotten spanheight of the original plus the reduced length of the path below it, this gives the expression $s(\{x, v\}) + l(\{x, w\}) + 1$. The represented back-edge $\{v, w\}$ is computed even though this back-edge may already exist in B. If it exist then it may already have a forgotten spanheight, and we pick the maximum $s(\{w, v\}) = \max(s(\{w, v\}), s(\{x, v\}) + l(\{x, w\}) + 1)$.

If x is a tree-root and contained in Z, then remove it from Z. Every child b of x becomes a tree-root after deleting x. Add b to Z to denote it has a forgotten parent. The spanheight cannot increase in a forget operation, therefore add the new class to R if it is not already.

9.2.5 Join operation

Let $R \leftarrow \emptyset$ be the set of restricted partial decompositions of the current bag, and let R_1, R_2 be sets of the two child bags. For any pair $(F_1, B_1, l_1, s_1, Z_1) \in R_1$, $(F_2, B_2, l_2, s_2, Z_2) \in R_2$ we merge them if the following holds:

- $F_1 = F_2$.
- $\forall e \in E(F)(l_1(e) = 0 \vee l_2(e) = 0)$, any tree-edge in the new forest can only represent one forgotten path. If both F_1 and F_2 have a forgotten path on any edge $e \in E(F)$ than they can never merge.
- $Z_1 \cap Z_2 = \emptyset$, only one side of the join may have a forgotten path/vertex above a tree-root.

Merge the partial decompositions and get (F, B, l, s, r), where

- $F = F_1 = F_2$.
- $B = B_1 \cup B_2$. Each child can have its own restricted back-edges.
- $\forall e \in E(F)(l(e) = \max(l_1(e), l_2(e)))$. Taking the maximum edge length in forest 1 and 2 is sufficient because one of them is guaranteed to be zero.
- $\forall e \in B(s(e) = \max(s_1(e), s_2(e)))$. The forgotten spanheight uses different forgotten vertices in forest 1 and forest 2. We need to handle the largest one and the smaller one will be satisfied implicitly.
- $Z = Z_1 \cup Z_2$.

Finally compute the spanheight of the new partial decomposition if it is lower or equal than k, and add it to R unless R already contains it.

9.3 Correctness of Algorithm

The correctness of the algorithm depends on the following properties:

- Partial decompositions can be used to solve the decision problem on a tree-decomposition.
- A restricted partial decomposition representation is equivalent to the original partial decomposition representation on a tree-decomposition.
- The defined operations for each bag-type of a nice tree-decomposition correctly compute all valid restricted partial decompositions.

As a consequence of these properties it follows using structural induction that for every spanheight decomposition, the tables of the algorithm compute its restriction. And that after processing the final table, every entry it contains is a restriction of an spanheight-decomposition of the graph. In this section we will proof that the algorithm admits these properties.

9.3.1 Properties of partial decompositions

The relation between an spanheight-decomposition and a partial decomposition is trivial.

Lemma 9.1. *Every spanheight-decomposition $T = (V, E)$ of the graph $G' = (V', E')$ has an equivalent partial decomposition F where T is the only tree in the forest F.*

It is essential that the algorithm can find every possible solution. To make this happen the edge creation relaxation was introduced.

Lemma 9.2. *The edge creation relaxation allows the creation of every spanheight-decomposition $T = (V, E)$ of the graph G on a nice tree-decomposition $T = (X, F)$.*

Proof. Let F be the partial decomposition equivalent to T. The introduce and join operation can be inverted to get a delete and separation operation. This allows the process to be inverted, starting with the final partial decomposition F, and moving down the nice tree-decomposition while decomposing F into smaller partial decompositions. After each introduce bag we delete the introduced vertex, and after at each join bag we delete the vertices that do not appear on the current side of the join. At each deletion we create a new tree-edge from the deleted vertex its parent to each of its children. This created edge may not exist in the original graph, but this is allowed under the edge relaxation in sub-steps. Once the leaf bag is reached the algorithm is finished. Each step can trivially be inverted, proving the lemma for a given spanheight-decomposition. □

9.3.2 Properties of restricted partial decompositions

Lemma 9.3. *Forgetting a vertex from a restricted partial decomposition on a tree-decomposition does not impact the creation of cross-edges.*

Proof. Let (F, B, l, s, Z) be a restricted partial decomposition of the graph G. Let a be a forgotten vertex that is already removed from F, and b a new vertex that we introduce in F. Observe that the forget bag of a is a descendant of the introduce bag of b. Assume the placement of the vertex b in F creates a cross-edge to the vertex a. The existence of an edge between a and b means that there exists a bag in the tree-decomposition containing both a and b, by definition of a tree-decomposition. With as consequence that the forget bag of a must be a ancestor of the introduce bag of b, this contradicts our observation. □

Lemma 9.4. *Let (F, B, l, s, Z) be a restricted partial decomposition of the graph G. At the forget operation on a tree-decomposition, if there exist a neighbor y of the forget vertex a such that they are not neighbors in G, and the tree-edge between them has length zero. Then the restricted partial decomposition is not a restriction of a spanheight-decomposition.*

Proof. The edge creation relaxation is only for sub-steps and requires that the final partial decomposition only uses edges from G. Assume that the tree-edge $\{a, y\}$ is replaced with a path P that exist in G using vertices introduced in higher bags.

The path P must contain a vertex b that is a neighbor of a in the graph G. The definition of a tree-decomposition describes that for every such neighbor b of a in G, there exist a tree-decomposition bag that contains both a and b. After the forget-bag of a, no single bag containing a can exist. Therefore there can be no such vertex b, contradiction. □

Corollary 9.5. *The neighbors of a forgotten vertex cannot be changed.*

Proof. Lemma 9.4 proofs that the vertex from an introduce bag cannot be placed as a neighbor of a forgotten vertex. As a consequence the endpoints of forgotten vertices are fixed, and it is impossible to insert a vertex between two vertices when there is a tree-edge with non-zero length between them. □

Lemma 9.6. *The spanheight of a spanning back-edge is unaffected by the forget operation on a tree-decomposition.*

Proof. Let (F, B, l, s, Z) be a restricted partial decomposition of the graph G. Forget the vertex $a \in V(F)$ from this restricted partial decomposition and get the new restricted partial decomposition (F', B', l', s', Z'). Assume that there exist an back-edge $\{x, y\}$ spanning the vertex a for which the spanheight has changed as result of the forget operation. We compute the spanheight in both cases and verify the assumption.

For case 1, let P be the path below the back-edge $\{x, y\}$ in F. The spanheight is equal to the length of this path $\sum_{e \in E(P)} l(e) + |V(P)| - 2 + s(\{x, y\})$. For case 2 two we subtract 1 vertex from the path P, remove its edge e_l to the left and e_r to the right on path P and replace it with and edge e_n with length function $l(e_n) = l(e_l) + l(e_r) + 1$. We conclude that the first term $\sum_{e \in E(P)} l(e)$ increases by 1 and the second term $|V(P)|$ decreases by 1, and the spanheight of the second case is equal to the first case. This contradicts the assumption that the spanheight has changed as result of the forget operation. □

Lemma 9.7. *The spanheight of a forgotten back-edge is enforced after the forget operation on a tree-decomposition.*

Proof. Let (F, B, l, s, Z) be a restricted partial decomposition of the graph G. Forget the vertex $a \in V(F)$ from this restricted partial decomposition and get the new restricted partial decomposition (F', B', l', s', Z'). After the back-edge $\{a, b\}$ to some $b \in V(F)$ has been forgotten and represented by the back-edge $\{x, b\}$ where x is the first vertex on the path P from a to b. Let $P' = P \setminus \{a\}$ be the path from x to b. Clearly if we are allowed to insert at most q vertices on the path P then we can also insert at most q vertices on the path P' since it is impossible to place a vertex between a and x after a is forgotten. And as a consequence the spanheight of an edge spanning P must be equal to an edge spanning P'. Assume that this is not the case and the spanheight of the back-edge $\{a, b\}$ equals q and the spanheight of the back-edge $\{x, b\}$ equals q' such that $q' \neq q$.

The algorithm defines the spanheight as the length of the path below the back-edge plus the forgotten spanheight $\sum_{e \in E(P)} l(e) + |V(P)| - 2 + s(\{a, b\})$. The representative back-edge has 1 less vertex $|V(P')| = |V(P)| - 1$, and 1 less edge $E(P') = E(P) \setminus \{a, x\}$. Therefore, the length of the path below it is decreased by $1 + l(\{a, x\})$, which is added as forgotten spanheight $s(\{x, b\}) = s(\{a, b\}) + 1 + l(\{a, x\})$ as described in the forget operation. We conclude that the spanheight of the representative back-edge equals the spanheight of the original, contradiction.

$$\sum_{e \in E(P)} l(e) + |V(P)| - 2 + s(\{a, b\}) = \sum_{e \in E(P) \setminus \{a, x\}} l(e) + |V(P)| - 3 + s(\{a, b\}) + 1 + l(\{a, x\})$$

□

A pair of vertices can have multiple representative back-edges assigned to it. The algorithm deals with these duplicates by keeping only 1 representative instance of each back-edge and deleting all others. The following lemma proves that this is does not interfere with the integrity of the algorithm.

Lemma 9.8. *Let (F, B, l, s, Z) be a restricted partial decomposition of the graph G. Let $e \in B$ be a back-edge in F with forgotten spanheight q. A new duplicate of the back-edge e with forgotten spanheight q' is created as consequence of a forget operation. The back-edge e with forgotten spanheight* $\max(q, q')$ *is representative for both instances.*

Proof. Let P be the path below e in F, and k the an integer upper bound on the restricted-spanheight of G. The instance 1 allows at most $k - (|P| - 2) - q$ vertices to be inserted on the path P, the instance 2 allows at most $k - (|P| - 2) - q'$ vertices to be inserted. Satisfying both instances we can insert at most $k - (|P| - 2) - \max(q, q')$ vertices on P. □

Corollary 9.9. *The restricted partial decomposition (F, B, l, s, Z) is equivalent to a partial decomposition F on a tree-decomposition $\mathcal{T}$ of the graph G.*

Proof. A vertex has 5 properties in a spanheight-decomposition.

1. Requires its neighbors from G to share an ancestor-descendant relationship in F in order to prevent cross-edges.
2. Requires its neighbors in the final partial decomposition F to be neighbors in G.
3. Effects the spanheight of spanning back-edges.
4. Effects the placement of new vertices below its back-edges.
5. Has at most one parent.

The equivalence follows from the fact that all the effects of these properties are guaranteed to persist without side-effects in the restricted partial decomposition. First in Lemma 9.3 proves that it is impossible to create cross-edges to forgotten vertices. Then Lemma 9.4 proves for property 2 that neighbors need to be fixed before the forget operation. Property 3 is guaranteed by the length function l as proven in Lemma 9.6. Property 4 is guaranteed by the representative back-edges with the forgotten spanheight function s as proven in Lemma 9.7 and 9.8. Property 5 is explicitly represented by the set Z containing vertices that have a forgotten parent and therefore the algorithm will not create new parents for these vertices. □

9.3.3 Properties of bag specific algorithms

The algorithm defines a separate function for each bag-type of a nice tree-decomposition. Each of these functions is designed to enumerate all restricted partial decompositions of the current bag, based on the restricted partial decompositions of the child bag(s). This section shows that the enumeration for each bag-type is correct.

Lemma 9.10. *Let $(F', B', l', s', Z') \in R'$ be a restricted partial decomposition of spanheight at most k of the child bag, then the introduce operation will enumerate every way of introducing the vertex x to (F', B', l', s', Z').*

Proof. We proof this by showing that the two cases described in the definition cover the complete search space. Either the introduce x has no parent in F or it has any vertex $p \in V(F')$ as parent, and it inherits a subset C of p's children. Clearly x cannot have other descendants or ancestors of p as children because this would create a cycle containing at least one vertex that has multiple parents. Or the cycle is broken and it creates a cross-edge. Additionally, x can have the root of any tree as child, except the root of the tree containing x. Trivially x cannot have internal and leaf nodes of other trees as children or those children would get multiple parents or cross-edges. This concludes every way of assigning a parent and children to the introduced vertex is enumerated, proving the lemma. □

Lemma 9.11. *Let $(F', B', l', s', Z') \in R'$ be a restricted partial decomposition of spanheight at most k of the child bag, then the forget operation outputs at most one restricted partial decomposition (F, B, l, s, Z) of spanheight at most k.*

Proof. For each restricted partial decomposition (F', B', l', s', Z'), the computed restriction (F, B, l, s, Z) is equivalent to it on a tree-decomposition. This equivalence follows from the Corollary 9.9. Clearly there is no reason to compute a second restriction, because it would be required to be equivalent to the first one, since $(a = b \wedge a = c) \rightarrow b = c$. $\square$

Lemma 9.12. *Let $(F_1, B_1, l_1, s_1, Z_1) \in R_1$, $(F_2, B_2, l_2, s_2, Z_2) \in R_2$ be restricted partial decompositions of spanheight at most k of the child bags, then the join operation outputs at most one restricted partial decomposition (F, B, l, s, Z) of spanheight at most k.*

Proof. First we proof the requirements for the merge to be fair:

- $F_1 = F_2$. The edge creation relaxation allows us to create every tree on both side of the join, we assume the sets R_1 and R_2 to contain a restriction with $F = F_1 = F_2$ if it is possible to create an spanheight decomposition equivalent to F on the sub-graph covered in the current state of the algorithm. It is redundant to mix features of F_1 and F_2 to create a forest F'. This follows from the fact that if F' can be achieved by mixing two trees then we must also be able to create F' on each side of the join, and therefore it is already represented in both R_1 and R_2.
- $\forall e \in E(F)(l_1(e) = 0 \vee l_2(e) = 0)$, any tree-edge in the new forest can only represent one forgotten path. If both F_1 and F_2 have a forgotten path on any tree-edge $e \in E(F)$ than they can never merge. This follows from Corollary 9.5 showing that the endpoints of the forgotten path are not allowed to change. If the merges keeps both forgotten paths, then this creates a cycle.
- $Z_1 \cap Z_2 = \emptyset$, only one side of the join may have a forgotten parent above a tree-root, otherwise this tree-root would have two parents.

Clearly the forest does not change in the merge. The forgotten paths are merged correctly since only 1 forgotten path between any pair of adjacent tree-vertices is allowed, and if this is not the case, then the merge is stopped. Merging back-edges can be done by copying all back-edges from both sides of the join. For the duplicate back-edges the Lemma 9.8 proofs that it is sufficient to store 1 representative with the maximum forgotten spanheight. The vertices with forgotten parents are also copied from both sides. At the end of the join operation the spanheight of the new restriction is computed, if the spanheight has increased above k then the two restrictions cannot be merged. We conclude that every step of the join operation is computed correctly. And since a merge can only be done in one specific way, there is only one merged restriction for any pair from R_1 and R_2. $\square$

9.3.4 Correctness main algorithm

Lemma 9.13. *Let algorithm 2 be called on $(G, \mathcal{T}, X_r, k)$, where $G = (V, E)$ is an connected undirected graph, $\mathcal{T} = (X, F)$ is a nice tree-decomposition of G, X_r is the root bag of the tree-decomposition $\mathcal{T}$, and k is an integer upper-bound on the restricted-spanheight of G. Then the algorithm the computes each bag X_i, the the set R_i that contains all restricted partial decompositions on the graph $G[V(\mathcal{T}_{X_i})]$ with spanheight at most k restricted to the vertices from the bag X_i, in which the vertices from only the current bag X_i are still under the edge creation relaxation.*

Proof. We proof the lemma using structural induction. The invariant is that for every restricted-spanheight decomposition of the sub-graph $G[V(\mathcal{T}_{X_i})]$ where the vertices in X_i are under the edge creation relaxation, the set R_i computed for every bag $i \in I$ contains a equivalent restricted partial decomposition. Furthermore, R_i contains no restricted partial decomposition for which no such equivalent restricted-spanheight decomposition exists. The set R_i is computed with one of the four algorithms associated with the four bag types. We proof for each of these algorithms that the invariant holds for the set R_i they compute. The algorithm $Leaf(X_i)$ computes the set R containing the unique restricted partial decomposition (F, B, l, s, Z) using one vertex. Trivially this set is computed correctly and there cannot exist more than one such restricted partial decomposition for the leaf bag.

Introduce bag Let X_i be the current bag of the tree-decomposition in the algorithm. Let X' be the child bag, and let v be the introduce vertex. By the induction hypothesis the set R' computed for the child bag X' contains all restricted partial decompositions on the graph $G[V(\mathcal{T}_{X'})]$ of spanheight at most k restricted to the vertices from the bag X', in which the vertices from only the current bag X' are still under the edge creation relaxation. We show that the invariant holds for the set R_i computed for the current bag.

For each restricted partial decomposition $(F', B', l', s', Z') \in R'$ of the child bag, the algorithm will enumerate every way of adding the introduce vertex v, as proven in Lemma 9.10. It follows that every possible restricted partial decomposition is enumerated and represented in R_i. Assume that this is not the case and there exist a restricted partial decomposition (F, B, l, s, Z) of spanheight at most k as a restriction of a partial decomposition F'', and that (F, B, l, s, Z) is not created by inserting the introduce vertex v into a restricted partial decomposition of a child bag. Then by the definition of edge relaxation we can delete the introduce vertex v from F'' to get the partial decomposition F'''. Then by the induction hypothesis the set R' for the childbag must contain a restriction of F''' of which we enumerate every possible way of inserting v and eventually get F'', contradiction.

We have left to show that there are no invalid restricted partial decompositions in R_i, this follows from the equivalence of the restricted partial decomposition to a partial decomposition. It allows the algorithm to check the spanheight and cross-edges of the computed restricted partial decomposition. The algorithm is also careful not to create edges to vertices of whom the edge creation relaxation is lifted. We conclude that invariant is preserved by the set R_i computed for an introduce bag X_i.

Forget bag Let X_i be the current bag of the tree-decomposition in the algorithm. Let X' be the child bag, and let v be the forget vertex. By the induction hypothesis the set R' computed for the child bag X' contains all restricted partial decompositions on the graph $G[V(\mathcal{T}_{X'})]$ of spanheight at most k restricted to the vertices from the bag X', in which the vertices from the bag X' are still under the edge creation relaxation. We show that the invariant holds for the set R_i computed for the current bag.

The forget operation takes all restrictions from the child bag and restricts them to the set $X_i = X' \setminus \{v\}$. As proven in 9.11 there is only one correct way to do this. Clearly all restrictions are computed in the set R_i. As part of the forget operation the edge creation relaxation is lifted for the forget vertex u, and the algorithm discards restrictions with erroneous guessed edges. We conclude that invariant is preserved by the set R_i computed for an forget bag X_i.

Join bag Let X_i be the current bag of the tree-decomposition in the algorithm. Let X_1, and X_2 be the child bags. By the induction hypothesis the set R_1, and R_2 computed for the child bags contain all restricted partial decompositions on the graphs $G[V(\mathcal{T}_{X_1})]$, and $G[V(\mathcal{T}_{X_2})]$ of spanheight at most k restricted to the vertices from the bag X_1, and X_2, in which the vertices from only the bags X_1, X_2 are still under the edge creation relaxation. We show that the invariant holds for the set R_i computed for the current bag. To join partial decompositions from both sides the algorithm considers every pair $c_1 \in R_1$, and $c_2 \in R_2$. From two such restrictions the algorithm computes at most one restricted partial decomposition of spanheight at most k. The steps taken by the algorithm are proved to be correct in the Lemma 9.12. We conclude that invariant is preserved by the set R_i computed for an join bag X_i.

The algorithm for each bag-type preserves the invariant, proving the lemma. □

Corollary 9.14. *Let algorithm 1 be called on $(G, \mathcal{T}, k)$, where $G = (V, E)$ is an connected undirected graph, $\mathcal{T} = (X, F)$ is a nice tree-decomposition of G, and k is an integer upper-bound on restricted-spanheight of G. Then the algorithm decides whether the graph G admits a restricted-spanheight decomposition of spanheight at most k.*

Proof. The set R computed by calling Algorithm 2 contains every restriction of any restricted-spanheight decomposition of G in which the vertices from the root bag X_r are still under the edge creation relaxation by Lemma 9.13. Normally the relaxation is lifted at the forget operation, only for the last bag we manually lift the relaxation by deleting all restrictions from R that contain tree-edges of length zero that are not contained in G. Any restricted-spanheight decomposition of spanheight at most k is a positive answer for the decision problem, and by Corollarry 9.9 and Lemma 9.1 the same holds for the existence of its restriction in R. $\square$

9.4 Running time

Let tw denote the treewidth upper bound, and k the spanheight upper bound of a graph. In order to simplify our running times, we will makes use of the fact that in an optimal scenario $tw \leq k+1$. In practice, an approximation of a tree-decomposition may be used such that $tw > k+1$.

Lemma 9.15. *For a tree-decomposition of treewidth tw, the number of unique restricted partial decompositions with spanheight at most k, for any bag X_i, is bounded by $O(2^{(3tw+2tw^2)\log(k+2)})$.*

Proof. A restricted partial decomposition is represented by the penta (F, B, l, s, Z). The number of rooted labeled forests on tw vertices is bounded by $O((tw+1)^{tw-1}))$. This is a famous result by Cayley [11]. For each forest F the set of (representative) back-edges is a subset of all possible edges on the graph 2^{tw^2}. For each of the at most $O(tw-1)$ tree-edges we assign a number between 0 and $k+1$ as the length of the edge, giving $O((k+2)^{tw-1})$ different values for the length function. The forgotten spanheight function s has at most $k+1$ values for each of the $O(tw^2)$ back-edges giving $O((k+1)^{tw^2})$ different definitions for the function s. The set Z is a subset of the vertices from the current bag and the different number of values for Z is bounded by $O(2^{tw})$. Taking all combinations gives $O((tw+1)^{tw-1} \cdot 2^{tw^2} \cdot (k+2)^{tw-1} \cdot (k+1)^{tw^2} \cdot 2^{tw}) \leq O(2^{(3tw+2tw^2)\log(k+2)})$. $\square$

Lemma 9.16. *Let algorithm 2 be called on $(G, \mathcal{T}, X_r, k)$, where $G = (V, E)$ is an connected undirected graph of size n, $\mathcal{T} = (X, F)$ is a nice tree-decomposition of G with treewidth $\boldsymbol{tw}(G)$, X_r is the root bag of the tree-decomposition $\mathcal{T}$, and k is an integer upper-bound on the restricted-spanheight of G. The running time complexity of the algorithm is $O(n \cdot 2^{(4tw^2+5tw)\cdot\log(k+2)} \cdot tw^2 \cdot k)$.*

Proof. For simplicity, let $O(\sigma)$ be the bound on the number of unique restricted partial decompositions. For each bag type the complexity is different, the leaf bag requires $O(1)$ time and the forget bag requires $O(1)$ time for all $O(tw)$ back-edges of the forgotten vertex for all $O(\sigma)$ restrictions of the child bag.

During the introduce operation for each of the $O(\sigma)$ restrictions of the child bags the introduced vertex has at most $O(tw \cdot 2^{tw})$ placements, and for every placement the spanheight for all new and effected edges $O(2 \cdot tw)$ has to be recomputed in $O(k)$ time. The total complexity of the operation is $O(\sigma \cdot 2^{tw} \cdot tw^2 \cdot k)$.

The join operation selects pairs of restrictions of the child bags. Let $(F_1, B_1, l_1, s_1, Z_1)$ and $(F_2, B_2, l_2, s_2, Z_2)$ be two restrictions of the child bags. F_1 is required to be the same as F_2, reducing the number of potential pairs by $O((tw+1)^{tw-1})$ combinations. The number of pairs is therefore bounded by $O((tw+1)^{tw-1} \cdot 2^{2\cdot tw^2} \cdot (k+2)^{2\cdot tw-2} \cdot (k+1)^{2\cdot tw^2} \cdot 2^{2\cdot tw}) \leq O(2^{(4tw^2+5tw)\cdot\log(k+2)})$. For each pair, merging the variables and computing the new restriction costs $O(tw^2 \cdot k)$ time. Clearly the join operation has the highest complexity of all bag types. The number of bags in a nice-tree decompositions is at most $O(5 \cdot n)$ and therefore the total amount of work for Algorithm 2 is bounded by $O(n \cdot 2^{(4tw^2+5tw)\cdot\log(k+2)} \cdot tw^2 \cdot k)$. $\square$

This allows us to state the main result of this chapter.

Theorem 9.17. *Algorithm 1 decides on the restricted-spanheight of an connected undirected graph G in $O(n \cdot 2^{(4tw^2+5tw)\cdot\log(k+2)} \cdot tw^2 \cdot k)$ time and $O(n2^{(3tw+2tw^2)\log(k+2)})$ space. And as a consequence, restricted-spanheight is contained in the complexity class FPT.*

Chapter 10

Spanheight FPT algorithm

The dynamic programming algorithm takes the quad $(G, \mathcal{T}, k, t)$ as input, where G is an undirected connected graph $G = (V, E)$, $\mathcal{T}$ a nice tree-decomposition of G with treewidth tw, and let k be an integer upper-bound on the spanheight of G, and let t be an integer upper-bound on the treedepth of G. The algorithm decides whether **spanheight**$(G) \leq k$ in $O(nt^4 \cdot 2^{7t^3 \cdot 2^t \cdot \log t})$ time and $O(n \cdot 2^{3t^3 \cdot 2^t \cdot \log t})$ space. For a yes-instance the algorithm can be modified to output a witnessing spanheight-decomposition using back-tracking in the dynamic programming tables.

10.1 Main algorithm

Our approach is exactly the same as for the algorithm in the previous chapter. The differences are mainly the state and the operations for each bag type. However, the state still uses definitions from the previous chapter. The correctness proofs will generally be the same, and we will often reference to proofs from the previous chapter. In order to avoid redefining definitions and proofs, in this chapter we assume the reader is familiar the previous chapter.

Definition (Partial decomposition). A partial decomposition is a rooted forest F with $V(F) \subseteq V(G)$ and $E(F) \not\subseteq E(G)$, where every tree $T = (V', E', r) \in F$ is a depth first search spanning tree. Arbitrary edge creation is allowed because spanheight considers super-graphs of G.

Definition (Spanheight of a Partial decomposition). The spanheight of a partial decomposition F is the maximum length of any path P between any two vertices with a back-edge to each other. The distance between two vertices is the length of P excluding the endpoints, which can be expressed as $|P| - 2$.

The key concept of the algorithm is that any tree T maintained in the state of the algorithm has height bounded by $O(2^t \cdot k)$, and that we need to maintain at most $O(tw)$ leaf nodes on a tree-decomposition.

We define three types of vertices, for which we store different information in the restricted partial decomposition. **Bag vertices** are vertices associated with the current bag of the tree-decomposition. **Internal vertices** are vertices that are forgotten in the tree-decomposition and an ancestor of at least one bag vertex. **Dangling vertices** are forgotten vertices that are not an ancestor of a bag vertex. On a tree-decomposition, we will delete all dangling vertices as illustrated in Figure 10.1. We call this a restriction of the partial decomposition to the bag-vertices X_i from the a tree-decomposition bag $i \in I$. A restriction may remember information about deleted vertices indirectly, in the form a representative back-edges. A restriction will also not remember the vertex labels of internal vertices. We will later proof that a restriction is equivalent to the original partial decomposition.

Definition (Restriction of a partial decomposition). A restriction of a partial decomposition is a quad (F, X, B, s) where,

- F is a forest of rooted trees $T \in F$, containing internal and bag vertices, $V(F) \subseteq V(G)$, and $E(F) \subseteq V(F) \times V(F)$, and $E(F) \not\subseteq E(G)$.
- X is the set of vertices that are not forgotten. On a tree-decomposition it is set of vertices from the current bag. We add X to the state to make our state more complete and explicit, it not not required as there is only one fixed set X for each bag of the tree-decomposition.

- $B \subseteq V(F) \times V(F)$ is the set of edges between vertices in F that are either back-edges or can become back-edges in a future state. Therefore, B contains all tree-edges and back-edges from G with both endpoints in F. The set B also contains representative back-edges in F that do not exist in G.
- $s : B \to \mathbb{N}^+$ is a function denoting the forgotten spanheight of the back-edges in B. The forgotten spanheight is clamped to the range $0 \leq s(e) \leq k$ because any higher would violate the spanheight k. The value zero denotes that an edge is not a representative back-edge.

We will abbreviate a restricted partial decomposition with "restriction" in cases where this cannot lead to confusion. It is not trivial why we can delete dangling vertices and replace them with the representative back-edges stored in the structures B and s. We will now give some intuition why we only care about the back-edges of the dangling vertices.

First we observe that for each restriction, the label of vertices are important because we are not allowed to create cross-edges. In Lemma 9.3 it is shown that we only need to know the labels of the bag-vertices or this. Therefore, for the dangling and internal vertices the label can be ignored in further computation. The DFS tree structure created by the bag-, internal-, and dangling vertices is important when introducing and inserting new vertices in a restriction. Therefore we cannot simply delete these vertices. In Corollary 10.1 it is shown that we cannot insert new vertices between a dangling vertex and the closest bag-vertex. The only possible effect the dangling vertex has on the placement of newly introduced vertices is the spanheight of its back-edges. The notion of a representative back-edge with a forgotten spanheight has an equivalent impact on the placement of new vertices as shown in Corollary 10.1. Therefore, dangling vertices can be deleted after the representative back-edge is created.

Definition (Spanheight of a restriction). Let (F, X, B, s) be a restriction, the spanheight of this restriction is the maximum spanheight of any back-edge covered by the restriction. The spanheight of a (representative) back-edge $e \in B$ equals the number of vertices it spans in F plus the forgotten spanheight $s(e)$.

Definition (Restriction equivalency). Let (F, X, B, s) and (F', X', B', s') be restrictions. They are equivalent if $X = X'$ and if there exist a graph isomorphic mapping $f : V(F) \to V(F')$ that maps every vertex in $v \in V(F)$ to a vertex in $v \in V(F')$, such that the following holds:

- For bag-vertex the labels matter, and therefore we require that the mapping f is an identity mapping $f(v) = v$ for vertices $v \in X$.
- If $v \in V(F)$ is the parent of $w \in V(F)$ in F, then $f(v)$ is required to be the parent of $f(w)$ in F'.
- If $\{v, w\} \in B$, then it is required that $\{f(v), f(w)\} \in B'$.
- $(\forall \{v, w\} \in B)(s(\{v, w\}) = s'(\{f(v), f(w)\}))$.

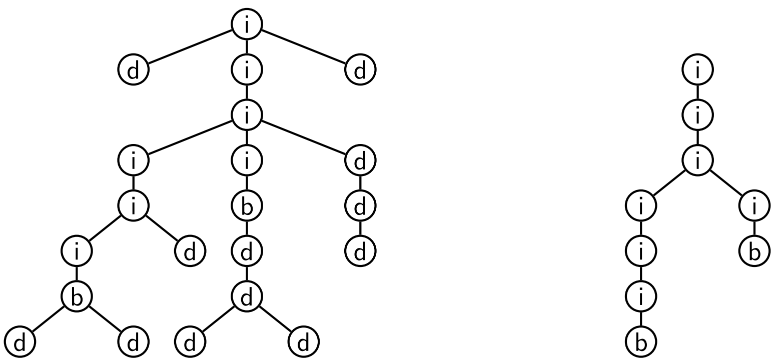


Figure 10.1: An example of a tree with bag-vertices b, internal vertices i, and dangling vertices d. The right tree is equivalent to the left one, created by removing all dangling vertices.

The last three rules test if the mapping is graph isomorphic. Note that that can only exist one mapping where f is an identity mapping for bag-vertices, and therefore the equivalency can be tested in polynomial time.

10.2 Algorithm operations

The Algorithm 3 uses the sub-algorithm 4 to compute the state for each bag, using a different function for each bag type of the nice tree-decomposition. First we define how we compute the spanheight of any restricted partial decomposition.

10.2.1 Computing the spanheight

Let (F, X, B, s) be a restricted partial decomposition, we compute the spanheight of each back-edge $\{v, w\} \in B$ and take the maximum $\max_{e \in B} as(e)$. Where we define the spanheight of an edge $as(e)$ to be the expression:

$$as(\{v, w\}) = |V(P)| - 2 + s(\{v, w\}) \tag{10.1}$$

Where:

- P is the tree-path from v to w in the forest F.
- $|V(P)| - 2$ be the number of vertices on the path P excluding the endpoints. These are the vertices in F spanned by the back-edge.

10.2.2 Leaf bag

Let $i \in I$ be a Leaf node of the tree-decomposition. The associated bag X_i contains only one vertex x. Create a class (F, X, B, s) with $F = \{(V = \{x\}, E = \emptyset, r = x)\}$, $X = \{x\}$, $B = \emptyset$, $(\forall e \in B)(s(e) = 0)$. Place it in the set of classes for the current bag $R_i = \{(F, X, B, s)\}$.

10.2.3 Introduce operation

Let $R \leftarrow \emptyset$ be the set of restricted partial decompositions of current bag, and let R' be set of the child bag. Let x be the introduce vertex. For each class $(F', X', B', s') \in R'$ we take the forest F', and generate new forests by inserting x in locations that do not create cross-edges or edges between the disjoint trees of the forest. Places we consider are:

- As child of any vertex $p \in V(F')$ and inheriting none or a subset of its children. Let C be the set of children of p in F'. The vertex x can take any subset of $C' \subseteq C$ as its children

Algorithm 3: Spanheight on graphs of bounded treedepth

```
input  : The triple $(G, \mathcal{T}, k, t)$, where $G = (V, E)$ is a connected undirected graph,
         $\mathcal{T} = (X, F)$ is a nice tree-decomposition of $G$, $k$ is an integer denoting an upper
         bound on the spanheight of $G$, and $t$ is an integer denoting an upper bound on
         the treedepth of $G$.
output : True if there exist a spanheight-decomposition of the graph $G$ with spanheight at
         most $k$, and False otherwise.
1 Let $X_r$ be the root bag of $\mathcal{T}$
2 Let $R$ be the set containing all restricted partial decompositions of the root bag computed
  with spanheight-recursive$(G, \mathcal{T}, X_r, k)$.
3 if $R \neq \emptyset$ then
4     return True
5 else
6     return False
```

Algorithm 4: Spanheight-recursive

```
input : The quad $(G, \mathcal{T}, X_i, k, t)$, where $G = (V, E)$ is a connected undirected graph,
        $\mathcal{T} = (X, F)$ is a nice tree-decomposition of $G$, $X_i$ is a bag of the
        tree-decomposition $\mathcal{T}$, $k$ is an integer denoting an upper-bound on the spanheight
        of $G$, and $t$ is an integer denoting an upper-bound on the treedepth of $G$.
output: A set $R$ containing all restricted partial decompositions on the sub-graph
        $G[V(\mathcal{T}_{X_i})]$.
1  $R := \emptyset$
2  if $X$ is a leaf then
3      $R = \text{Leaf}(X)$
4  else if $X$ is a forget bag then
5      Let $v$ be the forget vertex
6      Let $X'$ be the child bag of $X_i$.
7      $R' = \text{spanheight-recursive}(\mathcal{T}, G, X', k, t)$
8      $R = \text{Forget}(R', v)$
9  else if $X$ is an introduce bag then
10     Let $v$ be the introduce vertex
11     Let $X'$ be the child bag of $X_i$.
12     $R' = \text{spanheight-recursive}(\mathcal{T}, G, X', k, t)$
13     $R = \text{Introduce}(R', v, k)$
14 else if $X$ is a join bag then
15     Let $X_1$, and $X_2$ be the child bags of $X_i$.
16     $R_1 = \text{spanheight-recursive}(\mathcal{T}, G, X_1, k, t)$
17     $R_2 = \text{spanheight-recursive}(\mathcal{T}, G, X_2, k, t)$
18     $R = \text{Join}(R_1, R_2, k)$
19 return $R$
```

such that x lies on the path from p to $c \in C'$. The vertex x can also have the tree-roots of other trees as its children. Let Q be the set of tree-roots in F. The vertex x can take any subset $Q' \subseteq Q$ as children, including the empty subset.

- As the root of a tree, with possibly tree-roots of other trees as its children. Let Q be the set of tree-roots in F. The vertex x can take any subset $Q' \subseteq Q$ as children, including the empty subset.

The number of places we can insert the new vertex is exponential $O(|V(F)|2^{|X'|})$ in the number of vertices in X' because we can pick any subset of children for any parent. And every vertex has at most $|X'|$ children because all dangling vertices are removed. To finish the operation we add (back-)edges from and to the introduced vertex to B, and set $X = X' \cup \{x\}$.

For each placement j we compute the class (F_j, X, B, s') and put it in R if the spanheight of the computed class is lower than or equal to k. If R already contains a class equivalent to (F_j, X, B, s') then we don't add it.

10.2.4 Forget operation

Let $R \leftarrow \emptyset$ be the set of restricted partial decompositions of the current bag, and let R' be set of the child bag. Let x be the forget vertex. For each class $(F', X', B', s') \in R'$ we remove x from X' to get X. Then we delete dangling vertices from F' and create new representative back-edges to replace their effect.

If the forget vertex x has at least one child in F, then it must have at least one bag-vertex below it. (Or this child would be a dangling vertex and must have been removed in a previous step). In this case the operation for this individual restriction is finished, because no dangling vertices will be created as consequence of the forget operation for x.

In the case that x has no children, then all dangling vertices including x must be identified and removed. First mark all dangling vertices by visiting all vertices from F' in post-fix order by testing if they are not an ancestor of a bag-vertex. Then for all dangling vertices compute the respresentative back-edges.

Compute the representative back-edges As a consequence of deleting a dangling vertex v, its back-edges will also be deleted from B. For each vertex from the tree $w \in V(F')$ that has an back-edge to v create a new back-edge representing the old back-edge as shown in Figure 9.2:

- Let $u \in P \setminus \{v\}$ be the first non-dangling vertex on the tree-path P from v to w in F'. If there is no such vertex, or $u = w$, then stop since $P \setminus \{w\}$ will be deleted entirely, and the spanheight of $\{v, w\}$ cannot increase.
- Otherwise compute the the representative back-edge:
  - Let P' be the path from v to u.
  - Add the back-edge $\{u, w\}$ to B.
  - The forgotten spanheight of the new representing back-edge equals the forgotten spanheight of the original plus the reduced length of the path below it, this gives the expression $s(\{u, w\}) = s'(\{v, w\}) + |P'| - 2$.
  - The represented back-edge $\{v, w\}$ is computed even though this back-edge may already exist in B. If it exist then the new forgotten spanheight is the maximum of two $s(\{u, w\}) = \max(s(\{u, w\}), s(\{v, w\}) + |P'| - 2)$.

After all representative back-edges are computed, remove every back-edge e from B of which at least one endpoint is a dangling vertex.

To finish the operation remove all dangling vertices from F' to get F and Set X to $X' \setminus \{x\}$. The spanheight cannot increase in a forget operation, therefore add the new restriction (F, X, B, s) to R if it is not already represented by an equivalent restriction in R.

10.2.5 Join operation

Let $R \leftarrow \emptyset$ be the set of restricted partial decompositions of the current bag, and let R_1, R_2 be sets of the two child bags. For any pair $(F_1, X, B_1, s_1) \in R_1$, $(F_2, X, B_2, s_2) \in R_2$ we merge them if the following holds:

- For the bag vertices $v, w \in X$, v is an ancestor of w in F_1 if, and only if, v is an ancestor of w in F_2. This means that the structure imposed by the bag vertices is the same in both trees.

Merge the restrictions in every possible way to get a collection of new restrictions using the following operation:

- Observe that every leaf is a bag vertex, and there are $O(tw)$ leafs in the forests. Add a special root vertex r connected to every tree-root in F_1 and F_2 to simplify this definition. Let v be a bag vertex and w be either the closest ancestor bag-vertex or the special root vertex r if there is no such ancestor. Let P_1 and P_2 be the paths between v and w in the respective sides of the join. Let F be a merged forest containing all vertices from F_1 and F_2, rooted at the special root r. The path P from v to w in F contains all vertices between v and w from both sides of the join $P = P_1 \cup P_2$. The number of ways that P can be constructed is in the order of $\left(\!\!\binom{|P_1|}{|P_2|}\!\!\right) < 3^{|P|}$. The join operation enumerates all combinations of merging the path P above every $O(tw)$ bag-vertices. This creates at most $O(3^{|P| \cdot tw})$ different forests. For each case the special vertex r is removed, and is added to the set R of restriction if the spanheight is lower or equal than k.

10.3 Correctness of Algorithm

The correctness of the algorithm depends on the following properties:

- Partial decompositions can be used to solve the decision problem on a tree-decomposition.
- A restricted partial decomposition representation is equivalent to the original partial decomposition representation on a tree-decomposition.
- The defined operations for each bag-type of a nice tree-decomposition correctly compute all valid restricted partial decompositions.

As a consequence of these properties it follows using structural induction that for every bag $i \in I$, the state contains an restriction of every spanheight-decomposition of the sub-graph $G[V(\mathcal{T}_{X_i})]$. And that in the final table, every entry is a restriction of a spanheight-decomposition of the graph. In this section we will proof that the algorithm admits these properties and give the structural induction.

10.3.1 Properties of partial decompositions

Every spanheight-decomposition $T = (V, E, r)$ of a graph $G' = (V', E')$ has an equivalent partial decomposition F where T is the only tree in the forest F. It is trivial that any such partial decomposition T can be created by iteratively inserting vertices using any arbitrary ordering of the vertices.

10.3.2 Properties of restricted partial decompositions

The FPT algorithm for restricted-spanheight and spanheight use similar definitions of a restricted partial decomposition. A few properties of these restrictions are also shared, however their proofs require trivial differences. For example, the restricted-spanheight FPT algorithm has an additional length function $l(e)$. By trivially assuming that $l(e) = 0$, we can re-use proofs from the previous chapter.

Corollary 10.1. *The restricted partial decomposition (F, X, B, s) is equivalent to a partial decomposition F' on a tree-decomposition $\mathcal{T}$ of the graph G.*

Proof. A vertex has 5 properties in a spanheight-decomposition.

1. Requires adjacent vertices in the graph G to share an ancestor-descendant relationship in the forest F in order to prevent cross-edges.
2. Effects the spanheight of spanning back-edges.
3. Effects the placement of new vertices below its back-edges.
4. Each vertex can only have 1 parent.
5. Each vertex can have any neighbor.

The equivalence follows from the fact that all the effects of these properties are guaranteed to persist without side-effects in the restricted partial decomposition. First Lemma 9.3 proves that it is impossible to create cross-edges to forgotten vertices, and therefore we are allowed to forget the label of internal and dangling vertices. Property 2 persists on a restriction because the forgotten spanheight remembers the number of deleted vertices below every back-edge. Property 3 is guaranteed by the representative back-edges and the forgotten spanheight as shown in Lemma 9.7 and 9.8. Furthermore, created tree-edges that do not exist in G are never added to B, preventing the creation of unnecessary and erroneous back-edges. Property 4 is trivial because parents are only deleted when all children are deleted. Property 5 implies that we cannot delete vertices and replace them with a forgotten path, because new vertices can be placed on this forgotten path. Therefore, we do not delete internal vertices, and bag-vertices, Therefore, this property is preserved for those types. For dangling vertices it is redundant to create new neighbors, because any new neighbors can be added to new branch, independent of the dangling vertex. □

10.3.3 Properties of bag specific algorithms

The algorithm defines a separate function for each bag-type of a nice tree-decomposition. Each of these functions is designed to enumerate all restricted partial decompositions of the current bag, based on the restricted partial decompositions of the child bag(s). For the introduce and forget algorithms the Lemmas 9.10 and 9.11 trivially proof this property. The property trivially holds for the join algorithm, because it enumerates all possible ways to merge paths from each side the join.

10.3.4 Correctness main algorithm

Lemma 10.2. *Let algorithm 4 be called on* $(G, \mathcal{T}, X_r, k, t)$*, where* $G = (V, E)$ *is an connected undirected graph,* $\mathcal{T} = (X, F)$ *is a nice tree-decomposition of* G*,* X_r *is the root bag of the tree-decomposition* $\mathcal{T}$*,* k *is an integer denoting an upper bound on the spanheight of* G*, and* t *is an integer denoting an upper bound on the treedepth of* G*. Then the algorithm the computes each bag* X_i*, the set* R_i *that contains all restricted partial decompositions on the graph* $G[V(\mathcal{T}_{X_i})]$ *of spanheight at most* k *restricted to the vertices from the bag* X_i*.*

Proof. We proof the lemma using structural induction. The invariant is that for every spanheight decomposition of the sub-graph $G[V(\mathcal{T}_{X_i})]$, the set R_i computed for every bag $i \in I$ contains an equivalent restricted partial decomposition. Furthermore, R_i contains no restricted partial decomposition for which no equivalent spanheight decomposition of spanheight at most k exist. The set R_i is computed with one of the four algorithms associated with the four bag types. We proof for each of these algorithms that the invariant holds for the set R_i they compute. The algorithm $Leaf(X_i)$ computes the set R containing the unique restricted partial decomposition (F, X, B, s) using one vertex. Trivially this set is computed correctly and there cannot exist more than one such restricted partial decomposition for the leaf bag.

Introduce bag Let X_i be the current bag of the tree-decomposition in the algorithm. Let X' be the child bag, and let v be the introduce vertex. By the induction hypothesis the set R' computed for the child bag X' contains all restricted partial decompositions on the graph $G[V(\mathcal{T}_{X'})]$ of spanheight at most k restricted to the vertices from the bag X'. We show that the invariant holds for the set R_i computed for the current bag.

For each restricted partial decomposition $(F', X', B', s') \in R'$ of the child bag the algorithm will enumerate every way of adding the introduce vertex v. Lemma 9.10 with some trivial changes proofs this. It follows that every possible restricted partial decomposition is enumerated and represented in R_i. Assume that this is not the case and there exist a restricted partial decomposition (F, X, B, s) of spanheight at most k as a restriction of a partial decomposition F'', and that (F, X, B, s) is cannot be created by inserting the introduce vertex v into a restricted partial decomposition of a child bag. Then we can delete the introduce vertex v from F'' to get the partial decomposition F'''. Then by the induction hypothesis the set R' for the childbag must contain a restriction of F''', contradiction.

We have left to show that there are no bad restricted partial decompositions in R_i, this follows from the equivalence of the restricted partial decomposition to a partial decomposition. It allows the algorithm to check the spanheight and cross-edges of the computed restricted partial decomposition. We conclude that invariant is preserved by the set R_i computed for an introduce bag X_i.

Forget bag Let X_i be the current bag of the tree-decomposition in the algorithm. Let X' be the child bag, and let v be the forget vertex. By the induction hypothesis the set R' computed for the child bag X' contains all restricted partial decompositions on the graph $G[V(\mathcal{T}_{X'})]$ of spanheight at most k restricted to the vertices from the bag X'. We show that the invariant holds for the set R_i computed for the current bag.

The forget operation takes all restrictions from the child bag and restricts them to the set $X_i = X' \setminus \{v\}$. Lemma 9.11 proofs in combination with the equivalence relationship from Corollary 10.1 that exists only one correct way to do this. Clearly all restrictions are computed in the set R_i. We conclude that invariant is preserved by the set R_i computed for an forget bag X_i.

Join bag Let X_i be the current bag of the tree-decomposition in the algorithm. Let X_1, and X_2 be the child bags. By the induction hypothesis the set R_1, and R_2 computed for the child bags contain all restricted partial decompositions on the graphs $G[V(\mathcal{T}_{X_1})]$, and $G[V(\mathcal{T}_{X_2})]$ of spanheight at most k restricted to the vertices from the bag X_1, and X_2. We show that the invariant holds for the set R_i computed for the current bag.

To join partial decompositions from both sides the algorithm considers every pair $c_1 \in R_1$, and $c_2 \in R_2$. For each pair the algorithm computes all $O(3^{|P| \cdot tw})$ ways of merging them. It does require that the ancestor/descendant relationship between bag-vertices is the same in c_1 and c_2. If not, then there either exist some $c_3 \in R_2$ where this is the case, and c_3 can be merged with c_1 instead. Or this ancestor/descendant relationship from c_1 cannot be created at the other side of the join, and c_1 will never lead to a yes-instance. We conclude that invariant is preserved by the set R_i computed for an join bag X_i.

The algorithm for each bag-type preserves the invariant, proving the lemma. □

Lemma 10.3. *Let algorithm 3 be called on $(G, \mathcal{T}, k, t)$, where $G = (V, E)$ is an connected undirected graph, $\mathcal{T} = (X, F)$ is a nice tree-decomposition of G, k is an integer denoting an upper-bound on the spanheight of G, and t is an integer denoting an upper-bound on the treedepth of G. Then the algorithm decides if the graph G admits a spanheight decomposition of spanheight at most k.*

Proof. The set R computed by calling Algorithm 4 contains all and only all restrictions of all spanheight decompositions with spanheight at most k of the graph G as proven in Lemma 10.2. As a direct consequence, the existence of a restriction of the graph G proofs the existence of an spanheight decomposition. □

10.4 Running time

Let tw denote the treewidth upper bound, k the spanheight upper bound, and t the treedepth upper bound of a graph. In order to simplify our running times, we will makes use of the fact that in an optimal scenario $tw \leq k+2 \leq t$. In practice, an approximation of a tree-decomposition may be used such that $tw > t$. Note that $t \geq k+2$ iff $k > 0$, but since $k = 0$ is a trivial easy case, we are allowed to assume that $k > 0$.

Lemma 10.4. *For a graph G, with bounded tree-width tw, and bounded treedepth t, the number of possible restricted partial decompositions with spanheight at most k, for any bag X_i, is bounded by $O(2^{3t^3 \cdot 2^t \cdot \log t})$.*

Proof. A restricted partial decomposition is represented by the quad (F, X, B, s). First we provide an upper bound on the number of forests F that can exist. Recall that only the vertex labels of bag-vertices matter, and the internal vertices are graph isomorphic. Using only the $|X| \leq tw(G)$ bag-vertices we create at most $O((tw+1)^{tw-1}))$ different rooted forests (Cayley [11]). Because dangling vertices are removed, there are only bag and internal vertices in F. The internal vertices are between two bag-vertices or ancestors of bag vertices. We count the number of ways we can insert the internal vertices in the $O((tw+1)^{tw-1}))$ different rooted forest on bag-vertices. We can place at most $k \cdot 2^t$ internal vertices above each bag-vertex. This gives at most $(k \cdot 2^t)^{tw} \cdot (tw+1)^{tw-1}$ unique possibilities for F. Every forest F has at most $tw \cdot k \cdot 2^t$ vertices. To count back-edges only once, we count them only for the lower vertex. Each vertex in F can have at most (back-)edges to $k+1$ vertices above it, and the total number of back-edges is in the order of $O((k+1)k \cdot tw \cdot 2^t)$. The set B contains a subset of back-edges giving $O(2^{(k+1)k \cdot tw \cdot 2^t})$ different definitions of B. Then every single back-edge B can have $O(k+1)$ different values for its forgotten spanheight, giving $O((k+1)^{(k+1)k \cdot tw \cdot 2^t})$ unique definitions for the s function. Finally we combine the upper-bounds for each variable to get the upper bound on the number of unique restrictions: $O((k \cdot 2^t)^{tw} \cdot (tw+1)^{tw-1}) \cdot O(2^{(k+1)k \cdot tw \cdot 2^t}) \cdot O((k+1)^{(k+1)k \cdot tw \cdot 2^t})$. Combining all terms gives $O(2^{tw \log(k \cdot 2^t \cdot (tw+1)) + (k+1)k \cdot tw \cdot 2^t + (k+1)k \cdot tw \cdot 2^t \cdot \log(k+1)}) \leq O(2^{3t^3 \cdot 2^t \cdot \log t})$. □

Lemma 10.5. *Let algorithm 2 be called on $(G, \mathcal{T}, X_r, k)$, where $G = (V, E)$ is an connected undirected graph of size n, $\mathcal{T} = (X, F)$ is a nice tree-decomposition of G with treewidth* $\boldsymbol{tw}(G)$,

X_r is the root bag of the tree-decomposition $\mathcal{T}$, and k is an integer upper bound on the spanheight of G. The running time complexity of the algorithm is $O(nt^4 \cdot 2^{7t^3 \cdot 2^t \cdot \log t})$

Proof. For simplicity, let $O(\sigma)$ be the upper bound on the number of unique restricted partial decompositions. The running time complexity of the whole algorithm depends on algorithm for each bag type. The leaf bag requires $O(1)$ time to create a single restriction.

The forget operation iterates each of the $O(\sigma)$ restrictions of the child bag. It removes at most $k \cdot 2^t$ dangling vertices. Removing a single dangling vertex and computing representative back-edges can take at most $O(k^2)$ time. All dangling vertices are deleted in $O(k^3 \cdot 2^t)$ time. Finally every restriction needs to be stored in the set R for the current bag. Storing a restriction can be done by representing a restriction as a bit string that is unique up to equivalent restrictions. Meaning that two equivalent restrictions will have the same unique identifying bit string, and can therefore trivially be placed only once in R. The length of a bit string is $O(\log \sigma)$, and computing it requires time linear in the size of a restriction.

The introduce operation iterates each of the $O(\sigma)$ restrictions of the child bag. For each restriction it considers at most $O(tw \cdot 2^{tw})$ placements for the introduced vertex, and for every placement the spanheight for all new and effected edges $O(2 \cdot tw)$ has to be recomputed in $O(k)$ time. The total complexity of the operation is $O(\sigma(2^{tw} \cdot tw^2 \cdot k))$.

The join operation takes selects pairs of restrictions of the child bags. Let (F_1, X, B_1, s_1) and (F_2, X, B_2, s_2) be two restrictions of the child bags. There are at most $O(\sigma^2)$ ways to pick such pairs. For every pair, the paths above and between bag-vertices in F_1 and F_2 are merged in at most $O(3^{|P|})$ ways, giving at most $O(3^{|P| \cdot tw})$ different merged forests. Therefore the running time equals $O(\sigma^2 \cdot 3^{|P| \cdot tw})$ which is roughly $O(2^{7t^3 \cdot 2^t \cdot \log t}/2^t)$.

Clearly the join operation has the highest complexity of all bag types. Trivial operations like computing the variables, the spanheight, and the representative bit-string for each restriction take at most $O(tw^2 \cdot k^2 \cdot 2^t) \leq O(t^4 2^t)$ time. The number of bags in a nice-tree decompositions is at most $O(5 \cdot n)$, and therefore the total amount of work for Algorithm 4 is bounded by $O(nt^4 \cdot 2^{7t^3 \cdot 2^t \cdot \log t})$. □

Combining the correctness result from Lemma 10.3 and the complexity result from Lemma 10.5 we present the following theorem.

Theorem 10.6. *Let G be an undirected connected graph of size n, of bounded tree-depth t. Deciding whether G admits an spanheight decomposition of spanheight at most k and constructing such a decomposition takes at most $O(nt^4 \cdot 2^{7t^3 \cdot 2^t \cdot \log t})$ time and $O(n \cdot 2^{3t^3 \cdot 2^t \cdot \log t})$ space. As a consequence, spanheight is FPT when restricted to graphs of bounded treedepth.*

Chapter 11

Conclusion

We have studied the spanheight problem, which is an extension of bandwidth to depth first spanning trees. It can also be seen as an extension of treedepth to treewidth. Spanheight also closely resembles the treespan problem. Treedepth and treewidth have various practical applications. As far as we know, there is no application for spanheight. It would also be interesting to know if there exists a search game equivalent to it.

Open problem 1. *Are there practical applications for spanheight, and/or can it be defined as a search game?*

We also studied a restricted version of spanheight called restricted-spanheight. Comparing results between the two versions gives us insight into the complexity of spanheight on graphs versus supergraphs.

First we were able to proof NP-Completeness for spanheight on through a reduction to treedepth. The same reduction also works for restricted-spanheight to restricted-treedepth. As a side result we show the NP-Completeness of restricted-treedepth for cobipartite graphs by altering a known NP-Completeness proof for treedepth. This reduction to the balanced bipartite sub-graph problem is likely to work for the spanheight problems as well.

Open problem 2. *Are spanheight and restricted-spanheight NP-Hard on cobipartite graphs?*

We find a $O(n^2 9^n)$ time and $O(n^2 2^n)$ space algorithm for spanheight by adapting known algorithms for bandwidth. The algorithm forgets all structural information of the tree. For the restricted-spanheight problem we need to be guarantee that no new edges are created. This demanded that we extended the state, which leads to more states and state transitions. The algorithm for restricted-spanheight uses $O(n^2 21^n)$ time and $O(n^2 5^n)$ space. A single exponential time algorithm for treespan is still an open problem [16], we believe that the approach of our algorithms can be used to find one. In [12] measure and conquer is used to improve the bandwidth algorithm, which brings up the following question:

Open problem 3. *Can the exact algorithm for spanheight and restricted-spanheight from Theorem 6.5 and Theorem 6.7 be improved using measure and conquer?*

Finally we search for parameterized algorithms in order to make the problem tractable. We argue that spanheight cannot be FPT on graphs of bounded spanheight using tree-decompositions. To strengthen this argument we give a negative result for the graph minor theorem. Our argument leads us to believe that if we bounding the distance between any pair of vertices will lead to an FPT algorithm. Using Corollary 4.3 we bound the maximum distance between the root and any leaf when the input is restricted to graphs of bounded treedepth. Then we apply Courcelle's theorem to proof that this algorithm exists. For restricted-spanheight we show membership in FPT using Courcelle's theorem.

The FPT algorithm for spanheight is parameterized by treedepth. There may exist graphs that have a bounded spanheight indepedent of the graph size, but treedepth relative to the graph size. For these cases the existence of an FPT algorithm is an open problem.

Open problem 4. *Is spanheight contained in the complexity class fixed parameter tractable by parameterizing with spanheight instead of treedepth?*

We informally stated that spanheight is FPT on graph of bounded vertex cover number, based on its resemblance to treespan. However, formally it is an open problem.

Open problem 5. *Does there exist a FPT algorithm solving spanheight on graphs of bounded vertex cover number?*

In the last chapters we present actual FPT algorithms for both problems. The algorithm for restricted-spanheight uses $O(n \cdot 2^{(4tw^2+5tw)\cdot\log(k+2)} \cdot tw^2 \cdot k)$ time and $O(n2^{(3tw+2tw^2)\log(k+2)})$ space. And the algorithm for spanheight on graphs of bounded treedepth uses $O(nt^4 \cdot 2^{7t^3 \cdot 2^t \cdot \log t})$ time and $O(n \cdot 2^{3t^3 \cdot 2^t \cdot \log t})$ space. It is interesting to see that the difference in running time of these algorithms is so large. We observe that that dynamic programming techniques designed to forget information about the state work well for problems on supergraphs. In contrast, dynamic programming techniques like treewidth, that exploit the graph structure, perform better when the problem is restricted to the edges from the original graph. It would be interesting to see whether restricting other problems can be used to make them tractable. For example, does restricted-treedepth admit a faster algorithm than treedepth on tree-decompositions?

Bibliography

[1] Oswin Aichholzer and Klaus Reinhardt. “A quadratic distance bound on sliding between crossing-free spanning trees”. In: *Computational Geometry* 37.3 (2007), pp. 155–161.

[2] Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. “Complexity of finding embeddings in a k-tree”. In: *SIAM Journal on Algebraic Discrete Methods* 8.2 (1987), pp. 277–284.

[3] Hans L Bodlaender. “A linear time algorithm for finding tree-decompositions of small treewidth”. In: *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing.* ACM. 1993, pp. 226–234.

[4] Hans L Bodlaender. “A partial k-arboretum of graphs with bounded treewidth”. In: *Theoretical computer science* 209.1 (1998), pp. 1–45.

[5] Hans L. Bodlaender. “On linear time minor tests with depth-first search”. In: *Journal of Algorithms* 14.1 (1993), pp. 1–23.

[6] Hans L Bodlaender, Bart MP Jansen, and Stefan Kratsch. “Preprocessing for treewidth: A combinatorial analysis through kernelization”. In: *SIAM Journal on Discrete Mathematics* 27.4 (2013), pp. 2108–2142.

[7] Hans L Bodlaender et al. “Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth”. In: *Information and Computation* (2014).

[8] Hans L Bodlaender et al. “Rankings of graphs”. In: *SIAM Journal on Discrete Mathematics* 11.1 (1998), pp. 168–181.

[9] Hans L Bodlaender et al. “Solving weighted and counting variants of connectivity problems parameterized by treewidth deterministically in single exponential time”. In: *arXiv preprint arXiv:1211.1505* (2012).

[10] Richard B Borie, R Gary Parker, and Craig A Tovey. “Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families”. In: *Algorithmica* 7.1-6 (1992), pp. 555–581.

[11] Arthur Cayley. “A theorem on trees”. In: *Quart. J. Math* 23.376-378 (1889), p. 69.

[12] Marek Cygan and Marcin Pilipczuk. “Even faster exact bandwidth”. In: *ACM Transactions on Algorithms (TALG)* 8.1 (2012), p. 8.

[13] Marek Cygan and Marcin Pilipczuk. “Faster exact bandwidth”. In: *Graph-Theoretic Concepts in Computer Science.* Springer. 2008, pp. 101–109.

[14] Marek Cygan et al. “Solving connectivity problems parameterized by treewidth in single exponential time”. In: *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on.* IEEE. 2011, pp. 150–159.

[15] Rodney G Downey and Michael R Fellows. *Fundamentals of parameterized complexity.* Vol. 4. Springer, 2013.

[16] Markus Sortland Dregi. “Computation of Treespan. A Generalization of Bandwidth to Treelike Structures”. In: (2012).

[17] Uriel Feige. “Coping with the NP-hardness of the graph bandwidth problem”. In: *Algorithm Theory-SWAT 2000.* Springer, 2000, pp. 10–19.

[18] Michael R Fellows et al. “Graph layout problems parameterized by vertex cover”. In: *Algorithms and Computation.* Springer, 2008, pp. 294–305.

[19] Fedor V Fomin, Archontia C Giannopoulou, and Michał Pilipczuk. “Computing tree-depth faster than 2 n”. In: *Parameterized and Exact Computation.* Springer, 2013, pp. 137–149.

[20] Fedor V Fomin, Pinar Heggernes, and Jan Arne Telle. “Graph searching, elimination trees, and a generalization of bandwidth”. In: *Algorithmica* 41.2 (2005), pp. 73–87.

[21] Michael R Garey and David S Johnson. “Computers and intractability: a guide to the theory of NP-completeness. 1979”. In: *San Francisco, LA: Freeman* (1979).

[22] Wayne Goddard and Henda C Swart. “Distances between graphs under edge operations”. In: *Discrete Mathematics* 161.1 (1996), pp. 121–132.

[23] Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. “Known algorithms on graphs of bounded treewidth are probably optimal”. In: *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*. SIAM. 2011, pp. 777–789.

[24] Jaroslav Nešetřil and Patrice Ossona de Mendez. “Sparsity: Graphs, Structures, and Algorithms”. In: *Algorithms* (2012).

[25] Dieter Rautenbach. “Lower bounds on treespan”. In: *Information processing letters* 96.2 (2005), pp. 67–70.

[26] Felix Reidl et al. “A Faster Parameterized Algorithm for Treedepth”. In: *Automata, Languages, and Programming*. Springer, 2014, pp. 931–942.

[27] Ruben van der Zwaan. “Vertex Ranking with Capacity”. In: *SOFSEM 2010: Theory and Practice of Computer Science*. Springer, 2010, pp. 767–778.