

Treewidth Parameterized by Feedback Vertex Number

Hendrik Molter ✉ 

Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel

Meirav Zehavi ✉ 

Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel

Amit Zivan ✉

Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel

Abstract

We provide the first algorithm for computing an optimal tree decomposition for a given graph G that runs in single exponential time in the *feedback vertex number* of G , that is, in time $2^{\mathcal{O}(\text{fvn}(G))} \cdot n^{\mathcal{O}(1)}$, where $\text{fvn}(G)$ is the feedback vertex number of G and n is the number of vertices of G . On a classification level, this improves the previously known results by Chapelle et al. [Discrete Applied Mathematics '17] and Fomin et al. [Algorithmica '18], who independently showed that an optimal tree decomposition can be computed in single exponential time in the *vertex cover number* of G .

One of the biggest open problems in the area of parameterized complexity is whether we can compute an optimal tree decomposition in single exponential time in the *treewidth* of the input graph. The currently best known algorithm by Korhonen and Lokshtanov [STOC '23] runs in $2^{\mathcal{O}(\text{tw}(G)^2)} \cdot n^4$ time, where $\text{tw}(G)$ is the treewidth of G . Our algorithm improves upon this result on graphs G where $\text{fvn}(G) \in o(\text{tw}(G)^2)$. On a different note, since $\text{fvn}(G)$ is an upper bound on $\text{tw}(G)$, our algorithm can also be seen either as an important step towards a positive resolution of the above-mentioned open problem, or, if its answer is negative, then a mark of the tractability border of single exponential time algorithms for the computation of treewidth.

2012 ACM Subject Classification Theory of computation → Parameterized complexity and exact algorithms

Keywords and phrases Treewidth, Tree Decomposition, Exact Algorithms, Single Exponential Time, Feedback Vertex Number, Dynamic Programming

Funding This work was supported by the Israel Science Foundation, grant nr. 1470/24, by the European Union's Horizon Europe research and innovation programme under grant agreement 949707, and by the European Research Council, grant nr. 101039913 (PARAPATH).

1 Introduction

Treewidth is (arguably) both the most important and the most well-studied structural parameter in algorithmic graph theory [15, 17, 19, 20, 21, 46, 57]. Widely regarded as one of the main success stories, most textbooks in parameterized complexity theory dedicate at least one whole chapter to this concept; see e.g. Niedermeier [68, Chapter 10], Flum and Grohe [45, Chapters 11 & 12], Downey and Fellows [43, Chapters 10–14], Cygan et al. [38, Chapter 7], and Fomin et al. [48, Chapter 14]. In particular, treewidth is a powerful tool that allows us to leverage the conspicuous observation that many NP-hard graph problems are polynomial-time solvable on trees. Here, many fundamental graph problems can be solved by simple greedy algorithms that operate from the leaves to the (arbitrarily chosen) root [35, 41, 49, 67]. Treewidth, intuitively speaking, measures how close a graph is to a tree. In slightly more formal terms, it quantifies the width of a so-called *tree decomposition*, which generalizes the property that trees can be broken up into several parts by removing

single (non-leaf) vertices.¹ Tree decompositions naturally provide a tree-like structure on which dynamic programming algorithms can operate in a straightforward bottom-up fashion, from the leaves to the root. The concept of treewidth has been proven to be extremely useful and has led to a plethora of algorithmic results, where NP-hard graph problems are shown to be efficiently solvable if we know a tree decomposition of small width for the input graph [4, 6, 7, 12, 14, 27]. Maybe the most impactful such result is Courcelle’s theorem [29, 36, 37], which states that all problems expressible in monadic second-order logic can be solved in linear time on graphs for which we know a tree decomposition of constant width.

In the early 1970s, Bertelè and Brioschi [13] first observed the above-described situation. They showed that a large class of graph problems can be efficiently solved by “non-serial” dynamic programming, as long as the input graph has a bounded *dimension*. Roughly 15 years later, this parameter was proven to be equivalent to treewidth [4, 18]. In the meantime, the concept of treewidth was rediscovered numerous times, for example by Halin [51] and, more prominently, by Robertson and Seymour [69, 70, 71] as an integral part of their graph minor theory, one of the most ground-breaking achievements in the field of discrete mathematics in recent history.

In order to perform dynamic programming on a certain structure, it is crucial that this structure is known to the algorithm. In the case of efficient algorithms for graphs with bounded treewidth, this means that the algorithm needs to have access to the tree decomposition. Since computing a tree decomposition of optimal width is NP-hard [5], many early works on such algorithms assumed that a tree decomposition (or an equivalent structure) is given as part of the input [4, 7, 12]. The first algorithm for computing a tree decomposition of width k for an n -vertex graph (or deciding that no such tree decomposition exists) had a running time in $\mathcal{O}(n^{k+2})$ [5]. Shortly afterwards, an algorithm with a running time in $f(k) \cdot n^2$ was given by Robertson and Seymour [71]. It consists of two steps. In a first step, a tree decomposition of width at most $4k + 3$ is computed in $\mathcal{O}(3^{2k} \cdot n^2)$ time (or concluded that the treewidth of the input graph is larger than k). In a second, non-constructive step (for which the function f in the running time is not specified), this tree decomposition is improved to one with width at most k . In 1991, Bodlaender and Kloks [25]², and Lagergren and Arnborg [62] independently discovered algorithms with a running time in $2^{\mathcal{O}(k^3)} \cdot n$ for the second step. Furthermore, Bodlaender [16] gave an improved algorithm for the first step which allowed to compute a tree decomposition of width k (or concluding that no such tree decomposition exists) in time $2^{\mathcal{O}(k^3)} \cdot n$. These results have been extremely influential and allowed (among other things) Courcelle’s theorem [29, 36, 37] to be applicable without the prerequisite that a tree decomposition for the input graph is known. Roughly 30 years later, in 2023, Korhonen and Lokshtanov [60] presented an algorithm with a running time in $2^{\mathcal{O}(k^2)} \cdot n^4$, which is considered a substantial break-through. Whether a tree decomposition of width k (if one exists) can be computed in *single exponential time* in k , that is, a running time in $2^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$, remains a major open question in parameterized complexity theory [60].³

For many other graph problems, however, single exponential time algorithms in the treewidth of the input graph are known [7, 14, 22, 27, 39, 43, 45, 68]. If we want to use those algorithms to solve a graph problem, computing a tree decomposition of sufficiently

¹ We give a formal definition of *treewidth* and *tree decomposition* in Section 3.

² An extended abstract of the paper by Bodlaender and Kloks [25] appeared in the proceedings of the 18th International Colloquium on Automata, Languages, and Programming (ICALP) in 1991.

³ We remark that under the exponential time hypothesis (ETH) [52, 53], the existence of algorithms with a running time in $2^{o(k)} \cdot n^{\mathcal{O}(1)}$ can be ruled out [28].

small width becomes the bottleneck (in terms of the running time). Since we do not know how to compute a tree decomposition of minimum width in single exponential time, we have to rely on near-optimal tree decompositions if we do not want to significantly increase the running time of the overall computation. Therefore, much effort has been put into finding single exponential time algorithms that compute a tree decomposition with approximately minimum width, more specifically, with a width that is at most a constant factor away from the optimum. The first such algorithm was given by Robertson and Seymour [71] in 1995, and since then many improvements both in terms of running time and approximation factor have been made [3, 10, 11, 23, 59]. The current best algorithm by Korhonen [59] has an approximation factor of two and runs in $2^{\mathcal{O}(k)} \cdot n$ time. For an overview on the specific improvements, we refer to the paper by Korhonen [59].

1.1 Our Contribution

In this work, we focus on computing optimal tree decomposition. We overcome the difficulties of obtaining a single exponential time algorithm by moving to a larger parameter. The arguably most natural parameter that measures “tree-likeness” and is larger than treewidth is the *feedback vertex number*. It is the smallest number of vertices that need to be removed from a graph to turn it into a forest. Bodlaender, Jansen, and Kratsch [24] studied feedback vertex number in the context of kernelization algorithms for treewidth computation. The main result of our work is the following.

► **Theorem 1.1.** *Given an n -vertex graph G and an integer k , we can compute a tree decomposition for G with width k , or decide that no such tree decomposition exists, in $2^{\mathcal{O}(\text{fvn}(G))} \cdot n^{\mathcal{O}(1)}$ time. Here, $\text{fvn}(G)$ is the feedback vertex number of graph G .*

Theorem 1.1 directly improves a previously known result, that an optimal tree decomposition for an n -vertex graph G can be computed in $2^{\mathcal{O}(\text{vcn}(G))} \cdot n^{\mathcal{O}(1)}$ time (if it exists) [33, 47], where $\text{vcn}(G)$ is the *vertex cover number* of G , which is the smallest number of vertices that need to be removed from a graph to remove all of its edges, and hence always at least as large as the feedback vertex number. To the best of our knowledge, the only other known result for treewidth computation that uses a different parameter to obtain single exponential running time is by Fomin et al. [47], who showed that an optimal tree decomposition can be computed in $2^{\mathcal{O}(\text{mw}(G))} \cdot n^{\mathcal{O}(1)}$, where $\text{mw}(G)$ is the *modular width* of G . We remark that the modular width can be upper-bounded by a (non-linear) function of the vertex cover number, but is incomparable to both the treewidth and the feedback vertex number of a graph. It follows that our result improves the best-known algorithms for computing optimal tree decompositions (in terms of the exponential part of the running time) for graphs G where

$$\text{fvn}(G) \in o(\max\{\text{tw}(G)^2, \text{vcn}(G), \text{mw}(G)\}),$$

where $\text{tw}(G)$ denotes the treewidth of G . On a different note, since $\text{fvn}(G)$ is an upper bound on $\text{tw}(G)$, our algorithm can also be seen either as an important step towards a positive resolution of the open problem of finding an $2^{\mathcal{O}(\text{tw}(G))} \cdot n^{\mathcal{O}(1)}$ time algorithm for computing an optimal tree decomposition, or, if its answer is negative, then a mark of the tractability border of single exponential time algorithms for the computation of treewidth.

Our algorithm constitutes a significant extension of a dynamic programming algorithm by Chapelle et al. [33] for computing optimal tree decompositions which runs in $2^{\mathcal{O}(\text{vcn}(G))} \cdot n^{\mathcal{O}(1)}$ time. The building blocks of the algorithm are fairly simple and easy to implement, however

the analysis is quite technical and involved. First of all, our algorithm needs to have access to a *minimum feedback vertex set*, that is, a set of vertices of minimum cardinality, such that the removal of those vertices turns the graph into a forest. It is well-known that such a set can be computed in $2.7^{\text{fvn}(G)} \cdot n^{\mathcal{O}(1)}$ time (randomized) [63] or in $3.6^{\text{fvn}(G)} \cdot n^{\mathcal{O}(1)}$ deterministic time [58]. As a second step, we apply the kernelization algorithm by Bodlaender, Jansen, and Kratsch [24] to reduce the number of vertices in the input graph to $\mathcal{O}(\text{fvn}(G)^4)$. We remark that this kernelization algorithm needs a minimum feedback vertex set as part of the input and ensures that this set remains a feedback vertex set for the reduced instance [24]. This second step is neither necessary for the correctness of our algorithm nor to achieve the claimed running time bound, but it ensures that the polynomial part of our running time is bounded by the maximum of the running time of the kernelization algorithm and the polynomial part of the running time needed to compute a minimum feedback vertex set. Neither Bodlaender, Jansen, and Kratsch [24], Li and Nederlof [63], nor Kociumaka and Pilipczuk [58] specify the polynomial part of the running time of their respective algorithms. However, an inspection of the analysis for the kernelization algorithm suggests that its running time is in $\mathcal{O}(n^5)$ and it is known that a minimum feedback vertex set can be computed in $2^{\mathcal{O}(\text{fvn}(G))} \cdot n^2$ time [30].

In the following, we give a brief overview of the main ingredients of our algorithm. The main purpose of this overview is to convey an intuition and an idea of how the algorithm works; it is not meant to be completely precise. The basic terminology from graph theory that we use here is formally introduced and defined in Section 3.

- Chapelle et al. [33] introduced notions and terminology to formalize how a (rooted) tree decomposition interacts with a minimum vertex cover. We adapt and extend those notions for minimum feedback vertex sets. Inspired by the well-known concept of *nice* tree decompositions [25], we define a class of rooted tree decompositions that interact with a given feedback vertex set in a “nice” way. We call those tree decompositions *S-nice*, where S denotes the feedback vertex set. In *S-nice* tree decompositions, we classify each node t by which vertices of the feedback vertex set are contained in the bag and which are only contained in bags of nodes in the subtree rooted at t . We show that nodes of the same class form (non-overlapping) paths in the tree decomposition.
- We give algorithms to compute candidates of bags for the top and the bottom node of a path that corresponds to a given node class. Here, informally speaking, we differentiate between the cases where nodes (and their neighbors) have “full” bags or not, where we consider a bag to be full if adding another vertex to it increases the width of the tree decomposition. We need two novel algorithmic approaches for the two cases.
- We give an algorithm to compute the maximum width of any bag “inside” a path that corresponds to a given node class. This algorithm needs as input the bag of the top node and the bottom node of the path, and the set of vertices that should be contained in some bags of the path.
- The above-described algorithm allows us to perform dynamic programming on the node classes. Informally, we show that node classes form a partially ordered set. This means that for a given node class, we can look up a partial tree decomposition where the root belongs to a preceding (according to the poset) node class in a dynamic programming table. Then we use the above-described algorithm to extend the looked-up tree decomposition to one where the root has the given node class.

We show that in the above-described way, we can find a tree decomposition of width k whenever one exists. To this end, we introduce (additional) novel classes of tree decompositions that have properties that we can exploit algorithmically. We introduce so-called *slim* tree decompositions, which, informally speaking, have a minimum number of full nodes and

some additional properties. Furthermore, we introduce *top-heavy* tree decompositions, where, informally speaking, vertices are pushed into bags that are as close to the root as possible. We show that there exists tree decompositions with minimum width that are S -nice, slim, and top-heavy. Lastly, we give a number of ways to modify tree decompositions. We will show that we can modify any optimal tree decomposition that is S -nice, slim, and top-heavy into one that our algorithm is able to find.

1.2 Further Related Work

We already gave an overview of the work related to computing tree decompositions. The feedback vertex number is a fundamental graph parameter that has found applications in many contexts. The problem of computing the feedback vertex number is included in Karp's original list of 21 NP-hard problems [56]. As already mentioned, it can be computed in single exponential time and there is a long lineage of improvements in the running time [9, 50, 58, 63]. For more specific information on the improvements in the running time, we refer to the paper by Li and Nederlof [63]. Furthermore, many variations of this problem have been studied [2, 32, 34, 40, 55, 64, 65, 66]. The feedback vertex number has also been considered as a parameter in various contexts. In particular, as mentioned before, Bodlaender, Jansen, and Kratsch [24] studied feedback vertex number in the context of kernelization algorithms for treewidth computation. It has also been used as a parameter for algorithmic and hardness results in various other contexts [1, 32, 44, 54, 61, 72].

2 Algorithm Overview

In this section, we take a closer look at the different building blocks of our algorithm before we explain each one in full detail. The goal of this section is to provide an intuition about how the algorithm works, that is fairly close to the technical details but not completely precise. We describe the main ideas of each part of the algorithm and how they interact with each other.

2.1 How Feedback Vertex Sets Interact With Tree Decompositions

Given a rooted tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of a graph G and a minimum feedback vertex set S for G , the feedback vertex set interacts with the bags of \mathcal{T} in a very specific way. Given a bag X_t of some node t , we use $X_t^S = X_t \cap S$ to denote the vertices from S that are inside the bag X_t . We use $L_t^S \subseteq S$ to denote the set of vertices from S that are *only* in bags $X_{t'}$ “below” X_t . More formally, bags $X_{t'}$ of nodes t' that are descendants of node t . We use $R_t^S = S \setminus (L_t^S \cup X_t^S)$ to denote the set of vertices in S that are *only* in bags of nodes that are outside the subtree of \mathcal{T} rooted at t . These three sets build a so-called S -trace, a concept introduced by Chapelle et al. [33]. One can show that the nodes in a tree decomposition that share the same S -trace partition the tree decomposition into path segments [33]. These path segments (or the corresponding S -traces) form a partially ordered set which, informally speaking, allows us to design a bottom-up dynamic programming algorithm that constructs a rooted tree decomposition for G from the leaves to the root.

A similar approach was used by Chapelle et al. [33] to obtain a dynamic programming algorithm that has single exponential running time in the vertex cover number of the input graph. They use a minimum vertex cover instead of a minimum feedback vertex set to define S -traces. We extend this idea to work for feedback vertex sets, which requires a significant extension of almost all parts of the algorithm.

The main idea of the overall algorithm is the following. We give a polynomial-time algorithm to compute a partial rooted tree decomposition containing nodes corresponding to a directed path of an S -trace and, informally speaking, some additional nodes that only cover vertices from the forest $G - S$. We combine this partial rooted tree decomposition with one for the preceding directed paths according to the above-mentioned partial order. In other words, intuitively, we give a subroutine to compute partial tree decompositions for directed paths corresponding to S -traces, and try to assemble them to a rooted tree decomposition for the whole graph. We start with the directed paths that do not have any predecessor paths and then try to extend them until we cover all vertices. We save the partial progress we made in a dynamic programming table. The overall algorithm is composed out of fairly simple subroutines, which makes the algorithm easy to implement. However, a very sophisticated analysis is necessary to prove its correctness and the desired running time bound.

In order to bound the number of dynamic programming table look-ups that we need to compute a new entry, we need to be able to assume that the rooted tree decomposition we are aiming to compute behaves nicely (in a very similar sense as in the well-known concept of *nice tree decompositions* [25]) with respect to the vertices in S . To this end, we introduce the concept of *S -nice tree decompositions* and we prove that every graph G admits an S -nice tree decomposition with minimum width for every choice of S .

2.2 Computing a “Local Tree Decomposition” for a Directed Path

The main technical difficulty is to compute the part of the rooted tree decomposition that contains the nodes corresponding to a directed path of an S -trace. As described before, these parts are the main building blocks with which we assemble our tree decomposition for the whole graph.

Our approach here is to compute candidates for the bags of the top node and the bottom node of the directed path corresponding to an S -trace. Intuitively, since the S -trace specifies where in the tree decomposition the other vertices from S are located and since we know that $G - S$ is a forest, we have some knowledge on how these bags need to look like. If the bag X we are computing is for a node that, when removed, splits the tree decomposition into few parts, then we can “guess” which vertices of S are located in which of the different parts in an optimal tree decomposition. To this end, we adapt the concept of “introduce nodes”, “forget nodes”, and “join nodes” for S -nice tree decompositions. A consequence is that if a node does not share its bag X with any neighboring node, then, when the node is removed, the tree decomposition splits into at most three relevant parts (that is, parts that contain bags with vertices from S that are not in X). In other word, the bag X acts as a separator for at most three parts of S . This allows us to compute a candidate for X in a greedy fashion, and argue that we can modify an optimal tree decomposition to agree with our choice for X . An important property that we need for this is, that neighboring bags are not “full”, that is, their size is not $\text{tw}(G) + 1$, which gives us flexibility to shuffle around vertices.

However, if the optimal tree decomposition contains a large subtree of nodes that all have the same bag and that are all full, then removing those nodes can split the tree decomposition into an unbounded number of parts that each can contain a different subset of S . Here, informally speaking, we cannot afford to “guess” anymore how the vertices of S are distributed among the parts and we cannot shuffle around vertices. Hence, we have to find a way to compute a candidate bag for this case where we do not have to change the tree decomposition to agree with our choice.

To resolve this issue, we need to refine the concept of S -nice tree decompositions. We

introduce the *slim S -nice tree decompositions* that, informally speaking, ensure subtrees of the tree decomposition where all nodes have the same full bag have certain desirable properties that allow us to compute candidate bags efficiently. We prove that there are slim S -nice tree decomposition of minimal width for all graphs and all choices of S . Furthermore, we prove (implicitly) that we can bound the size of the search-space for a potential bag by a function that is single exponential in the feedback vertex number. Therefore, we will be able to provide small “advice strings”, that lead our algorithm for this case to a candidate bag and we can prove that there is an advice string that will lead the algorithm to the correct bag.

2.3 Coordinating Between Different Directed Paths

Since we need to modify the optimal tree decomposition in order to agree with the bags that we compute, we have to make sure that later modifications do not invalidate earlier choices for bags. To ensure this, we introduce so-called *top-heavy* slim S -nice tree decompositions. Informally speaking, we try to push all vertices that are not in S into bags as close to the root as possible. Intuitively, this ensures that if we make modifications to those tree decompositions to make them agree with our bag choices, those modifications do not affect bags that are sufficiently far “below” the parts of the tree decomposition that we modify.

Now, having access to the bag of the top node and the bottom node of a directed path corresponding to some S -trace, we have to decide which other vertices (that are not in S) we wish to put into bags that are attached to that path. Here, we also have to make sure that our decisions are consistent. We will compute a three-partition of the complete vertex set: one part that represents the vertices that are contained in some bag of the directed path or a bag that is attached to it, one part of vertices that are only in bags “below” the directed path, and the last part with the remaining vertices, that is, the vertices that are only in bags outside of the subtree of the tree decomposition that is rooted at the top node of the directed path. The first described part is needed to compute the partial tree decomposition of the directed path. We provide a polynomial-time algorithm to do this. Furthermore, the three-partition will allow us to decide which choices for bags to the top and bottom nodes of preceding directed paths are compatible with each other.

These are all the ingredients we need to design our algorithm.

2.4 Organization of the Paper

We organize our paper as follows. In Section 3 we introduce all standard notation and terminology that we use. In Section 4 we present some concepts by Chapelle et al. [33] that we adopt for our results and we introduce novel concepts, such as S -nice tree decompositions, slim tree decompositions, and top-heavy tree decompositions. Furthermore, we describe our methods to modify optimal tree decompositions such that they agree with our bag choices. In Section 5 we present the main dynamic programming algorithm. Section 6 is dedicated to the subroutines we use to compute the top bag and the bottom bag of directed paths corresponding to S -traces. In Section 7 we give a polynomial-time algorithm to compute the part of the tree decomposition for an directed path, provided that we know the bag of the top node, the bag of the bottom node, and the set of vertices that we wish to be contained in some bag attached to the directed path. In Section 8 we describe how we decide whether our decisions for different directed paths are consistent. Finally, in Section 9 we prove that our algorithm is correct and we give a running time analysis. In Section 10, we conclude and give some future research directions.

3 Preliminaries

We use standard notation and terminology from graph theory [42]. Let $G = (V, E)$ with $E \subseteq \binom{V}{2}$ be a graph. We denote by $V(G) = V$ and $E(G) = E$ the sets of its vertices and edges, respectively. We use n to denote the number of vertices of G . We call two vertices $u, v \in V$ *adjacent* if $\{u, v\} \in E$. The *open neighborhood* of a vertex $v \in V$ is the set $N_G(v) = \{u \mid \{u, v\} \in E\}$. The *degree* of a vertex $v \in V$ is $\deg(v) = |N_G(v)|$. The *closed neighborhood* of a vertex $v \in V$ is the set $N_G[v] = N_G(v) \cup \{v\}$. If a vertex v has degree one, then we call v a *leaf*. For some vertex set $V' \subseteq V$, we define its open neighborhood as $N_G(V') = \bigcup_{v \in V'} N_G(v) \setminus V'$ and its closed neighborhood as $N_G[V'] = N_G(V') \cup V'$. If G is clear from the context, we drop the subscript G from N . For some vertex set $V' \subseteq V$, we denote by $G[V']$ the *induced* subgraph of G on the vertex set V' , that is, $G[V'] = (V', E')$ where $E' = \{\{v, w\} \mid \{v, w\} \in E \wedge v \in V' \wedge w \in V'\}$. We further denote $G - V' = G[V \setminus V']$. We say that a sequence $P = (\{v_{i-1}, v_i\})_{i=1}^\ell$ of edges in E forms a *path* in G from v_0 to v_ℓ if $v_i \neq v_j$ for all $0 \leq i < j \leq \ell$. We denote with $V(P) = \{v_0, v_1, \dots, v_\ell\}$ the vertices visited by path P . We say that vertices u and v are *connected* in G if there is a path from u to v in G . We say that graph G is connected if for all $u, v \in V$ we have that u and v are connected. We say that $G[V']$ for some $V' \subseteq V$ is a *connected component* of G if $G[V']$ is connected and for all $v \in V \setminus V'$ we have that $G[V' \cup \{v\}]$ is not connected. We say that a vertex set $V' \subseteq V$ *separates* a vertex set $V_1 \subseteq V \setminus V'$ and a vertex set $V_2 \subseteq V \setminus V'$ if $V_1 \cap V_2 = \emptyset$ and there is no $u \in V_1$ and $v \in V_2$ such that there is a path from u to v in $G - V'$. We call G a *tree* if for each pair of vertices $u, v \in V$ there is exactly one path from u to v in G . We call G a *forest* if for each pair of vertices $u, v \in V$ there is at most one path from u to v in G . We call G *trivial* if $|V| = 1$.

Let $e = \{u, v\} \in E$. *Contracting* edge e in G results in the graph obtained by deleting v from G and then adding all edges between u and $u' \in N_G(v)$. A graph H is called a *minor* of G if H can be obtained from G by deleting edges, vertices, and by contracting edges.

Next, we formally define the *treewidth* of a graph and related concepts such as *tree decompositions* and *nice tree decompositions*.

► **Definition 3.1** (Tree Decomposition and Treewidth). *A tree decomposition of a graph $G = (V, E)$ is a pair $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ consisting of a tree T and a family of bags $\{X_t\}_{t \in V(T)}$ with $X_t \subseteq V$, such that:*

1. $\bigcup_{t \in V(T)} X_t = V$.
2. For every $\{u, v\} \in E$, there is a node $t \in V(T)$ such that $\{u, v\} \subseteq X_t$.
3. For every $v \in V$, the set $X^{-1}(v) = \{t \in V(T) \mid v \in X_t\}$ induces a subtree of T .

The width of a tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ is $\max_{t \in V(T)} |X_t| - 1$. The treewidth $\text{tw}(G)$ of a graph G is the minimum width over all tree decomposition of G .

Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a tree decomposition of a graph $G = (V, E)$ with width k . To avoid confusion, we refer to the elements of $V(T)$ as *nodes* and call elements of V *vertices*. Let X_t be the bag of a node t in \mathcal{T} , if $|X_t| = k + 1$, then we say the bag X_t is *full*, that is, increasing the size of the bag would increase the width of the tree decomposition. The following lemmas are well-known [8, 26, 42] and give us useful properties of tree decompositions.

► **Lemma 3.2** ([8, 42]). *Let H be a minor of a graph G . Then $\text{tw}(H) \leq \text{tw}(G)$.*

► **Lemma 3.3** ([26, 42]). *Let \mathcal{T} be a tree decomposition of a graph $G = (V, E)$. Let $V' \subseteq V$ such that $G[V']$ is complete. Then, there exists a node t in \mathcal{T} such that $V' \subseteq X_t$.*

► **Observation 3.4** ([42]). Let \mathcal{T} be a tree decomposition of a graph $G = (V, E)$ with width at most k , and let t be a node in \mathcal{T} . Let $G' = (V, E \cup \binom{X_t}{2})$. Then \mathcal{T} is a tree decomposition of a graph G' with width at most k .

In many applications, it is very convenient to use rooted tree decompositions with a particular structure. A *rooted tree* $T = (V, E, r)$ is a triple where (V, E) is a tree and $r \in V$ is a designated root node. Let $v \in V$ and $u \in N(v)$. If the unique path from r to u is shorter than the unique path from r to v , then we say that u is the *parent* of v . Otherwise, we say that u is a *child* of v . Let $P = (\{v_{i-1}, v_i\})_{i=1}^\ell$ be a path in T from v_0 to v_ℓ . If v_{i-1} is the parent of v_i for all $1 \leq i \leq \ell$, then we say that P is a *directed path* from v_0 to v_ℓ . If for two vertices $u, v \in V$ there is a directed path from u to v , then we say that u is an *ancestor* of v and v is a *descendant* of u . A *rooted tree decomposition* is a tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ where T is a rooted tree with root node $r \in V(T)$.

► **Definition 3.5** (Nice Tree Decomposition). A *nice tree decomposition* is a rooted tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ that satisfies the following properties. The bag of X_r is empty, and the bag X_ℓ is empty for every leaf ℓ in T . Additionally, every internal node t of T is of one of the following types:

Introduce Node: t has exactly one child t' , such that $X_t = X_{t'} \cup \{v\}$ for some $v \notin X_{t'}$.

Forget Node: t has exactly one child t' , such that $X_t = X_{t'} \setminus \{v\}$ for some $v \in X_{t'}$.

Join Node: t has exactly two children t_1, t_2 , such that $X_t = X_{t_1} = X_{t_2}$.

It is well-known that every tree decomposition can be straightforwardly transformed into a nice tree decomposition with the same width.

► **Lemma 3.6** ([25]). If a graph G admits a tree decomposition \mathcal{T} with width k , then it also admits a nice tree decomposition \mathcal{T}' with width k .

For our algorithm, we will use the feedback vertex number as a parameter and our algorithm will need access to a feedback vertex set. It is formally defined as follows.

► **Definition 3.7** (Feedback Vertex Set and Feedback Vertex Number). A *feedback vertex set* of a graph $G = (V, E)$ is a set $C \subseteq V$ such that $G - C$ is a forest. The *feedback vertex number* $\text{fvn}(G)$ of a graph G is the cardinality of a minimum feedback vertex set.

It is well-known that a minimum feedback vertex set can be computed in $2^{\mathcal{O}(\text{fvn})} \cdot n^{\mathcal{O}(1)}$ time. To the best of our knowledge, the following are the currently best-known algorithms.

► **Theorem 3.8** ([58, 63]). A minimum feedback vertex set of a graph G can be computed in $2.7^{\text{fvn}(G)} \cdot n^{\mathcal{O}(1)}$ randomized time or in $3.6^{\text{fvn}(G)} \cdot n^{\mathcal{O}(1)}$ deterministic time.

4 Main Concepts for the Algorithm

One of the main ingredients to prove Theorem 1.1 is showing that there are tree decompositions (in particular, also minimum-width ones) that interact with a given vertex set in a very structured way. To this end, we present several concepts introduced by Chapelle et al. [33] that we also use for our algorithm in Section 4.1. In particular, we give the definitions of *traces*, *directed paths* thereof [33], and several additional concepts. Then, in Section 4.2, we introduce *S-nice* tree decompositions, which interact with a vertex set S in a “nice” way, but are more flexible with vertices not in S . In Section 4.3, we present several ways to modify tree decompositions. Those are an important tool to show that we can modify an optimal tree decomposition into one that our algorithm finds. Finally, we introduce slim and

top-heavy S -nice tree decompositions in Section 4.4. Those types of tree decompositions have crucial additional properties that we need to show that our algorithm is correct. We use the modifications from Section 4.3 to show that we can transform S -nice tree decompositions into ones of those types.

4.1 S -Traces, Directed Paths, S -Children, and S -Parents

Let t be a node of a rooted tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of a graph $G = (V, E)$. We denote with T_t the subtree of T rooted at node t . We use the following notation throughout the document:

- The set $V_t = \bigcup_{t' \in V(T_t)} X_{t'}$ is the union of all bags in the subtree T_t .
- The set $L_t = V_t \setminus X_t$ is the set of vertices “below” node t .
- The set $R_t = V \setminus V_t$ is the set of the “remaining” vertices, that is, the ones that are neither in the bag of node t nor below t .

Clearly, for every node t , the sets L_t, X_t, R_t are a partition of V .

► **Definition 4.1** ([33]) (S -Trace). *Let $G = (V, E)$ be a graph and let $S \subseteq V$. Let t be a node of a rooted tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G . The S -trace of t is the triple (L_t^S, X_t^S, R_t^S) such that $L_t^S = L_t \cap S$, $X_t^S = X_t \cap S$, and $R_t^S = R_t \cap S$. A triple (L^S, X^S, R^S) is an S -trace if it is the S -trace of some node in some tree decomposition of G .*

We remark that in the work of Chapelle et al. [33], the set S in Definition 4.1 is fixed to be a minimum vertex cover. In our work, we will (later) fix it to be a minimum feedback vertex set. It is easy to see that S -traces have the following useful properties.

► **Observation 4.2** ([33]). *Let $G = (V, E)$ be a graph and $S \subseteq V$.*

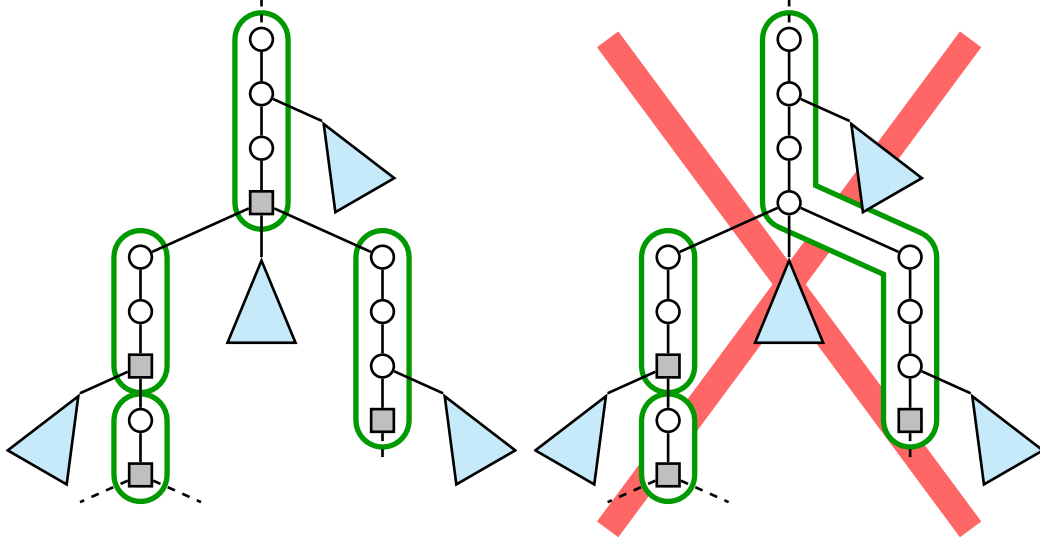
1. *Let t, t' be nodes of a rooted tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of a graph G . Let (L_t^S, X_t^S, R_t^S) and $(L_{t'}^S, X_{t'}^S, R_{t'}^S)$ be the S -traces of t and t' , respectively. If t is an ancestor of t' , then $L_{t'}^S \subseteq L_t^S$.*
2. *Let L^S, X^S, R^S be a partition of the set S . The triple (L^S, X^S, R^S) is an S -trace if and only if there is no path from a vertex in L^S to a vertex in R^S in $G[S] - X^S$.*

Chapelle et al. [33] show that the nodes of a rooted tree decomposition that have the same S -trace form a directed path. They prove this for S -traces where S is a minimum vertex cover and the rooted tree decomposition is nice, however inspecting their proof reveals that this property also holds for arbitrary sets S and arbitrary rooted tree decompositions. Formally, we have the following.

► **Lemma 4.3** ([33]). *Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a rooted tree decomposition of a graph $G = (V, E)$. Let $S \subseteq V$ and let (L^S, X^S, R^S) be an S -trace such that $L^S \neq \emptyset$. Moreover, let N be the set of nodes of \mathcal{T} whose S -trace is (L^S, X^S, R^S) . Then the subgraph of T induced by N is a directed path.*

Lemma 4.3 allows us to define so-called *directed paths of S -traces*, which will be central to our algorithm. Informally, given an S -trace our algorithm will compute candidates for the corresponding directed path, and then use a dynamic programming approach to connect the path to the rest of a partially computed rooted tree decomposition.

► **Definition 4.4** (Directed Path of an S -Trace, Top Node, and Bottom Node). *Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a rooted tree decomposition of a graph $G = (V, E)$, $S \subseteq V$ and (L^S, X^S, R^S) an S -trace such that $L^S \neq \emptyset$. We call the directed path induced by the set of nodes of \mathcal{T} whose S -trace is (L^S, X^S, R^S) the directed path of (L^S, X^S, R^S) in \mathcal{T} from node t_{\max} to*



(a) Illustration of an S -nice tree decomposition. (b) Observation 4.6 implies that this cannot happen.

■ **Figure 1** Illustrations for the concepts of S -traces (Definition 4.1) and their directed paths (Definition 4.4), S -bottom nodes (Definition 4.5), and S -nice tree decompositions (Definition 4.8). Subfigure 1a shows an illustration of a part of an S -nice tree decomposition. The directed paths of different S -traces are surrounded by green lines. The S -bottom nodes are depicted as gray squares. The blue triangles illustrate parts of the tree decompositions where all nodes have S -traces with $L^S = \emptyset$, and hence those nodes are not part of any directed paths of S -traces. Subfigure 1b shows a configuration of directed paths of different S -traces that we can rule out by Observation 4.6.

node t_{\min} . We call t_{\max} the top node of the directed path and we call t_{\min} the bottom node of the directed path.

We use the above concepts and some additional ones that we introduce below to define a specific type of rooted tree decomposition that interacts with the set S in a very structured way. The concepts are illustrated in Figure 1. First, we distinguish nodes that are at the end of a directed path of some S -trace.

► **Definition 4.5** (S -Bottom Node). Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a rooted tree decomposition of a graph $G = (V, E)$ and $S \subseteq V$. Let t be a node of \mathcal{T} . We say that t is an S -bottom node if there is an S -trace (L^S, X^S, R^S) such that $L^S \neq \emptyset$, and t is the bottom node t_{\min} of the directed path of (L^S, X^S, R^S) in \mathcal{T} .

Next, we observe that for every directed path of some S -trace, the parent of t_{\max} is also an S -bottom node. Note that this implies that a situation as depicted in Figure 1b cannot happen.

► **Observation 4.6.** Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a rooted tree decomposition of a graph $G = (V, E)$, $S \subseteq V$ and (L^S, X^S, R^S) an S -trace such that $L^S \neq \emptyset$. Let the directed path of (L^S, X^S, R^S) go from t_{\max} to node t_{\min} . We have that if t_{\max} has a parent, then the parent is an S -bottom node.

Proof. TOPROVE 0 ◀

We now define so-called S -parent and S -children. Informally, an S -child is a child of a node such that there is a vertex in S that is only contained in nodes in the subtree of the

tree decomposition rooted at the S -child. The S -parent of a node is defined analogously. Formally, we define them as follows.

- **Definition 4.7** (*S-Parent and S-Child*). Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a rooted tree decomposition of a graph $G = (V, E)$ and $S \subseteq V$. Let t and t' be two nodes of \mathcal{T} such that t is the parent of t' in T . Let (L_t^S, X_t^S, R_t^S) and $(L_{t'}^S, X_{t'}^S, R_{t'}^S)$ be the S -traces of t and t' , respectively.
- If $(R_t^S \cup X_t^S) \setminus (L_{t'}^S \cup X_{t'}^S) \neq \emptyset$, then we say t is an S -parent of t' .
 - If $(L_{t'}^S \cup X_{t'}^S) \setminus (R_t^S \cup X_t^S) \neq \emptyset$, then we say t' is an S -child of t .

4.2 S-Nice Tree Decompositions

We now give the definition of a so-called S -nice tree decomposition. Intuitively, this tree decomposition behaves nicely (in the sense of Definition 3.5) when interacting with vertices from the vertex set S , but is more flexible for the vertices in $V \setminus S$. Similarly to nice tree decomposition, S -nice tree decompositions distinguish three main types of S -bottom nodes, analogous to *introduce*, *forget*, and *join* nodes. We associate each S -bottom node with S -trace (L^S, X^S, R^S) with a so-called S -operation, which can be $\text{introduce}(v)$ for some $v \in X^S$, $\text{forget}(v)$ for some $v \in S \setminus X^S$, or $\text{join}(X^S, X_1^S, X_2^S, L_1^S, L_2^S)$ for some $X_1^S, X_2^S \subseteq X^S$ and $L_1^S, L_2^S \subseteq L^S$.

► **Definition 4.8** (*S-Nice Tree Decomposition and S-Operations*). A rooted tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of a graph $G = (V, E)$ is S -nice for $S \subseteq V$ if the following hold. Let t be a node. Then the following holds.

1. If t has a parent t' , then $X_t \subseteq X_{t'}$ or $X_t \supseteq X_{t'}$, and $-1 \leq |X_t| - |X_{t'}| \leq 1$.
If t is an S -bottom node in \mathcal{T} with S -trace (L_t^S, X_t^S, R_t^S) , then additionally one of the following holds.
2. Node t has exactly one S -child t_1 . We have that $X_t^S = X_{t_1}^S \cup \{v\}$ for some $v \in S \setminus X_{t_1}^S$. Then we say that t admits the S -operation $\text{introduce}(v)$.
3. Node t has exactly one S -child t_1 . We have that $X_t^S = X_{t_1}^S \setminus \{v\}$ for some $v \in S \cap X_{t_1}^S$. Then we say that t admits the S -operation $\text{forget}(v)$.
4. Node t has exactly two S -children t_1, t_2 . We have that $X_{t_1}^S \cup X_{t_2}^S \subseteq X_t^S$.
Let $(L_{t_1}^S, X_{t_1}^S, R_{t_1}^S)$ and $(L_{t_2}^S, X_{t_2}^S, R_{t_2}^S)$ be the two S -traces of t_1 and t_2 , respectively. We say that t admits the S -operation $\text{join}(X_t^S, X_{t_1}^S, X_{t_2}^S, L_{t_1}^S, L_{t_2}^S)$.

See Figure 1a for an illustration of an S -nice tree decomposition. We show that if there is a tree decomposition for a graph $G = (V, E)$, then for every $S \subseteq V$ there is also an S -nice tree decomposition with the same width for G .

► **Lemma 4.9.** Let \mathcal{T} be a nice tree decomposition for a graph $G = (V, E)$, then for every $S \subseteq V$, the tree decomposition \mathcal{T} is S -nice.

Proof. TOPROVE 1 ◀

Note that, however, not every S -nice tree decomposition is a nice tree decomposition, since, for example, we allow S -bottom nodes to have an arbitrary amount of children that are not S -children. Furthermore, the S -children of an S -bottom nodes that admits join as its S -operation do not have to have the same bags as their parent.

Next, we show that if we know the S -trace of an S -bottom node in an S -nice tree decomposition, and we know which S -operation is admitted by that node, then we can determine the S -traces of the S -children of that node. We start with the S -operation $\text{introduce}(v)$.

► **Observation 4.10.** Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be an S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$. Let t be an S -bottom node in \mathcal{T} that admits the S -operation $\text{introduce}(v)$. Let t' be the S -child of t . Then we have that $L_{t'}^S = L_t^S$, $X_{t'}^S = X_t^S \setminus \{v\}$, and $R_{t'}^S = R_t^S \cup \{v\}$.

Proof. TOPROVE 2 ◀

Next, we consider the S -operation $\text{forget}(v)$.

► **Observation 4.11.** Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be an S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$. Let t be an S -bottom node in \mathcal{T} that admits the S -operation $\text{forget}(v)$. Let t' be the S -child of t . Then we have that $L_{t'}^S = L_t^S \setminus \{v\}$, $X_{t'}^S = X_t^S \cup \{v\}$, and $R_{t'}^S = R_t^S$.

Proof. TOPROVE 3 ◀

Finally, we consider the case that the admitted S -operation is $\text{join}(X_t^S, X_{t_1}^S, X_{t_2}^S, L_{t_1}^S, L_{t_2}^S)$.

► **Observation 4.12.** Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be an S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$. Let t be an S -bottom node in \mathcal{T} that admits the S -operation $\text{join}(X_t^S, X_{t_1}^S, X_{t_2}^S, L_{t_1}^S, L_{t_2}^S)$. Let t_1 and t_2 be the two S -children of t . Then we have the following:

- $L_{t_1}^S \cup L_{t_2}^S = L_t^S$, $L_{t_1}^S \cap L_{t_2}^S = \emptyset$, $L_{t_1}^S \neq \emptyset$, and $L_{t_2}^S \neq \emptyset$, and
- $R_{t_1}^S = R_t^S \cup (X_t^S \setminus X_{t_1}^S) \cup L_{t_2}^S$ and $R_{t_2}^S = R_t^S \cup (X_t^S \setminus X_{t_2}^S) \cup L_{t_1}^S$.

Proof. TOPROVE 4 ◀

From Observations 4.10–4.12 we can deduce the following simple corollary. This is the main reason that enables us to perform dynamic programming on the S -traces.

► **Corollary 4.13.** Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be an S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$. Let t be an S -bottom node in \mathcal{T} . If t has an S -child t' , then one of the following holds.

1. $L_{t'}^S \subset L_t^S$, or
2. $L_{t'}^S = L_t^S$ and $X_{t'}^S \subset X_t^S$.

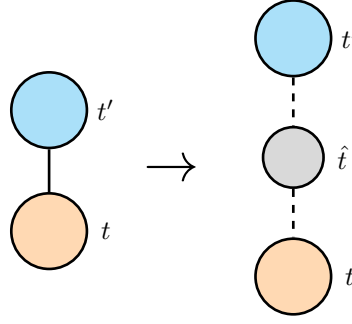
Formally, Corollary 4.13 allows us to define the following partial ordering for S -traces.

► **Definition 4.14 (Preceding S -Traces).** Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be an S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$. Let (L^S, X^S, R^S) and $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$ be two S -traces. We say that (L^S, X^S, R^S) is a direct predecessor of $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$ in \mathcal{T} if there is an S -bottom node in \mathcal{T} with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$ that has an S -child with S -trace (L^S, X^S, R^S) . The predecessor relation on S -traces in an S -nice tree decomposition \mathcal{T} is defined as the transitive closure of the direct predecessor relation of \mathcal{T} .

4.3 Modifications for Tree Decompositions

In several proofs, we will use several operations to modify S -nice tree decompositions. We introduce those modifications in this section.

First, we introduce two basic modifications. The first one, when applied to any rooted tree decomposition, ensures that afterwards, for any two nodes t, t' such that t' is the parent of t we have that $X_t \subseteq X_{t'}$ or $X_t \supseteq X_{t'}$, and $-1 \leq |X_t| - |X_{t'}| \leq 1$.



■ **Figure 2** Illustration of the modification **Normalize** (Modification 1). Nodes t (blue) and t' (orange) have bags $X_t, X_{t'}$ respectively with $X_t \setminus X_{t'} \neq \emptyset$ and $X_{t'} \setminus X_t \neq \emptyset$. A new node \hat{t} (gray) is inserted between t and t' with bag $X_{\hat{t}} = X_t \cap X_{t'}$. The dashed edges represent paths resulting from edge subdivisions where, informally speaking, vertices are inserted or removed one by one from the bags of the nodes along the path to ensure that the sizes of the bags of adjacent nodes differ by at most one.

Modification 1 (Normalize).

Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a rooted tree decomposition of a graph $G = (V, E)$. The modification **Normalize** applies the following changes to \mathcal{T} .

Repeat the following. Let t, t' be two nodes in \mathcal{T} such that t' is the parent of t , such that at least one of the following operations applies. Perform the first applicable operation.

- If $X_t \setminus X_{t'} \neq \emptyset$ and $X_{t'} \setminus X_t \neq \emptyset$, then subdivide the edge between t and t' in T and let \hat{t} be the new node. Set $X_{\hat{t}} = X_t \cap X_{t'}$ be its bag.
- If $|X_t| < |X_{t'}| - 1$, then apply $|X_{t'}| - |X_t| - 1$ subdivisions to the edge between t and t' in T . The bags of the new nodes are defined as follows.

Iterate over the new nodes starting at the child of t' . Set the current node's bag as the bag of its parent and then remove an arbitrary vertex from $X_{t'} \setminus X_t$ that is contained in the parent's bag. Continue with the bag of the child.

- If $|X_{t'}| < |X_t| - 1$, then apply $|X_t| - |X_{t'}| - 1$ subdivisions to the edge between t and t' in T . The bags of the new nodes are defined as follows.

Iterate over the new nodes starting at the parent of t . Set the current node's bag as the bag of its child and then remove an arbitrary vertex from $X_t \setminus X_{t'}$ that is contained in the child's bag. Continue with the bag of the parent.

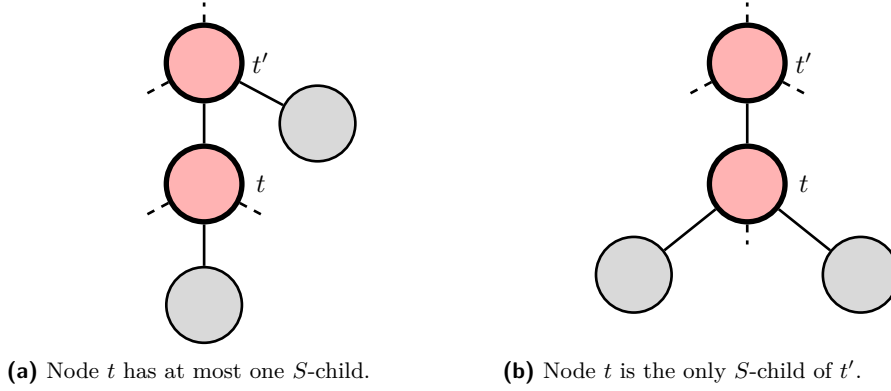
The modification **Normalize** is visualized in Figure 2. It is easy to observe that for every rooted tree decomposition \mathcal{T} , we have that after applying **Normalize** to it, it remains a rooted tree decomposition and gains the above-claimed properties. In fact, this modification (for $S = \emptyset$) is part of transforming any rooted tree decomposition into a nice tree decomposition [25, 42].

The second basic modification, intuitively, removes unnecessary full nodes from an S -nice tree decomposition.

Modification 2 (MergeFullNodes).

Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be an S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$ of width k . The modification **MergeFullNodes** applies the following changes to \mathcal{T} .

Repeat the following. Let t, t' be two nodes in \mathcal{T} , such that t' is the parent of t , $X_t = X_{t'}$ and $|X_t| = |X_{t'}| = k + 1$, that is, both bags are the same and full. If t has at most one S -child or t is the only S -child of t' , then remove t and connect all children of t to t' .



■ **Figure 3** Illustration of the modification **MergeFullNodes** (Modification 2). Nodes t and t' (red circles) have the same bags and the bag is full. This is indicated by the thick line. Nodes represented by gray circles are S -children of their respective parents. Further children that connected via dashed lines are not S -children. In both cases visualized in Figures 3a and 3b, nodes t and t' are merged.

The modification **MergeFullNodes** is visualized in Figure 3. We can observe the following. Let t, t' be two nodes in \mathcal{T} such that t' is the parent of t , and t has at most one S -child. If $X_t = X_{t'}$ and $|X_t| = |X_{t'}| = k + 1$, then in particular $X_t^S = X_{t'}^S$. If $L_t^S = \emptyset$, then Definition 4.7 we have that t does not have any S -children. It follows that merging t and t' does not add any new S -children to t' . Otherwise, we have that $L_t^S \setminus R_t^S \neq \emptyset$ and $R_{t'}^S \subseteq R_t^S$. It follows that $L_t^S \setminus R_{t'}^S \neq \emptyset$, and hence by Definition 4.7 we have that t is an S -child of t' . Again, it follows that merging t and t' does not add any new S -children to t' . We can conclude that **MergeFullNodes** preserves the property of tree decompositions of being S -nice. Clearly, it also does not increase the width.

Now we introduce two further modifications **MoveIntoSubtree** and **RemoveFromSubtree**. In contrast to the modifications introduced so far, **MoveIntoSubtree** and **RemoveFromSubtree** have some prerequisites and are not always applicable. The modification **MoveIntoSubtree**, intuitively, moves a M set of vertices (with $M \cap S = \emptyset$) into some subtree of the tree decomposition, if all neighbors of M are already in that subtree. The modification **RemoveFromSubtree**, intuitively, makes the opposite modification. It moves a set M of vertices out of a subtree if all neighbors of M are also out of the subtree. Formally, the modifications are defined as follows.

Modification 3 (MoveIntoSubtree).

Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be an S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$. Let t be a node in \mathcal{T} and let $M \subseteq V \setminus S$ such that $N(M) \subseteq V_t$. The modification **MoveIntoSubtree**(t, M) applies the following changes to \mathcal{T} .

1. Remove the vertices of M from each bag of a node that is not in T_t .
2. Let $M' = M \setminus V_t$. If $M' \neq \emptyset$, then let G' be the graph obtained from $G[N[M']]$ after adding edges between every two non-adjacent vertices in $N(M')$. Moreover, let \mathcal{T}' be some optimal tree decomposition for G' . By Lemma 3.3, there exists a node r in \mathcal{T}' such that $N(M') \subseteq X_r$. Root \mathcal{T}' at r and make r a child of t in \mathcal{T} .
3. Apply **Normalize** and **MergeFullNodes** to \mathcal{T} (Modifications 1 and 2).

Modification 4 (RemoveFromSubtree).

Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be an S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$. Let t be a node in \mathcal{T} and let $M \subseteq V \setminus S$ such that $N(M) \subseteq (V \setminus V_t) \cup X_{t'}$, where t' is the parent of t . The modification **RemoveFromSubtree**(t, M) applies the following changes to \mathcal{T} .

1. Remove the vertices of M from each bag of a node in T_t .
2. Let $M' = M \cap (V_t \setminus X_{t'})$. If $M' \neq \emptyset$, then let G' be the graph obtained from $G[N[M']]$ after adding edges between every two non-adjacent vertices in $N(M')$. Moreover, let \mathcal{T}' be some optimal tree decomposition for G' . By Lemma 3.3, there exists a node r in \mathcal{T}' such that $N(M') \subseteq X_r$. Root \mathcal{T}' at r and make r a child of t' in \mathcal{T} .
3. Apply **Normalize** and **MergeFullNodes** to \mathcal{T} (Modifications 1 and 2).

We can observe that both above-defined operations, when applied to an S -nice tree decomposition, produce another S -nice tree decomposition that does have at most the same width as the original one. Furthermore, it will be crucial that the modification do not change the “ S -trace structure” of the S -nice tree decomposition. To formalize this, we define *sibling* S -nice tree decompositions, which have the same predecessor relation (Definition 5.8) on their S -traces.

► **Definition 4.15** (Sibling Tree Decompositions). Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ and $\mathcal{T}' = (T', \{X'_t\}_{t \in V(T')})$ be two S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$. We say that \mathcal{T} and \mathcal{T}' are *siblings* if both tree decompositions have the same predecessor relation on S -traces.

Formally, we now show the following.

► **Observation 4.16.** Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be an S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$ with width k . Let t be a node in \mathcal{T} with parent t' , let $M \subseteq V \setminus S$ such that $N(M) \subseteq V_t$, and let $M' \subseteq V \setminus S$ such that $N(M') \subseteq V \setminus V_t$. The following holds.

- Let \mathcal{T}_{mod} be the result of applying **MoveIntoSubtree**(t, M) to \mathcal{T} . Then \mathcal{T}_{mod} is an S -nice tree decomposition for G with width at most k that is a sibling of \mathcal{T} .
- Let \mathcal{T}_{mod} be the result of applying **RemoveFromSubtree**(t, M') to \mathcal{T} . Then \mathcal{T}_{mod} is an S -nice tree decomposition for G with width at most k that is a sibling of \mathcal{T} .

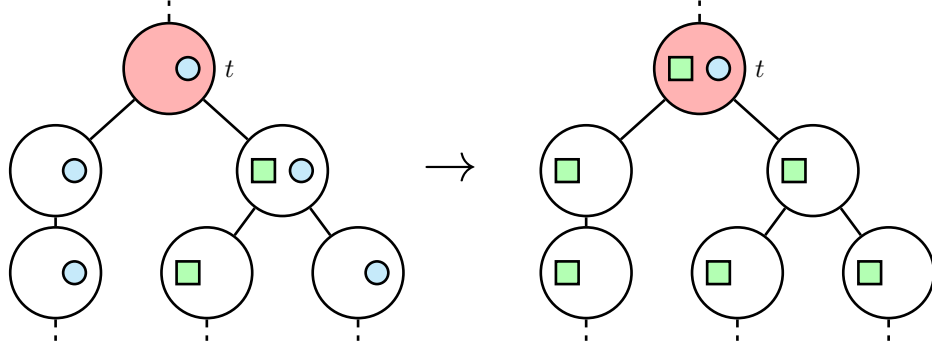
Proof. TOPROVE 5 ◀

Finally, we introduce two modifications **BringNeighborUp** and **BringNeighborDown** that, intuitively, move a neighbor of a vertex v into a specific node in order to ensure that we can safely remove v from a certain part of the tree decomposition.

Modification 5 (BringNeighborUp).

Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be an S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$ with width k . Let t be a node in \mathcal{T} such that $|X_t| \leq k$. Let $v \in X_t \setminus S$ such that $N(v) \cap (V_t \setminus X_t) = \{u\}$ for some $u \in V \setminus S$. The modification **BringNeighborUp**(t, v, u) applies the following changes to \mathcal{T} .

1. Replace v with u in all bags of nodes that are in T_t except X_t (since bags are sets, if a bag contains both v and u this implies that v is removed from the bag).
2. Add u to X_t .
3. Apply **Normalize** to \mathcal{T} (Modification 1).



■ **Figure 4** Illustration of the modification `BringNeighborUp` (Modification 5). Nodes t is represented by a red circle. Before the modification is applied, the bag t is not full. The small blue circle visualizes vertex $v \in X_t \setminus S$ and the green square vertex $u \in N(v) \setminus S$. The left side shows the configuration before the modification is applied, and the right side shows the configuration after the modification is applied.

Modification 6 (`BringNeighborDown`).

Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be an S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$ with width k . Let t be a node in \mathcal{T} such that $|X_t| \leq k$. Let $v \in X_t \setminus S$ such that $N(v) \cap (V \setminus V_t) = \{u\}$ for some $u \in V \setminus S$. The modification `BringNeighborDown`(t, v, u) applies the following changes to \mathcal{T} .

1. Replace v with u in all bags of nodes that are not in T_t (since bags are sets, if a bag contains both v and u this implies that v is removed from the bag).
2. Add u to X_t .
3. Apply `Normalize` to \mathcal{T} (Modification 1).

The modification `BringNeighborUp` is visualized in Figure 4. We can observe that both above-defined operations, when applied to an S -nice tree decomposition, produce another S -nice tree decomposition that does have at most the same width as and is a sibling of the original one.

► **Observation 4.17.** Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be an S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$ with width k . Let t be a node in \mathcal{T} such that $|X_t| \leq k$ and t is not an S -bottom node. Let $v \in X_t \setminus S$ such that $N(v) \cap (V_t \setminus X_t) = \{u\}$ for some $u \in V \setminus S$. Let $v' \in X_t \setminus S$ such that $N(v') \cap (V \setminus V_t) = \{u'\}$ for some $u' \in V \setminus S$. The following holds.

- Let \mathcal{T}_{mod} be the result of applying `BringNeighborUp`(t, v, u) to \mathcal{T} . Then \mathcal{T}_{mod} is an S -nice tree decomposition for G with width at most k that is a sibling of \mathcal{T} .
- Let \mathcal{T}_{mod} be the result of applying `BringNeighborDown`(t, v', u') to \mathcal{T} . Then \mathcal{T}_{mod} is an S -nice tree decomposition for G with width at most k that is a sibling of \mathcal{T} .

Proof. TOPROVE 6 ◀

4.4 Slim and Top-Heavy S -Nice Tree Decompositions

In this section, we introduce two special types of S -nice tree decomposition that are central to our algorithm. They are called *slim* and *top-heavy*. We first present the concept of slim S -nice tree decompositions. Intuitively, they do not contain nodes that unnecessarily have full bags. To this end, we first define *full join trees* of an S -nice tree decomposition, which,

intuitively, are subtrees of the tree decomposition where all nodes have the same (full) bag and admit a bigjoin operation.

► **Definition 4.18** (Full Join Tree). *Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be an S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$ with width k . A full join tree T' is a non-trivial, (inclusion-wise) maximal subtree of T such that there exist a set $X \subseteq V$ with $|X| = k + 1$ such each $t \in (T')$ has bag $X_t = X$.*

Furthermore, we introduce the following terminology. Let C be a connected component in $G[V']$ for some $V' \subseteq V$.

- If $V(C) \cap S = \emptyset$, then we call C an F -component.
- If $V(C) \cap S \neq \emptyset$, then we call C an S -component.

It is easy to see that every connected component of $G[V']$ falls into one of the above categories, for every choice of $V' \subseteq V$.

Now we are ready to define *slim* S -nice tree decompositions.

► **Definition 4.19** (Slim S -Nice Tree Decomposition). *Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be an S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$ with width k . We call \mathcal{T} slim if the following holds.*

1. The root of \mathcal{T} is not an S -bottom node.
2. For each S -bottom node t in \mathcal{T} that has two S -children, we have that either t is part of a full join tree, or both S -children and the parent of t have bags that are not full and not larger than the bag of t .
3. For each S -bottom node t in \mathcal{T} that has a bag which is not full, we have that for the parent t' of t it holds that $X_t = X_{t'}$ and t is the only S -child of t' .
4. For each S -bottom node t in \mathcal{T} that admits S -operation $\text{forget}(v)$ for some $v \in S$, we have that if the bag of the S -child t' of t is not full, then t' has at most one S -child t'' . If t'' exists, then it holds that $X_{t'} = X_{t''}$.
5. For each S -bottom node t in \mathcal{T} we have that if the bag of the parent t' of t is not full, then the parent of t' is not an S -bottom node or t' is the root.

Moreover, for each full join tree T' of \mathcal{T} , the following holds.

6. Each node t in $V(T')$ is an S -bottom node that has two S -children.
7. Let X denote the bag of nodes in $V(T')$, let t_r denote the root of T' , let t' denote the parent of t_r (if it exists), and let T'_C denote the set of S -children of nodes in $V(T')$ that are not contained in $V(T')$. For each $v \in X \setminus S$ there exist three different vertices $u_1, u_2, u_3 \in N(v) \setminus X$ and three different nodes $t_1, t_2, t_3 \in T'_C \cup \{t'\}$ such that
 - For all $1 \leq i \leq 3$, vertex u_i is connected to S in $G - X$.
 - For all $1 \leq i \leq 3$, if $t_i \neq t'$, then vertex u_i is contained in V_{t_1} . Otherwise vertex u_i is contained in $V \setminus V_{t_r}$.

Before we show that we can make any S -nice tree decomposition slim, we give some intuition on why the properties of slim S -nice tree decompositions are desirable for us.

1. Condition 1 allows us to assume that all S -bottom nodes have a parent. This is important since, informally speaking, we want to remove all vertices that do not need to be in nodes that are below some S -bottom node, and move them above it. We formalize this later in this section when we define top-heavy tree decompositions.
2. Condition 2 allows us to assume that S -bottom nodes with two S -children, that is, the ones that admit S -operation join, are either part of a full join tree, or both the S -children and the parent do not have full bags and those bags. This is important since we will treat these two cases differently in our algorithm. In particular in the latter case, that is, if an S -bottom node admits the S -operation join and it is not part of a full join tree, we need

the property that both the S -children and the parent do not have full bags and that the bags are subsets of the bag of the S -bottom node.

3. Condition 3 allows us to assume that whenever an S -bottom node does not have a full bag, then the bag of its parent is also not full, and in particular, it is the same bag. This will make some proofs easier to achieve and easy to obtain by subdividing the edge between the S -bottom node and its parent, and giving the new node the same bag as the S -bottom node.
4. Condition 4 allows us to assume that whenever an S -bottom node admits the S -operation *forget* and its S -child (which always has a larger bag) is not full, then the bag of the S -child of the S -child (if it exists) is also not full. As with the previous condition, this is easy to achieve by edge subdivision and it simplifies some of our proofs.
5. Condition 5 allows us to assume that in a directed path corresponding to an S -trace (L^S, X^S, R^S) with $L^S \neq \emptyset$, if the parent of the bottom node has a non-full bag, then the top node is not the parent of the bottom node. This will simplify many steps in our algorithm.
6. Condition 6 allows us to assume that every S -bottom node that is part of a full join tree admits the S -operation *join*.
7. Condition 7 is the most technical one and also the most important one. Essentially it allows us to assume that every vertex $v \in V \setminus S$ that is contained in a bag X of a node that is part of a full join tree is a neighbor of at least three different S -components in $G - X$. This will be crucial for establishing the running time bound of a subroutine of our algorithm.

Now we show that if a graph G admits an S -nice tree decomposition with width k , then G also admits a slim S -nice tree decomposition with width k .

► **Lemma 4.20.** *Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be an S -nice tree decomposition of a graph $G = (V, E)$ with width k and some $S \subseteq V$. Then there exist a slim S -nice tree decomposition \mathcal{T}' of G with width at most k .*

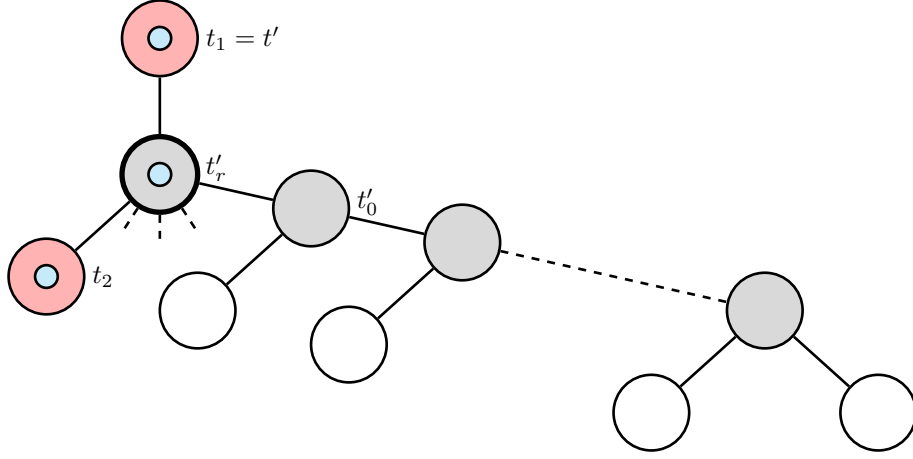
To prove Lemma 4.20, we give the following two procedures and prove that they transform an S -nice tree decomposition \mathcal{T} for a graph $G = (V, E)$ and some $S \subseteq V$ with width k , into a slim S -nice tree decomposition for G with width k .

Modification 7 (MakeSlimOne).

Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be an S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$ with width k . The modification **MakeSlimOne** applies the following changes to \mathcal{T} .

1. Apply the modification **MergeFullNodes** to \mathcal{T} (Modification 2).
2. If the root t_r of \mathcal{T} is an S -bottom node, then add a new node t' to \mathcal{T} and make it the parent of t_r . Set $X_{t'} = \emptyset$. Apply **Normalize**. If the new root t'_r is still an S -bottom node, add a new node t'' with $X_{t''} = \emptyset$ to \mathcal{T} and make it the parent of t'_r .
3. For each S -bottom node t in \mathcal{T} that has two S -children, if t is not contained in any full join tree and the parent t_p of t has a full bag, then subdivide the edge between t and t_p and let t' be the new node. Set $X_{t'} = X_t$.
4. For each full join tree T' in \mathcal{T} perform the following steps once. Let X denote the bag of nodes in $V(T')$, let t_r denote the root of T' , let t' denote the parent of t_r (if it exists), and let T'_C denote the set of S -children of nodes in $V(T')$ that are not contained in $V(T')$.
 - a. For each F -component C in $G - X$ such that $V(C) \cap V_{t^*} \neq \emptyset$ for some $t^* \in T'_C$, or, if t' exists, $V(C) \cap ((V \setminus V_{t'}) \cup X_{t'}) \neq \emptyset$, add a new child t_0 to t_r with bag X and apply **MoveIntoSubtree**($t_0, V(C)$) (Modification 3). Remove t_0 and connect all children of t_0 to t_r .
 - b. Assume there exist $v \in X \setminus S$ such that $N(v) \subseteq X$. Then remove v from the bags of all nodes except t_r .
 - c. Assume there exist $v \in X \setminus S$, $u_1, u_2 \in N(v) \setminus X$, and nodes $t_1, t_2 \in T'_C \cup \{t'\}$ with $t_1 \neq t_2$ and $t' \neq t_2$ such that the following holds. For all $t'' \in (T'_C \cup \{t'\}) \setminus \{t_1, t_2\}$ if $t'' \neq t'$, then vertex $N(v) \cap (V_{t''} \setminus X) = \emptyset$. Otherwise, $N(v) \cap (V \setminus V_{t_r}) = \emptyset$. Then make the following modification.
 We replace T' with a rooted path P (that is, the path is rooted at one of its endpoints) on $|T'_C| - 1$ nodes as follows. Remove T' from \mathcal{T} . Let t'_r be the root of P . Make t' (if it exists) the parent of t'_r , otherwise t'_r is the root of \mathcal{T} .
 - If $t_1 = t'$, then t_2 is a child of t'_r . Each child of a node in T' that is not an S -child is a child of t'_r . Each node in $T'_C \setminus \{t_2\}$ is a child of exactly one inner node of P and two nodes are children of the leaf of P . Set the bags of all nodes in P to X . Denote t'_0 the child of t'_r in P . Apply **RemoveFromSubtree**($t'_0, \{v\}$) (Modification 4). For an illustration see Figure 5.
 - Otherwise, t_1 and t_2 are children of the leaf of P . Each child of a node in T' that is not an S -child is a child of the leaf of P . Each node in $T'_C \setminus \{t_1, t_2\}$ is a child of exactly one of the remaining nodes of P . Set the bags of all nodes in P to X . Denote t'_ℓ the leaf of P . Apply **MoveIntoSubtree**($t'_\ell, \{v\}$).

Note that since Step 3 of **MakeSlimOne** can be applied at most once to each S -bottom node and Step 4 of **MakeSlimOne** is applied to each full join tree once, we have that the modification always terminates after a finite number of steps. Informally speaking, **MakeSlimOne** takes care of all conditions of Definition 4.19 except Conditions 3, 4, and 5, and partially Condition 2. Concerning Condition 2: **MakeSlimOne** does not guarantee that if an S -bottom node t has two S -children and is not part of a full join tree, then the bags of the parent and the children are not larger than the bag of t . For those, we introduce a second modification **MakeSlimTwo**. The reason for this is that we can use **MakeSlimTwo** after applying modifications **MoveIntoSubtree**, **RemoveFromSubtree**, **BringNeighborUp**, or **BringNeighborDown** (Modifications 3–6) to a slim S -nice treedecompositoin to ensure that the tree decomposition remains slim.



■ **Figure 5** Illustration of Step 4 in the procedure “Transformation into Slim S -Nice Tree Decomposition”. The gray circles represent nodes of the path P with which the full join tree T' is replaced. The red circles illustrate t_1 and t_2 . The remaining circles illustrate nodes in $T'_C \setminus \{t_2\}$. The dashed lines below t'_r indicate that all non- S -children of nodes in T' are attached to this node. The case where $t_1 = t'$ is shown. Vertex v is visualized with a small blue circle. It is removed from all bags of nodes of P except t'_r . The bag of t'_r remains full, this is visualized by the thick line.

Modification 8 (MakeSlimTwo).

Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be an S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$ with width k . The modification **MakeSlimTwo** applies the following changes to \mathcal{T} .

1. For each S -bottom node t in \mathcal{T} that has a bag which is not full, if for the parent t_p of t it holds that $X_t \neq X_{t_p}$, then subdivide the edge between t_p and t and let t' be the new node. Set $X_{t'} = X_t$. If t has two S -children t_1 and t_2 and for some t_i with $i \in \{1, 2\}$ it holds that $|X_{t_i}| > |X_t|$, then subdivide the edge between t_1 and t and let t'' be the new node. Set $X_{t''} = X_t$.
2. For each S -bottom node t in \mathcal{T} that admits S -operation **forget**(v) for some $v \in S$, if the bag of the S -child t' of t is not full and t' has an S -child t'' with $X_{t'} \neq X_{t''}$, then subdivide the edge between t' and t'' and let t''' be the new node. Set $X_{t'''} = X_{t'}$.
3. For each S -bottom node t in \mathcal{T} , if the bag of the parent t_p of t is not full and t_p is a child of an S -bottom node, then subdivide the edge between t_p and its parent and let t' be the new node. Set $X_{t'} = X_{t_p}$.

Now we are ready to prove Lemma 4.20.

Proof. TOPROVE 7

Furthermore, we can observe that all modifications introduced in Section 4.3 preserve the property of tree decomposition to be slim.

► **Observation 4.21.** Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a slim S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$ with width k . Let t be a node in \mathcal{T} with parent t_p , let $M \subseteq V \setminus S$ such that $N(M) \subseteq V_t$, and let $M' \subseteq V \setminus S$ such that $N(M') \subseteq (V \setminus V_t) \cup X_{t_p}$. Let t' be a node in \mathcal{T} such that $|X_{t'}| \leq k$ and t' is not parent of an S -bottom node. Let $v \in X_{t'} \setminus S$ such that $N(v) \cap (V_t \setminus X_{t'}) = \{u\}$ for some $u \in V \setminus S$. Let $v' \in X_{t'} \setminus S$ such that $N(v') \cap (V \setminus V_{t'}) = \{u'\}$ for some $u' \in V \setminus S$. The following holds.

- Let \mathcal{T}_{mod} be the result of applying $\text{MoveIntoSubtree}(t, M)$ and then MakeSlimTwo to \mathcal{T} . Then \mathcal{T}_{mod} is a slim S -nice tree decomposition for G with width at most k that is a sibling of \mathcal{T} .
- Let \mathcal{T}_{mod} be the result of applying $\text{RemoveFromSubtree}(t, M')$ and then MakeSlimTwo to \mathcal{T} . Then \mathcal{T}_{mod} is a slim S -nice tree decomposition for G with width at most k that is a sibling of \mathcal{T} .
- Let \mathcal{T}_{mod} be the result of applying $\text{BringNeighborUp}(t', v, u)$ and then MakeSlimTwo to \mathcal{T} . Then \mathcal{T}_{mod} is a slim S -nice tree decomposition for G with width at most k that is a sibling of \mathcal{T} .
- Let \mathcal{T}_{mod} be the result of applying $\text{BringNeighborDown}(t', v', u')$ and then MakeSlimTwo to \mathcal{T} . Then \mathcal{T}_{mod} is a slim S -nice tree decomposition for G with width at most k that is a sibling of \mathcal{T} .

Proof. TOPROVE 8 ◀

Now we formally define the second special type of S -nice tree decompositions. They are called *top-heavy* (for some node t) and, intuitively, have the property that if the modifications introduced in Section 4.3 are applied to the tree decomposition outside of the subtree rooted at t , then the bags in the subtree rooted at t are unaffected.

► **Definition 4.22 (Top-Heavy).** Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a slim S -nice tree decomposition of a graph $G = (V, E)$ with width k and some $S \subseteq V$. Let t be a parent of an S -bottom node in \mathcal{T} with $|X_t| \leq k$. We call \mathcal{T} *top-heavy* for t if the following holds.

1. For each S -bottom node t' in \mathcal{T} that is a descendant of t and for each F -component C in $G - X_{t'}$ we have that $V(C) \cap V_{t''} = \emptyset$ for each S -child t'' of t' .
- Moreover, the following holds with respect to Modifications 3–6. Let t^* be a parent of an S -bottom node and contained in T_t . Let t' be a node in \mathcal{T} that is not contained in T_{t^*} .
2. If there is $M \subseteq V \setminus S$ such that $N(M) \subseteq V_{t'}$, then applying $\text{MoveIntoSubtree}(t', M)$ to \mathcal{T} does not change T_{t^*} or any bag of a node in T_{t^*} .
 3. If there is $M \subseteq V \setminus S$ such that $N(M) \subseteq (V \setminus V_{t'}) \cup X_{t'_p}$, where t'_p is the parent of t' , then applying $\text{RemoveFromSubtree}(t', M)$ to \mathcal{T} does not change T_{t^*} or any bag of a node in T_{t^*} .
 4. If $|X_{t'}| \leq k$ and there is $v \in X_{t'} \setminus S$ such that $N(v) \cap (V_{t'} \setminus X_{t'}) = \{u\}$ for some $u \in V \setminus S$, then applying $\text{BringNeighborUp}(t', v, u)$ to \mathcal{T} does not change T_{t^*} or any bag of a node in T_{t^*} .
 5. If $|X_{t'}| \leq k$ and there is $v \in X_{t'} \setminus S$ such that $N(v) \cap (V \setminus V_{t'}) = \{u\}$ for some $u \in V \setminus S$, then applying $\text{BringNeighborDown}(t', v, u)$ to \mathcal{T} does not change T_{t^*} or any bag of a node in T_{t^*} .

We show that if a graph G admits a slim S -nice tree decomposition with width k , then we can transform it into a sibling slim S -nice tree decomposition for G with width k that is top-heavy for a node t . Formally, we prove the following

► **Lemma 4.23.** Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a slim S -nice tree decomposition of a graph $G = (V, E)$ with width k and some $S \subseteq V$. Then for every node t in \mathcal{T} with $|X_t| \leq k$ that is a parent of an S -bottom node there exist a slim S -nice tree decomposition \mathcal{T}' of G with width at most k that is top-heavy for t and that is a sibling of \mathcal{T} .

To prove Lemma 4.23, we give the following procedure and prove that it can be used to transform a slim S -nice tree decomposition \mathcal{T} for a graph $G = (V, E)$ and some $S \subseteq V$ with width k , into a slim top-heavy S -nice tree decomposition for G with width k that is a sibling of the original tree decomposition.

Modification 9 (MakeTopHeavy).

Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a slim S -nice tree decomposition of a graph $G = (V, E)$ and some $S \subseteq V$ with width k . Let t be a parent of an S -bottom node in \mathcal{T} such that $|X_t| \leq k$. The modification **MakeTopHeavy**(t) applies the following changes to \mathcal{T} .

Subdivide the edge between t and its parent and let t' be the new node. Set $X_{t'} = X_t$. Repeat the first applicable step until none of the steps applies.

1. Apply **MakeSlimTwo**.
2. For each S -bottom node t' in \mathcal{T} that is an S -child of t and each F -component C in $G - X_{t'}$, apply **RemoveFromSubtree**($t', V(C)$) for each S -child t'' of t' .
3. If there is an F -component C in $G - X_t$ such that $V(C) \cap V_t \neq \emptyset$, then apply **RemoveFromSubtree**($t, V(C)$) (Modification 4). Subdivide the edge between t and t' and let t'' be the new node. Set $X_{t''} = X_t$ and rename t'' to t' .
4. If there is $v \in X_t \setminus S$ with $N(v) \subseteq (V \setminus V_t) \cup X_{t'}$, then apply **RemoveFromSubtree**($t, \{v\}$) (Modification 4). Subdivide the edge between t and t' and let t'' be the new node. Set $X_{t''} = X_t$ and rename t'' to t' .
5. If there is $v \in X_{t'} \setminus S$ and $u \in V \setminus S$, such that $N(v) \cap (V_{t'} \setminus X_{t'}) = \{u\}$, then apply **BringNeighborUp**(t', v, u) (Modification 5). Subdivide the edge between t and t' and let t'' be the new node. Set $X_{t''} = X_t$ and rename t'' to t' .

We remark that the application of **MakeSlimTwo** in **MakeTopHeavy**(t) is only necessary to ensure that after applying **MakeTopHeavy**(t) to a slim S -nice tree decomposition, the tree decomposition remains slim. Note that since each application of the a step of **MakeTopHeavy**(t) removes a vertex from the subtree rooted at t , we have that the modification always terminates after a finite number of steps. Now we are ready to prove Lemma 4.23.

Proof. TOPROVE 9 ◀

The proof of Lemma 4.23 also implies the following.

► **Lemma 4.24.** *Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a slim S -nice tree decomposition of a graph $G = (V, E)$ with width k and some $S \subseteq V$ that is top-heavy for some node t in \mathcal{T} with $|X_t| \leq k$ that is parent of an S -bottom node. Let t' be a node in T_t with $|X_{t'}| \leq k$ that is parent of an S -bottom node. Then, when applying **MakeTopHeavy**(t') to \mathcal{T} , none of the steps of **MakeTopHeavy**(t') applies.*

► **Lemma 4.25.** *Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a slim S -nice tree decomposition of a graph $G = (V, E)$ with width k and some $S \subseteq V$. Let t be some node in \mathcal{T} with $|X_t| \leq k$ that is parent of an S -bottom node. If \mathcal{T} is top-heavy for each t' that is a descendant of t with $|X_{t'}| \leq k$ and that is parent of an S -bottom node, and none of the steps of **MakeTopHeavy**(t) applies, then \mathcal{T} is top-heavy for t .*

5 The Dynamic Programming Algorithm

In this section, we present the main dynamic programming table. Assume from now on that we are given a connected graph $G = (V, E)$ and a non-negative integer k that form an instance of **TREewidth**. Let the vertices in V be ordered in an arbitrary but fixed way, that is, $V = \{v_1, v_2, \dots, v_n\}$. Furthermore, we denote with $S \subseteq V$ a minimum feedback vertex set of G . We assume that $k \leq |S|$, since we can trivially obtain a tree decomposition with width $|S| + 1$ by taking any tree decomposition with width one for $G - S$ and adding S to every bag. Hence, if $k > |S|$, then (G, k) is a yes-instance of **TREewidth**. For the rest of

the document, we treat $G = (V, E)$, S , and k as global variables that all algorithms have access to.

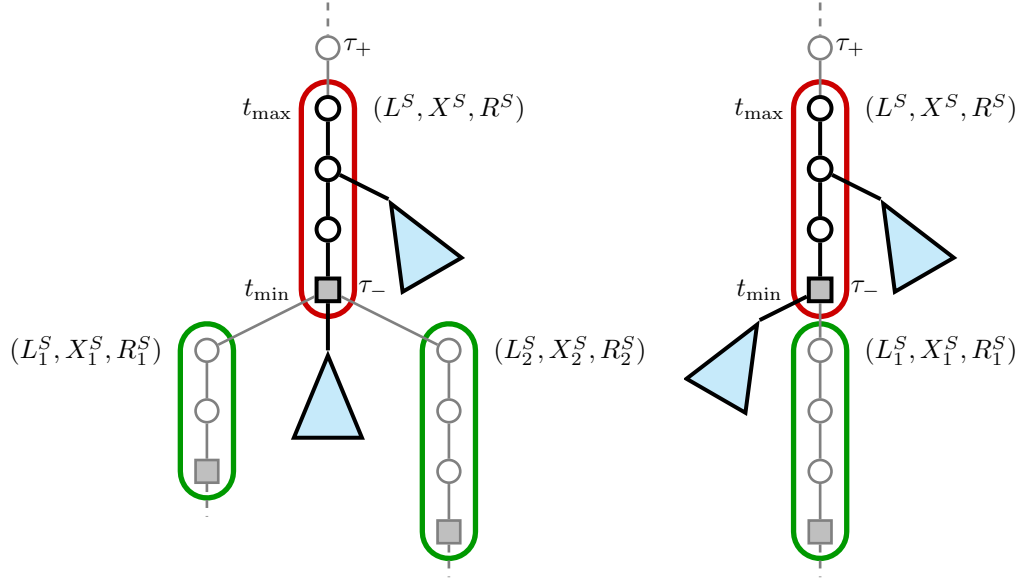
Roughly speaking, the algorithm we propose to prove Theorem 1.1 is a dynamic program on the potential directed paths of the S -traces (Definition 4.1) of a slim top-heavy S -nice tree decomposition (Definitions 4.8, 4.19, and 4.22) of G . By Lemmas 4.20 and 4.23, we have that if (G, k) is a yes-instance, then such a tree decomposition of width at most k exists. We remark that the main structure of our dynamic program table is similar to the one of Chapelle et al. [33]. However, there are major differences in how the table entries are computed. Informally speaking, a state of the dynamic program consists of the following elements.

- An S -trace (L^S, X^S, R^S) .
- The “extended” S -Operation τ_+ of the parent of the top node t_{\max} of the directed path of the S -trace.
 - If t_{\max} is the root, then τ_+ is the *dummy S -operation* void.
 - If $L^S = \emptyset$, then τ_+ is the “extended” S -operation of an S -bottom node that has an S -child with S -trace (L^S, X^S, R^S) .
- The “extended” S -Operation τ_- of the bottom node t_{\min} of the directed path of the S -trace.
 - If $L^S = \emptyset$, then τ_- is the *dummy S -operation* void.

Here, we use extended versions of the S -operations introduced in Definition 4.8. Those will contain additional information that we need to compute the entries of the dynamic program. In the following, we give some intuition on why we need this additional information, and then we give formal definitions.

Given the above-described information, the main strategy of our algorithm is to determine (candidates for) the bags X_{\max} and X_{\min} of the top node t_{\max} and the bottom node t_{\min} of the directed path of the S -trace, respectively. Furthermore, we want to identify (candidates for) the bag X_p of the parent of t_{\max} and (candidates for) a set X_{path} of vertices that we want to add to bags of the directed path of the S -trace or subtrees of the tree decomposition that are rooted at children of nodes in the directed path (that are not S -children). Using the sets X_{\max} , X_p , X_{\min} , and X_{path} , we compute the “local treewidth” of the directed path of the S -trace. Using the dynamic programming table, we look up the “partial treewidth” of the subgraphs “below” the S -children of the S -bottom node of the directed path. Using this information, we determine the partial treewidth of the subgraph below the top node of the directed path. For a visualization see Figure 6.

Informally speaking, one of the main difficulties in this approach is handling S -bottom nodes that are contained in full join trees (Definition 4.18), that is, their parent and S -children potentially have full bags, and at least one of them has. As a simple example that should convey some intuition consider the case where an S -bottom node has join as its S -operation. Let t_{\min} be such a node and let t_1, t_2 be its two S -children. In our algorithm, we wish to determine a candidate for the bag X_{\min} . However, if we face a yes-instance, a solution may use a different candidate X_{\min}^* for the bag of t_{\min} . In order to prove correctness, we have to argue that we can transform the solution tree decomposition (without increasing its width) to the one that we are computing tree decomposition. Assume two vertices $u, v \in V$ are adjacent, and in a solution S -nice tree decomposition we have that $u, v \in V_{t_1}$ and $v \in X_{\min}^*$. However, we chose X_{\min} such $u \in X_{\min}$ and $u, v \in V_{t_2}$. For a visualization, see Figure 7. Now we have to argue that we can do a transformation such as the one visualized in Figure 7 (from left to right). However, the transformation increases the size of the bag of t_2 . We have to make sure that this does not increase the width of the tree decomposition. If the bag of



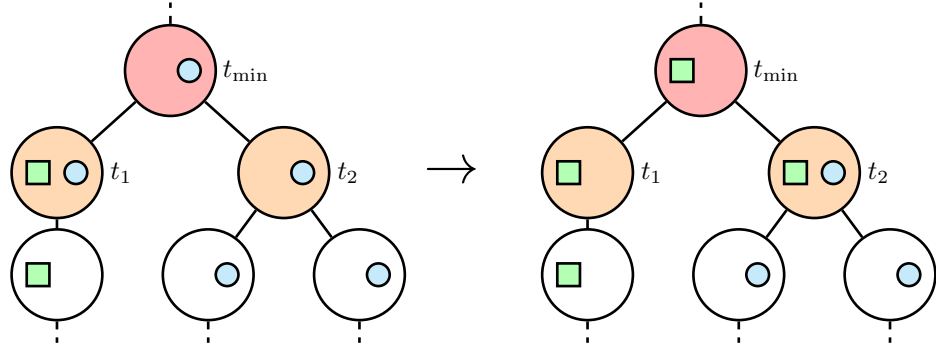
(a) S -operation τ_- of bottom t_{\min} node is $\text{join}(X^S, X_1^S, X_2^S, L_1^S, L_2^S)$. (b) S -operation τ_- of bottom node t_{\min} is $\text{introduce}(v)$ or $\text{forget}(v)$.

■ **Figure 6** Illustrations of the main idea of the dynamic programming algorithm. Let (L^S, X^S, R^S) be an S -trace. Let τ_+ be the S -operation of the parent of the top node t_{\max} of the directed path of the S -trace. Let τ_- be the S -operation of the bottom node t_{\min} of the directed path of the S -trace. The directed path of the S -trace L^S, X^S, R^S is surrounded by a red line. The S -bottom nodes are depicted as gray squares. The directed paths of the S -traces of S -children of the S -bottom node t_{\min} , for which we look up information in the dynamic programming table, are surrounded by green lines. The blue triangles illustrate parts of the tree decompositions where all nodes have S -traces with $L^S = \emptyset$, and hence those nodes are not part of any directed paths of S -traces. Furthermore, the roots of the blue triangles are not S -children of their respective parent nodes. The width of the part of the tree decomposition drawn in thick black lines is computed directly from the bags X_{\max} and X_{\min} of the top node t_{\max} and the bottom node t_{\min} of the directed path, respectively, and the set X_{path} of vertices that we want to add to bags in the directed path or subtrees illustrated by blue triangles. Subfigure 6a shows the situation where the S -bottom node t_{\min} has two S -children. Subfigure 6b shows the situation where the S -bottom node t_{\min} has one S -child.

node t_2 in the solution tree decomposition is not full, then we can perform the transformation illustrated in Figure 7.

If the bag of node t_2 in the solution tree decomposition is full (which may be the case if t_{\min} is contained in a full join tree), then we cannot perform the transformation. Then we face a situation as illustrated in Figure 8. Here, informally speaking, we need to make a transformation that changes the size of a bag that is (potentially) far away from t_{\min} in the tree decomposition. In order to be able to do this, we need to identify this situation and treat it differently. To this end, we introduce a new S -operation **bigjoin**. Informally speaking, if a slim top-heavy S -nice tree decompositions, there is an S -bottom node t that admits a join as its S -operation, then from now on we say that node t admits a **bigjoin** as its S -operation.

Finally, notice that X_{\max} is the bag of an S -child of an S -bottom node. To determine X_{\max} , we need to include additional information in the extended version of the S -operations, that help us determine the bags of S -children of the nodes that admit the extended S -operations. We remark that all additional information contained in the extended versions of S -operations is only relevant for the computation of the “local treewidth” of the directed



■ **Figure 7** Two illustrations (left and right) of the situations when t_{\min} is an S -bottom node of an S -nice tree decomposition that has join as its S -operation. The two S -children of t_{\min} are denoted t_1 and t_2 . The red circle visualizes the bag of t_{\min} and the orange circles visualize the bags of t_1 and t_2 . The green square visualizes a vertex $u \in V$ and the small blue circle a vertex $v \in V$ such that $\{u, v\} \in E$. The left side illustrates part of a tree decomposition where, in particular, u and v are both contained in the bag of t_1 , v is contained in the bag of t_{\min} , and u is not contained in the bag of t_{\min} . The right side illustrates part of a tree decomposition where, in particular, u and v are both contained in the bag of t_2 , u is contained in the bag of t_{\min} , and v is not contained in the bag of t_{\min} .

path of the S -trace. We give formal definitions of the extended S -operations in the next section.

5.1 Extended S -Operations

In this section, we describe how we extend the S -operations introduced in Definition 4.8. Since we are interested in slim S -nice tree decomposition, these operations will only be defined on slim S -nice tree decompositions. We do not need the “top-heavy” property for the definitions.

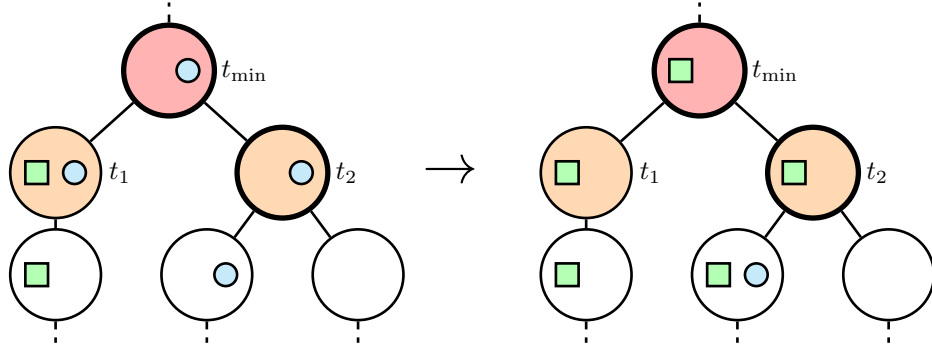
First of all, recall that we allow the dummy S -operation void if the top node is the root or if the bottom node does not have any S -children. Furthermore, as discussed before, we introduce a *big* version **bigjoin** of the S -operation join. When we compute bags for S -bottom nodes that admit this operation, we will have several possible candidates for the bag, and an additional bit string s encodes which one we are considering. Furthermore, we need the additional information on which vertex is (potentially) missing in each of the bags of the S -children or the parent, in the case that they are not full. This is encoded with three integers d_0 , d_1 and d_2 . We have the following big S -operation.

- **bigjoin**($X^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2$) with some vertex sets $X^S, X_1^S, X_2^S, L_1^S, L_2^S \subseteq S$ and some bit string $s \in \{0, 1\}^{2^{\text{fwn}(G)+1}}$ and $d_0, d_1, d_2 \in \{0, \dots, k+1\}$.

In order to define which S -bottom nodes admit the above-described big S -operations, we introduce a function that “decodes” its bag. Let

$$\text{bigbag} : 2^S \times \{0, 1\}^{2^{\text{fwn}(G)+1}} \rightarrow 2^V$$

be a function which takes as input a subset of S (which will be the set X^S of the S -bottom node that admits the big S -operation) and the bit string s that encodes which bag candidate is chosen. This will always be a full bag. The function is formally defined in Section 6. We additionally need to define a function that determines the bags of the children using the



■ **Figure 8** Two illustrations (left and right) of the situations when t_{\min} is an S -bottom node of an S -nice tree decomposition that has join as its S -operation. For further information on the visualization see the description of Figure 7. Additionally, thick circles indicate that the visualized bags are full. Furthermore, in this figure, the right side illustrates part of a tree decomposition where, in particular, u is contained in the bag of t_2 , v is not contained in the bag of t_2 , u is contained in the bag of t_{\min} , and v is not contained in the bag of t_{\min} , and u and v both are contained in the bag of the child of t_2 .

additional integers d_1 and d_2 . Let

$$\text{subbag} : 2^V \times \{0, \dots, k+1\} \rightarrow 2^V$$

be a function that takes as input a subset of V (which will be the bag of the S -bottom node that admits the S -operation **bigjoin**) and an additional integer that encodes which vertex is missing in the bag of a child (if the integer is zero, this will encode that the bag of the child is the same). Formally, $\text{subbag}(X, 0) = X$ and $\text{subbag}(X, i) = X \setminus \{v\}$, where v is at the i th ordinal position in X (recall that V is ordered in an arbitrary but fixed way).

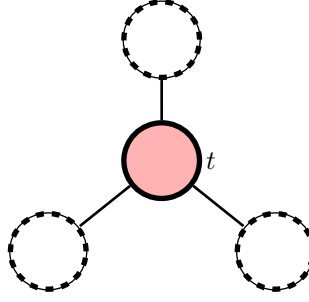
► **Definition 5.1** (**bigjoin**). Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a slim S -nice tree decomposition of G . Let t be an S -bottom node in \mathcal{T} that is contained in a full join tree and admits S -operation $\text{join}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S)$. Let t_1, t_2 be the two S -children of t and let t_0 be the parent of t . If $X_t = \text{bigbag}(X_t^S, s)$ and for all $0 \leq i \leq 2$ we have $X_{t_i} = \text{subbag}(X_t, d_i)$, then we say that node t admits S -operation $\text{bigjoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)$.

The big S -operation **bigjoin** is illustrated in Figure 9. Note that at this point, we cannot decide whether an S -bottom node admits a **bigjoin** S -operation since we have not defined the function **bigbag** yet. Informally speaking, there will be sufficiently many options for the bit string s such that for every possible bag that a node t that meets the conditions in Definition 5.1 has, the function **bigbag** maps the X_t^S and the bit string s to that bag. Formally, we will show the following.

► **Lemma 5.2.** Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a slim S -nice tree decomposition of G . Let t be an S -bottom node in \mathcal{T} that is contained in a full join tree and admits S -operation $\text{join}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S)$. Then there exists $s \in \{0, 1\}^{2^{\text{fvn}(G)+1}}$ and $d_0, d_1, d_2 \in \{0, \dots, k+1\}$ such that node t admits S -operation $\text{bigjoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)$.

We postpone the proof of Lemma 5.2 so Section 6.1, where we also formally define the function **bigbag**.

Now we introduce “small” S -operations for S -bottom nodes that are not contained in full join trees. Similar as in the case of the big operations, we need some additional information that allows us to compute bags of S -children or parent nodes. Here, we add an additional



■ **Figure 9** Illustration of the big S -operation **bigjoin** (Definition 5.1). Node t admits $\text{bigjoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)$. The red circle represents S -bottom node t . The remaining circles are the parent and two S -children, respectively. Children of t that are not S -children are not depicted. Thick circles represent nodes whose bags are full and dashed circles represent nodes whose bags are potentially full. Since t is contained in a full join tree, at least one of the nodes represented by dashed circles has a full bag.

integer d that will guide our algorithm to a specific bag candidate. The main reason why we add this information to the state is to guarantee that our algorithm computes the same bag for the bottom node of the directed path as we computed in the predecessor states for the parent of the top node. We have the following *small* S -operations.

- **smallintroduce** (v, d) with some vertex $v \in S$ and some integer $d \in \{1, \dots, (n+1)^3\}$.
- **smalljoin** $(X^S, X_1^S, X_2^S, L_1^S, L_2^S, d)$ with some vertex sets $X^S, X_1^S, X_2^S, L_1^S, L_2^S \subseteq S$ and some integer $d \in \{1, \dots, (n+1)^3\}$.

In order to define which S -bottom nodes admit the above-described small S -operations, we again use a function that “decodes” its bag. Let

$$\text{smallbag} : 2^S \times \mathbb{N} \rightarrow 2^V$$

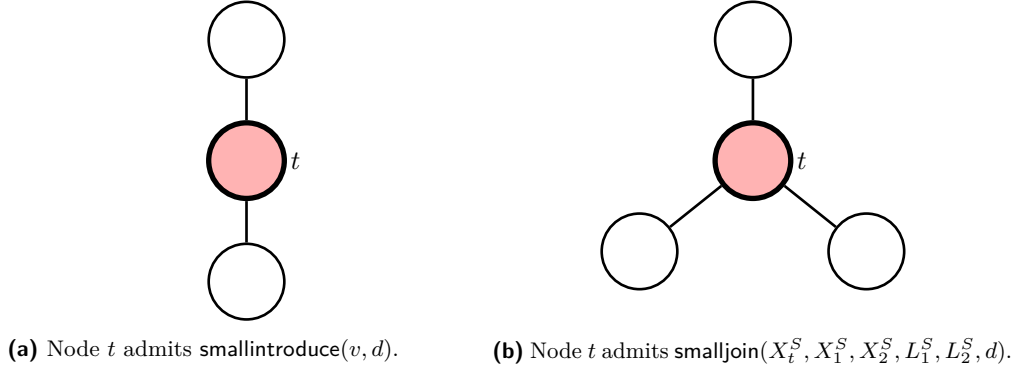
be a function which takes as input a subset of S and the number d that will guide the algorithm to a specific bag. The function is formally defined in Section 6. We additionally need to define a function that determines the bags of the children. We call this function **nbag** (for “neighboring bag”). It takes as input the bag, the integer d , and a flag from $\{P, L, R\}$ indicating whether we wish to compute the bag of the parent, the left child, or the right child (if the node only has one child, we consider this child to be a left child). Let

$$\text{nbag} : 2^V \times \mathbb{N} \times \{P, L, R\} \rightarrow 2^V.$$

We defer a formal definition of this function to Section 6.

► **Definition 5.3** (**smallintroduce**). Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a slim S -nice tree decomposition of G . Let t be an S -bottom node in \mathcal{T} that admits S -operation **introduce** (v) . Let t_1 be the S -child of t and let t_0 be the parent of t . If $X_t = \text{smallbag}(X_t^S, d)$, $X_{t_0} = \text{nbag}(X_t, d, P)$, and $X_{t_1} = \text{nbag}(X_t, d, L)$, then we say that node t admits S -operation **smallintroduce** (v, d) .

► **Definition 5.4** (**smalljoin**). Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a slim S -nice tree decomposition of G . Let t be an S -bottom node in \mathcal{T} that admits S -operation **join** $(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S)$ and is not contained in a full join tree. Let t_1, t_2 be the two S -children of t and let t_0 be the parent of t . If $X_t = \text{smallbag}(X_t^S, d)$, $X_{t_0} = \text{nbag}(X_t, d, P)$, $X_{t_1} = \text{nbag}(X_t, d, L)$, and $X_{t_2} = \text{nbag}(X_t, d, R)$, then we say that node t admits S -operation **smalljoin** $(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, d)$.



■ **Figure 10** Illustration of the small S -operations (Definitions 5.3 and 5.4). Here, the red circles are S -bottom nodes. The remaining circles are parents and S -children, respectively. Children of the S -bottom nodes that are not S -children are not depicted. Thick circles represent nodes whose bags may be full. The bags of nodes represented by thin circles are not full.

The small S -operations are illustrated in Figure 10. Note that in particular, if an S -bottom node admits the S -operation **smallintroduce**, then the bags of its parent and its S -child are not full. Similarly, if an S -bottom node admits the S -operation **smalljoin**, then the bags of its parent and both its S -children are not full.

In the case that an S -bottom node t that admits the S -operation **forget**(v), we have that its S -child t' has a potentially larger bag. In order to determine the bag of t , we will first compute the bag of the S -child t' . Informally speaking, we need some information about the child of t' that is on the path from t to its closest descendant that is an S -bottom node. Let t'' be the closest descendant of t that is an S -bottom node. If t'' has the same bag as t' , then we essentially have to compute the bag of the S -bottom node t'' . Hence, we need a flag $f \in \{\text{true}, \text{false}\}$ that indicates whether we are in this case. And if we are (that is, $f = \text{true}$), then we need all information about the S -operation τ of t'' . Note that if t'' also admits the S -operation **forget**(v), then we know that the bag of t'' is not full, which will be enough. If we are not in the above-described case (that is, $f = \text{false}$), then, similar as in the other small S -operations, we need an additional integer that will guide our algorithm to a specific bag candidate. To capture all the above-described extra information, we introduce the following extended version of the S -operation **forget**(v).

- **extendedforget**(v, d, f, τ) with some vertex $v \in S$, $d \in \{1, \dots, (n+1)^3\}$, $f \in \{\text{true}, \text{false}\}$, and some extended S -operation τ that is different from **extendedforget**.

Similarly to the previously introduced extended S -operations, we define some auxiliary functions that serve analogous purposes. Let

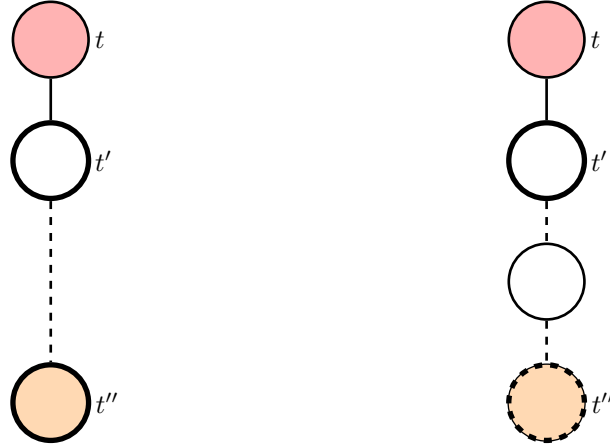
$$\text{smallbag}_{\text{forget}} : 2^S \times \mathbb{N} \times \{\text{true}, \text{false}\} \times \Pi \rightarrow 2^V,$$

where Π denotes the set of all extended S -operations. The function is formally defined in Section 6.2. Furthermore, let

$$\text{nbag}_{\text{forget}} : 2^V \times \mathbb{N} \times \{\text{true}, \text{false}\} \times \Pi \times \{P, L, R\} \rightarrow 2^V.$$

We give a formal definition of both functions to Section 6.

► **Definition 5.5** (**extendedforget**). Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a slim S -nice tree decomposition of G . Let t be an S -bottom node in \mathcal{T} that admits S -operation **forget**(v). Let t' be the S -child of t . Let τ be an extended S -operation.



(a) Node t admits $\text{extendedforget}(v, d, \text{true}, \tau)$. (b) Node t admits $\text{extendedforget}(v, d, \text{false}, \text{void})$.

■ **Figure 11** Illustration of the extended S -operation extendedforget (Definition 5.5). Here, the red and orange circles are S -bottom nodes. Children of the S -bottom nodes that are not S -children are not depicted. Thick and dashed circles represent nodes whose bags may be full. The bags of nodes represented by thin circles are not full. The dashed edges in Figures 11a and 11b indicate ancestor-descendant relationships. The node with a dashed circle in Figure 11b indicates that the bag of this node may be full, but it is different from the bag of the child of node t .

- If there is a node t'' that is the closest descendant of t which is an S -bottom node, $|X_t| = k$, $X_t \cup \{v\} = X_{t'} = X_{t''}$, t'' admits S -operation τ , $X_{t'} = \text{smallbag}_{\text{forget}}(X_t^S, d, \text{true}, \tau)$, $X_t = \text{nbag}_{\text{forget}}(X_{t'}, d, \text{true}, \tau, P)$, and $X_{t''} = \text{nbag}_{\text{forget}}(X_t, d, \text{true}, \tau, L)$, then we say that node t admits S -operation $\text{extendedforget}(v, d, \text{true}, \tau)$.
- Otherwise, if τ , $X_{t'} = \text{smallbag}_{\text{forget}}(X_t^S, d, \text{false}, \text{void})$, $X_t = \text{nbag}_{\text{forget}}(X_{t'}, d, \text{false}, \text{void}, P)$, and $X_{t''} = \text{nbag}_{\text{forget}}(X_t, d, \text{false}, \text{void}, L)$, then we say that node t admits S -operation $\text{extendedforget}(v, d, \text{false}, \text{void})$.

The extended S -operations extendedforget is illustrated in Figure 11. If the S -operation τ in $\text{extendedforget}(v, d, f, \tau)$ is a big S -operation, then we consider $\text{extendedforget}(v, d, f, \tau)$ a big S -operation. Otherwise, we consider $\text{extendedforget}(v, d, f, \tau)$ a small S -operation. We can observe that if an S -bottom node admits some extended S -operation $\text{extendedforget}(v, d, f, \tau)$, then $\tau \neq \text{extendedforget}(v', d', f', \tau')$. This follows from the fact that the bag of an S -bottom node that admits S -operation $\text{forget}(v)$ (and hence potentially an extendedforget S -operation) is never full.

► **Observation 5.6.** Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a slim S -nice tree decomposition of G . Let t be an S -bottom node in \mathcal{T} that admits S -operation $\text{extendedforget}(v, d, f, \tau)$ for some $v \in S$, $d \in \{1, \dots, (n+1)^3\}$, $f \in \{\text{true}, \text{false}\}$, and some S -operation τ . Then we have that $\tau \neq \text{extendedforget}(v', d', f', \tau')$ for every $v' \in S$, $d' \in \{1, \dots, (n+1)^3\}$, $f' \in \{\text{true}, \text{false}\}$, and every extended S -operation τ' .

Finally, we remark that for the small S -operations, we do not have an analog to Lemma 5.2, that is, in a slim S -nice tree decomposition, it is generally not the case that every S -bottom node that is not contained in a full join tree admits some small S -operation. We have to refine the tree decompositions further to ensure that all S -bottom nodes admit some extended (big or small) S -operation. This is discussed further in Section 9.

5.2 Dynamic Programming States and Table

Now we define a state of the dynamic program. From now on, we only consider extended S -operations, that is, whenever we refer to S -operations, we only refer to extended ones. As described at the beginning of the section, it contains an S -trace and two S -operations, one for the S -bottom node of the directed path of the S -trace, and one for the parent (if it exists) of the top node of the directed path of the S -trace. Formally, we define a state for the dynamic program as follows.

► **Definition 5.7** (State, Witness). *Let (L^S, X^S, R^S) be an S -trace and let τ_+ and τ_- be two S -operations. We call $\phi = (\tau_-, L^S, X^S, R^S, \tau_+)$ a state. We say that a slim S -nice tree decomposition \mathcal{T} of G witnesses ϕ if the following conditions hold.*

- *If $\tau_+ = \text{void}$, then the root of \mathcal{T} has S -trace (L^S, X^S, R^S) .*
- *If $L^S = \emptyset$, then $\tau_- = \text{void}$ and there exists an S -bottom node that admits S -operation τ_+ and has an S -child with S -trace (L^S, X^S, R^S) .*
- *If $L^S \neq \emptyset$, then the S -bottom node of the directed path of S -trace (L^S, X^S, R^S) admits S -operation τ_- and, if additionally $\tau_+ \neq \text{void}$, then the parent of the top node of the directed path of S -trace (L^S, X^S, R^S) admits S -operation τ_+ .*

We denote with \mathcal{S} the set of all states.

Note that Definition 5.7 implies that if a slim S -nice tree decomposition \mathcal{T} of G witnesses $\phi = (\tau_-, L^S, X^S, R^S, \tau_+)$, then we have that $\tau_- = \text{void}$ if and only if $L^S = \emptyset$, since S -bottom nodes never admit the S -operation void .

Intuitively, we wish to obtain a dynamic programming table $\text{PTW} : \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ (for “partial treewidth”) that maps states to true if and only if there is a tree decomposition \mathcal{T} of the subgraph “below” the top node of the directed path of the S -trace contained in the state that has width at most k and that witnesses ϕ . Note that this is not a precise definition, since there may be many vertices in the graph G that we may add to bags in the directed path of the S -trace contained in the state, or above, or below it. Informally speaking, we will only add vertices of G to bags in the directed path of the S -trace contained in the state if they *need* to be added to those bags. This allows us to compute a function $\text{LTW} : \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ (for “local treewidth”) that outputs true if and only if the size of any bag in the directed path is at most $k + 1$. We will give a precise definition later. Our definition for $\text{PTW} : \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ will be somewhat similar to the one by Chapelle et al. [33], however, our definition of $\text{LTW} : \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ will be significantly more sophisticated than the one used by Chapelle et al. [33].

Given a state $(\tau_-, L^S, X^S, R^S, \tau_+)$ of the dynamic program, by Observations 4.10–4.12 we can immediately deduce the S -traces of the S -children of the bottom node of the directed path of the S -trace (L^S, X^S, R^S) . This gives rise to a “preceding”-relation on states.

► **Definition 5.8** (Preceding States). *Let $\phi = (\tau_-, L^S, X^S, R^S, \tau_+)$ be a state with $L^S \neq \emptyset$. Denote with Π the set of all extended S -operations. Then, if $\tau_- = \text{smallintroduce}(v, d)$ or $\tau_- = \text{extendedforget}(v, d, f, \tau)$, the set $\Psi(\phi)$ of preceding states of ϕ is defined as follows.*

- *If $\tau_- = \text{smallintroduce}(v, d)$, then*

$$\Psi(\phi) = \{(\tau'_-, L^S, X^S \setminus \{v\}, R^S \cup \{v\}, \text{smallintroduce}(v, d)) \mid \tau'_- \in \Pi\}.$$

- *If $\tau_- = \text{extendedforget}(v, d, f, \tau)$, then*

$$\begin{aligned} \Psi(\phi) &= \{(\tau, L^S \setminus \{v\}, X^S \cup \{v\}, R^S, \text{extendedforget}(v, d, f, \tau))\} \text{ if } f = \text{true, and} \\ \Psi(\phi) &= \{(\tau'_-, L^S \setminus \{v\}, X^S \cup \{v\}, R^S, \text{extendedforget}(v, d, f, \tau)) \mid \tau'_- \in \Pi\} \text{ otherwise.} \end{aligned}$$

If $\tau_- = \text{smalljoin}(X^S, X_1^S, X_2^S, L_1^S, L_2^S, d)$ or $\tau_- = \text{bigjoin}(X^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)$, then the sets $\Psi_1(\phi)$ and $\Psi_2(\phi)$ of preceding states of ϕ are defined as follows.

■ If $\tau_- = \text{smalljoin}(X^S, X_1^S, X_2^S, L_1^S, L_2^S, d)$, then

$$\begin{aligned}\Psi_1(\phi) &= \{(\tau'_-, L_1^S, X_1^S, R^S \cup (X^S \setminus X_1^S) \cup L_2^S, \text{smalljoin}(X^S, X_1^S, X_2^S, L_1^S, L_2^S, d)) \\ &\quad \mid \tau'_- \in \Pi\} \text{ and} \\ \Psi_2(\phi) &= \{(\tau'_-, L_2^S, X_2^S, R^S \cup (X^S \setminus X_2^S) \cup L_1^S, \text{smalljoin}(X^S, X_1^S, X_2^S, L_1^S, L_2^S, d)) \\ &\quad \mid \tau'_- \in \Pi\}.\end{aligned}$$

■ If $\tau_- = \text{bigjoin}(X^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)$, then

$$\begin{aligned}\Psi_1(\phi) &= \{(\tau'_-, L_1^S, X_1^S, R^S \cup (X^S \setminus X_1^S) \cup L_2^S, \text{bigjoin}(X^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)) \\ &\quad \mid \tau'_- \in \Pi\} \text{ and} \\ \Psi_2(\phi) &= \{(\tau'_-, L_2^S, X_2^S, R^S \cup (X^S \setminus X_2^S) \cup L_1^S, \text{bigjoin}(X^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)) \\ &\quad \mid \tau'_- \in \Pi\}.\end{aligned}$$

Furthermore, by Corollary 4.13 we know that the “preceding”-relation on states is acyclic. Hence, we can look up the value of PTW for preceding states in our dynamic programming table when computing $\text{PTW}(\phi)$ for some state ϕ . However, Definition 5.8 is purely syntactic and not for every pair of a state and its preceding states there is a tree decomposition that witnesses all states at the same time. Hence, we need to check which preceding states are *legal*. To this end, we introduce functions $\text{legal}_1 : \mathcal{S} \times \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ and $\text{legal}_2 : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ that check whether for a pair of states ϕ, ψ or a triple of states ϕ, ψ_1, ψ_2 , respectively, the second state or the second and third state are legal predecessors for the first state, respectively. In other words, it checks whether there exists a tree decomposition that witnesses all states and their preceding-relation. We give a formal definition of this function in Section 8. For the ease of presentation, we state the dynamic program as an algorithm that only decides whether the input graph has treewidth at most k or not. However, the algorithm can easily be adapted to compute and output a corresponding tree decomposition. We comment more on this in Sections 7 and 9. Formally, we define the dynamic programming table as follows.

► **Definition 5.9** (Dynamic Programming Table PTW). *The dynamic programming table is a recursive function $\text{PTW} : \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ which is defined as follows. Let $\text{LTW} : \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$, let $\text{legal}_1 : \mathcal{S} \times \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$, and let $\text{legal}_2 : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$. Let $\phi = (\tau_-, L^S, X^S, R^S, \tau_+) \in \mathcal{S}$ be a state, then make the following case distinction based on τ_- .*

■ If $\tau_- = \text{void}$, then

$$\text{PTW}(\phi) = \text{LTW}(\phi).$$

■ If $\tau_- = \text{smallintroduce}(v, d)$ or $\tau_- = \text{extendedforget}(v, d, f, \tau)$, then

$$\text{PTW}(\phi) = \bigvee_{\psi \in \Psi(\phi)} (\text{LTW}(\phi) \wedge \text{PTW}(\psi) \wedge \text{legal}_1(\phi, \psi)).$$

■ If $\tau_- = \text{smalljoin}(X^S, X_1^S, X_2^S, L_1^S, L_2^S, d)$ or $\tau_- = \text{bigjoin}(X^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)$, then

$$\text{PTW}(\phi) = \bigvee_{\psi_1 \in \Psi_1(\phi) \wedge \psi_2 \in \Psi_2(\phi)} (\text{LTW}(\phi) \wedge \text{PTW}(\psi_1) \wedge \text{PTW}(\psi_2) \wedge \text{legal}_2(\phi, \psi_1, \psi_2)).$$

5.3 Local Treewidth

The function LTW depends on the following three sets of vertices. Let $\phi = (\tau_-, L^S, X^S, R^S, \tau_+)$ be a state.

- X_{\max}^ϕ are the vertices in V that we want to be in the bag of the top node t_{\max} of the directed path of (L^S, X^S, R^S) , or, if $L^S = \emptyset$ in the bag of the S -child with S -trace (L^S, X^S, R^S) of an S -bottom node.
- X_p^ϕ are the vertices in V that we want to be in the bag of the parent of t_{\max} . If t_{\max} is the root, we will assume that $X_p^\phi = \emptyset$.
- X_{\min}^ϕ are the vertices in V that we want to be in the bag of the bottom node t_{\min} of the directed path of (L^S, X^S, R^S) , or if $L^S = \emptyset$ then we set $X_{\min} = X^S$.
- X_{path}^ϕ are additional vertices in $V \setminus S$ that we want to place in some bag of the directed path of (L^S, X^S, R^S) , or in some bag of a subtree rooted in a child of a node in the directed path that is not an S -child, or if $L^S = \emptyset$ in some bag of a node below t_{\max} that is not an S -child.

As an example, if a vertex v has neighbors both in L^S and R^S , then there are bags in the subtree below t_{\min} where v already met its neighbors in L^S and there must still be bags above t_{\max} where v meets its neighbors in R^S . Hence, to meet Condition 3 of Definition 3.1, vertex v needs to be in the bags of both t_{\max} and t_{\min} and also every bag between them. The set X_{path}^ϕ , intuitively, contains all vertices of connected components of $G - S$ such that all their neighbors are in bags between the ones of t_{\max} and t_{\min} , and it is not the case that all neighbors are in bags above t_{\max} .

In order to define X_{path}^ϕ , we also need to know which vertices are contained in the bags of the S -children of t_{\min} (if there are any). Let X_c^ϕ be the union of the bags of the S -children of t_{\min} and the empty set if $L^S = \emptyset$. Computing X_{\max}^ϕ , X_p^ϕ , X_{\min}^ϕ , and X_c^ϕ for a given state ϕ is the main challenge here and we will dedicate the next section to this. We give a formal definition of these three sets and algorithms to compute them in Section 6. Given X_{\max}^ϕ , X_p^ϕ , and X_{\min}^ϕ , we define X_{path}^ϕ as follows.

$$v \in X_{\text{path}}^\phi \Leftrightarrow \text{there exists an } F\text{-component } C \text{ in } G - (X_{\max}^\phi \cup X_p^\phi \cup X_{\min}^\phi) \text{ such that} \\ v \in V(C), N[V(C)] \cap (X_c^\phi \setminus X_{\min}^\phi) = \emptyset, \text{ and } N(V(C)) \setminus X_p^\phi \neq \emptyset.$$

Using these four sets, we are ready to define the function $\text{LTW} : \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$.

► **Definition 5.10 (Local Treewidth LTW).** *Let $\phi = (\tau_-, L^S, X^S, R^S, \tau_+)$ be a state, and let G' be the graph obtained from G by adding edges between all pairs of vertices $u, v \in X_{\max}^\phi$ with $\{u, v\} \notin E$, all pairs of vertices $u, v \in X_p^\phi$ with $\{u, v\} \notin E$, and all pairs of vertices $u, v \in X_{\min}^\phi$ with $\{u, v\} \notin E$. Then*

$$\text{LTW}(\phi) = \text{true} \Leftrightarrow \text{tw}(G'[X_{\max}^\phi \cup X_p^\phi \cup X_{\min}^\phi \cup X_{\text{path}}^\phi]) \leq k.$$

6 Computing the Top and Bottom Bag of a Directed Path

For this section, assume we are given a state $\phi = (\tau_-, L^S, X^S, R^S, \tau_+)$ in addition to the graph $G = (V, E)$, integer k , and set $S \subseteq V$. Throughout the section, let t_{\max} the top node of the directed path of (L^S, X^S, R^S) and let t_{\min} the bottom node of the directed path of (L^S, X^S, R^S) . We give algorithms to compute the following.

- A candidate for the bag X_{\max}^ϕ of t_{\max} .
- A candidate for the bag X_p^ϕ of the parent of t_{\max} except for the case where $\tau_+ = \text{void}$ (in this case t_{\max} is the root of the tree decomposition).

- A candidate for the bag X_{\min}^ϕ of t_{\min} .
- A candidate for the union of the bags X_c^ϕ of the S -children of t_{\min} except for the case where $\tau_- = \text{void}$ (in this case t_{\min} does not have any S -children).
- A candidate for the set X_{path}^ϕ of vertices that we want to add to bags of the directed path or subtrees of the tree decomposition that are rooted at children of nodes in the directed path.

We first discuss the case where we want to compute X_{\max}^ϕ . We postpone the discussion for the remaining cases to Section 6.3. We will show how to compute a set X that we can use in the following way. In the case that we want to compute X_{\max}^ϕ , the set X will either be X_{\max}^ϕ , or the bag of the parent of t_{\max} (if it exists). Informally speaking, X will be the bag of the parent, if the parent (potentially) has a larger bag. This can happen if $\tau_+ = \text{smallintroduce}(v, d)$, $\tau_+ = \text{smalljoin}(X^S, X_1^S, X_2^S, L_1^S, L_2^S, d)$, or $\tau_+ = \text{bigjoin}(X^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)$. In the case of $\tau_+ = \text{extendedforget}(v, d, f, \tau)$, the set X will be the bag of t_{\max} . The set X that we compute is the bag of the nodes visualized with a thick circle in the illustrations in Figures 9–11. In the case where $\tau_+ = \text{void}$, we will show that we can set $X = X_{\max}^\phi = X^S$. We will give two algorithms, one for the case that $\tau_+ = \text{bigjoin}(X^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)$ and one for all other cases.

Note that by Observation 4.6 we have that the parent of t_{\max} is an S -bottom node. It is the node that admits the extended S -operation τ_+ . Let that node be called t^* . Observe that in all cases except $\tau_+ = \text{void}$, the information in the extended S -operations allows us to determine from X (and hence also X_{\max}^ϕ) the bag of t^* , the bag of the parent of t^* , and the bags of the S -children of t^* .

A crucial property of our algorithms is that the computation of X is independent of the S -operation τ_- (except for one special case that we will discuss later). This allows us to do the following: Instead of computing X_{\min}^ϕ , we compute the set X_{\max}^ψ for some $\psi \in \Psi(\phi)$ or some $\psi \in \Psi_1(\phi)$ if τ_- is a **smalljoin** or a **bigjoin** S -operation (it will become clear that picking some $\psi \in \Psi_2(\phi)$ will yield the same result).

In order to prove that our algorithms are correct, we will exploit the additional properties of slim top-heavy S -nice tree decompositions (in comparison to “normal” S -nice tree decompositions). In Section 6.1, we present the algorithm for the case where τ_+ is S -operations **bigjoin**. Afterwards, we present the algorithm for the case where τ_+ is different from S -operation **bigjoin** in Section 6.2. These algorithms implicitly define the functions **bigbag**, **smallbag**, **nbag**, **smallbag_{forget}**, and **nbag_{forget}** used in Section 5.1.

6.1 Big S -Operations

To make notation more convenient, assume that the input state is $\phi = (\tau_-, \hat{L}^S, \hat{X}^S, \hat{R}^S, \tau_+)$. Our goal is to compute the bag X_{\max}^ϕ for states ϕ where τ_+ is a **bigjoin** S -operation or $\tau_+ = \text{extendedforget}(v, d, \text{true}, \tau)$ and τ is a **bigjoin** S -operation. In this case we say that τ_+ is a big S -operation.

Setup. As described at the beginning of Section 6, we compute a set X of vertices that is a candidate for a *big target node* (or just *target node*) t^* , which is not necessarily the same node as t_{\max} . Let \mathcal{T} be a slim S -nice tree decomposition of G that contains a directed path for S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, or if $\hat{L}^S = \emptyset$ (and $\tau_- = \text{void}$) let \mathcal{T} be a slim S -nice tree decomposition of G that contains an S -bottom node that has an S -child (which we also call t_{\max}) with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$. The *big target node* t^* is defined as follows.

- If $\tau_+ = \text{bigjoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)$, then t^* is the parent of t_{\max} .

- If $\tau_+ = \text{extendedforget}(v, d, \text{true}, \tau)$ and $\tau = \text{bigjoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)$, then $t^* = t_{\max}$.

It follows that the target node t^* is always an S -bottom node that admits some S -operation $\text{bigjoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)$. Let from now on s, d_0, d_1, d_2 be the bit string and the three integers, respectively, that are contained in the S -operation of the target node. We describe an algorithm to compute the function $\text{bigbag}(X_{t^*}^S, s)$ and use it to compute X , that is $X = \text{bigbag}(X_{t^*}^S, s)$. To this end, we need to know the set $X_{t^*}^S$ from the S -trace of t^* . To make notation simpler, we drop the subscript t^* and let (L^S, X^S, R^S) be the S -trace of the target node t^* .

- Let $\tau_+ = \text{bigjoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)$. Then by Definition 5.1 and Observation 4.12 we either have that
 - $\hat{L}^S = L_1^S$, then $(L^S, X^S, R^S) = (\hat{L}^S \cup L_2^S, X_t^S, \hat{R}^S \setminus (L_2^S \cup X_t^S))$, or we have that
 - $\hat{L}^S = L_2^S$, then $(L^S, X^S, R^S) = (\hat{L}^S \cup L_1^S, X_t^S, \hat{R}^S \setminus (L_1^S \cup X_t^S))$.
- Let $\tau_+ = \text{extendedforget}(v, d, \text{true}, \tau)$ and $\tau = \text{bigjoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)$. Then $(L^S, X^S, R^S) = (\hat{L}^S, \hat{X}^S, \hat{R}^S)$.

Algorithm. Now, we describe the algorithm to compute $\text{bigbag}(X^S, s)$. The algorithm will maintain a 3-partition (**Rest**, **In**, **Out**) of the vertex set V , where, intuitively,

- **In** contains vertices that we want to be in X ,
- **Out** contains vertices that we do not want to be in X , and
- **Rest** contains vertices where we have not decided yet whether we want them to be in X or not.

Algorithm 1 (bigbag).

Input: A set $X^S \subseteq S$, and a bit string s .

Output: A set $X \subseteq V$.

Set **In** = X^S , **Out** = $S \setminus X^S$, and **Rest** = $V \setminus S$. Let i be an integer variable. Set i to one. Perform the first applicable of the following steps until **Rest** is empty or $|\text{In}| = k + 1$. Break all ties in an arbitrary but fixed way.

1. If there is a vertex v in **Rest** that has degree at most one in $G[\text{Rest}]$, such that at least two S -components in $G[\text{Out}]$ contain some neighbor of v , then do the following.
 - If the i th bit of s is one, then remove v from **Rest** and add v to **In**. Increase i by one.
 - Otherwise, remove v from **Rest** and add v to **Out**. Increase i by one.
2. If there is a vertex v in **Rest** that has degree at most one in $G[\text{Rest}]$, then remove v from **Rest** and add v to **Out**.

Output **In**.

Since $G[\text{Rest}]$ is initially a forest and we never add new vertices to **Rest**, we have that as long as **Rest** is non-empty, at least one of the above two steps of Algorithm 1 is applicable. We set $X = \text{bigbag}(X^S, s)$. We can observe that the computation takes polynomial time.

► **Observation 6.1.** *Algorithm 1 runs in polynomial time.*

Correctness. Now we show that Algorithm 1 is correct. To this end, we give a proof for Lemma 5.2. We first make the following observation.

► **Lemma 6.2.** *During every iteration of Algorithm 1, the following holds. For every connected component C in $G[\text{Out} \setminus S]$ we have that $|N(V(C)) \cap \text{Rest}| \leq 1$.*

Proof. TOPROVE 10 ◀

Now we show the following result, which directly implies Lemma 5.2.

► **Lemma 6.3.** *Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a slim S -nice tree decomposition of G . Let T' be a full join tree in \mathcal{T} where all nodes in T' have bag X . Then there exist a bit string $s \in \{0, 1\}^{2\text{fvn}(G)+1}$ such that $\text{bigbag}(X \cap S, s) = X$.*

Proof. TOPROVE 11 ◀

It is straightforward to see that Lemma 6.3 directly implies Lemma 5.2.

6.2 Small S -Operations

As in the previous section, to make notation more convenient, assume that the input state is $\phi = (\tau_-, \hat{L}^S, \hat{X}^S, \hat{R}^S, \tau_+)$. Our goal is to compute the bag X_{\max}^ϕ for states ϕ where τ_+ is not a big S -operation, where $\tau_+ = \text{extendedforget}(v, d, f, \tau)$ with $f = \text{false}$ (by Definition 5.5, we then also always have that $\tau = \text{void}$), and where $\tau_+ = \text{extendedforget}(v, d, f, \tau)$ with $f = \text{true}$ and τ is not a big S -operation. In this case we say that τ_+ is a small S -operation.

Setup. As described at the beginning of Section 6, we compute a set X of vertices that is a candidate for a *small target node* (or just *target node*) t^* , which is not necessarily the same node as t_{\max} . Formally, the target node is defined as follows. Let \mathcal{T} be a slim S -nice tree decomposition of G that contains a directed path for S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, or if $\hat{L}^S = \emptyset$ (and $\tau_- = \text{void}$) let \mathcal{T} be a slim S -nice tree decomposition of G that contains an S -bottom node that has an S -child (which we also call t_{\max}) with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$. We assume that neither t_{\max} nor the parent of t_{\max} are part of a full join tree, as in this case is handled in Section 6.1. The *small target node* t^* is defined as follows.

- If $\tau_+ = \text{smallintroduce}(v, d)$ or $\tau_+ = \text{smalljoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, d)$, then t^* is the parent of t_{\max} .
- If $\tau_+ = \text{extendedforget}(v, d, f, \tau)$ and τ is not a **bigjoin** S -operation, then $t^* = t_{\max}$.

If $\tau_- = \text{void}$, then t_{\min} is not defined. Otherwise, Definition 4.19 (the definition of slim) implies that either $t_{\min} = t_{\max} = t^*$, this happens if $\tau_+ = \text{extendedforget}(v, d, f, \tau)$ and bag of t_{\max} is full, or we have that t_{\max} and t_{\min} are different, and t_{\max} is not the parent of t_{\min} . In the former case, we have also that $X_{\max}^\phi = X_{\min}^\phi$ hence, we can use the computation of X_{\min}^ϕ to obtain X_{\max}^ϕ . This is explained in Section 6.3 and uses information in τ_- . This is also the case mentioned in the beginning of the section where the computation of X_{\max}^ϕ depends on τ_- . From now on, we assume that we are not in this case, that is, we assume that if t_{\min} is defined, then t_{\max} and t_{\min} are different and t_{\max} is not the parent of t_{\min} .

If t_{\min} is defined and $\tau_+ \neq \text{smalljoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, d)$, then we denote with $t_{d_1}^*$ the closest descendant of t^* that is parent of an S -bottom node. If t_{\min} is defined and $\tau_+ = \text{smalljoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, d)$, then we denote with $t_{d_1}^*$ and $t_{d_2}^*$ the closest descendants of the two S -children of t^* , respectively, that are parent of an S -bottom node. Furthermore, we denote by t_a^* the closest ancestor of t^* that is parent of an S -bottom node. Note that we have that $t_{d_1}^*$ or $t_{d_2}^*$ is the parent of t_{\min} , or, in other words, that the parent of t_{\min} is not t^* . Furthermore, note that in the case of $\tau_+ = \text{smallintroduce}(v, d)$ and $\tau_+ = \text{smalljoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, d)$ we have that t^* is an S -bottom node and hence t_a^* is the parent of t^* .

To make notation simpler, we drop the subscript t^* and let (L^S, X^S, R^S) be the S -trace of the target node t^* . We get the following using Observations 4.10–4.12.

- Let $\tau_+ = \text{smallintroduce}(v, d)$. Then $(L^S, X^S, R^S) = (\hat{L}^S, \hat{X}^S \cup \{v\}, \hat{R}^S \setminus \{v\})$.
- Let $\tau_+ = \text{smalljoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, d)$. Then by Definition 5.4 and Observation 4.12 we either have that
 - $\hat{L}^S = L_1^S$, then $(L^S, X^S, R^S) = (\hat{L}^S \cup L_2^S, X_t^S, \hat{R}^S \setminus (L_2^S \cup X_t^S))$, or we have that
 - $\hat{L}^S = L_2^S$, then $(L^S, X^S, R^S) = (\hat{L}^S \cup L_1^S, X_t^S, \hat{R}^S \setminus (L_1^S \cup X_t^S))$.
- If $\text{extendedforget}(v, d, f, \tau)$ and τ is not a **bigjoin** S -operation, then $(L^S, X^S, R^S) = (\hat{L}^S, \hat{X}^S, \hat{R}^S)$.

Furthermore, we define sets L_1^S and L_2^S as follows.

- If $\tau_+ = \text{smalljoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, d)$, then L_1^S and L_2^S are the respective sets in τ_+ .
- If $\tau_+ = \text{extendedforget}(v, d, f, \tau)$ with S -operation $\tau = \text{smalljoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, d)$, then L_1^S and L_2^S are the respective sets in τ .
- In all other cases, we set $L_1^S = L^S$ and $L_2^S = \emptyset$.

Finally, we can observe the following useful properties of target nodes.

► **Observation 6.4.** *Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a slim S -nice tree decomposition of G with width k that contains a directed path of length at least three with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, or if $\hat{L}^S = \emptyset$ then \mathcal{T} contains an S -bottom node that has an S -child with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, such that neither t_{\max} nor the parent of t_{\max} are contained in a full join tree. Let t^* denote the target node in \mathcal{T} .*

1. Node t^* has S -trace (L^S, X^S, R^S) .
2. The bag X_{t_p} of the parent t_p of t^* is not full and $X_{t_p} = X_{t_a} \subseteq X_{t^*}$.
3. If t^* has exactly one S -child t_1 , then the bag X_{t_1} of t_1 is not full and $X_{t_1} \subseteq X_{t^*}$.
4. If t^* has exactly two S -children t_1, t_2 , then the bag X_{t_1} of t_1 is not full, the bag X_{t_2} of t_2 is not full, $X_{t_1} \subseteq X_{t^*}$, and $X_{t_2} \subseteq X_{t^*}$.

Proof. TOPROVE 12 ◀

Algorithm. Now, we describe the algorithm to compute X . The algorithm, informally speaking, works as follows. It uses the additional integer d from the extended S -operation to make an initial “guess” of up to three vertices which are contained in the bag X of t^* , but not in the bag of the parent or the S -children, respectively. By Observation 6.4 we know that if the bag X is full, then those vertices must exist. Using this guess, we compute a candidate for bag X in a greedy fashion. The guess also lets us determine candidates for the bags of the parent of t^* and its S -children. We can prove that there is a tree decomposition for G with width k using these candidates which is S -nice and slim, but this tree decomposition is not necessarily top-heavy. However, top-heaviness is crucial for proving correctness of the algorithm. To this end, the algorithm modifies the candidate sets such that there is a tree decomposition for G with width k using the modified sets which is S -nice, slim, and top-heavy for t_a^* .

We use the integer d to obtain the initial guess as follows. Let $\mathcal{G} = (V \cup \{\perp\})^3$ be the set of all triples of vertices together with a special symbol \perp . Assume that the triples in \mathcal{G} are ordered in some arbitrary but fixed way. Note that there are $(n+1)^3$ different triples. Let (g_1, g_2, g_3) be the d th triple in \mathcal{G} according to the ordering. We define the sets D_R, D_{L_1}, D_{L_2} as follows.

- If $g_1 \neq \perp$, then we set $D_R = \{g_1\}$. Otherwise, we set $D_R = \emptyset$.
- If $g_2 \neq \perp$, then we set $D_{L_1} = \{g_2\}$. Otherwise, we set $D_{L_1} = \emptyset$.
- If $g_3 \neq \perp$, then we set $D_{L_2} = \{g_3\}$. Otherwise, we set $D_{L_2} = \emptyset$.

Algorithm 2 (smallbag).**Input:** Sets $R^S, X^S, L_1^S, L_2^S \subseteq S$, and $D_R, D_{L_1}, D_{L_2} \subseteq V$.**Output:** Sets $X, X_R, X_{L_1}, X_{L_2} \subseteq V$.

Set $X = \text{smallbagcandidate}(R^S, X^S, L_1^S, L_2^S, D_R, D_{L_1}, D_{L_2})$. Set $X_R = X \setminus D_R$, $X_{L_1} = X \setminus D_{L_1}$, and $X_{L_2} = X \setminus D_{L_2}$. Perform the first applicable step until no changes occur:

1. If there is an F -component C in $G - X_R$, then remove all vertices in $V(C)$ from the set X, X_R, X_{L_1} , and X_{L_2} .
2. If there is $v \in X_R \setminus S$ such that for every $u \in N(v)$ it holds that $u \in X_R$ or u is not connected to $L_1^S \cup L_2^S \cup ((X \setminus X_R) \cap S)$ in $G - X_R$, then remove v from X, X_R, X_{L_1} , and X_{L_2} .
3. If there is $v \in X_R \setminus S$ and $u \in N(v) \setminus (S \cup X_R)$ such that for every $u' \in N(v) \setminus \{u\}$ it holds that $u' \in X_R$ or u' is not connected to $L_1^S \cup L_2^S \cup ((X \setminus X_R) \cap S)$ in $G - X_R$, and u is connected to $L_1^S \cup L_2^S \cup ((X \setminus X_R) \cap S)$ in $G - X_R$, then replace v with u in all sets X, X_R, X_{L_1} , and X_{L_2} .

Output X, X_R, X_{L_1} , and X_{L_2} .

Note that we slightly overload notation here in comparison to the definition of the function **smallbag** in Section 5.1. Now we give a description of the subroutine **smallbagcandidate** called in the beginning of Algorithm 2. Similar to Algorithm 1, the subroutine will maintain a 3-partition (Rest, In, Out) of the vertex set V , where, intuitively,

- In contains vertices that we want to be in X ,
- Out contains vertices that we do not want to be in X , and
- Rest contains vertices where we have not decided yet whether we want them to be in X or not.

Furthermore, using the sets L_1^S, L_2^S , and R^S , we introduce the following terminology. Let C be a connected component in $G[V']$ for some $V' \subseteq V \setminus S$.

- If we have $N(V(C)) \cap L_1^S = \emptyset$, $N(V(C)) \cap L_2^S = \emptyset$, and $N(V(C)) \cap R^S = \emptyset$, then we call C an F -component.

Note that this is not equivalent to the definition of F -component in Section 4.4, but for convenience of presentation, we overwrite the terminology for this section. All occurrences of “ F -component” from now on in this section refer to the above definition.

- If we have $N(V(C)) \cap L_1^S \neq \emptyset$, $N(V(C)) \cap L_2^S = \emptyset$, and $N(V(C)) \cap R^S = \emptyset$, then we call C an L_1 -component.
- If we have $N(V(C)) \cap L_1^S = \emptyset$, $N(V(C)) \cap L_2^S \neq \emptyset$, and $N(V(C)) \cap R^S = \emptyset$, then we call C an L_2 -component.
- If we have $N(V(C)) \cap L_1^S = \emptyset$, $N(V(C)) \cap L_2^S = \emptyset$, and $N(V(C)) \cap R^S \neq \emptyset$, then we call C an R -component.
- If none of the above applies, then we call C a *mixed component*.

It is easy to see that every connected component of $G[V']$ falls into one of the above categories, for every choice of $V' \subseteq V \setminus S$.

Algorithm 3 (smallbagcandidate).**Input:** Sets $R^S, X^S, L_1^S, L_2^S \subseteq S$, and $D_R, D_{L_1}, D_{L_2} \subseteq V$.**Output:** A set $X \subseteq V$.Set $\text{In} = X^S \cup D_R \cup D_{L_1} \cup D_{L_2}$, $\text{Out} = S \setminus \text{In}$, and $\text{Rest} = V \setminus (\text{In} \cup \text{Out})$.Perform the first applicable of the following steps until Rest is empty. Break all ties in an arbitrary but fixed way.

1. If there is a vertex v in Rest that is contained in a mixed component in $G[(\text{Out} \cup \{v\}) \setminus S]$, then remove v from Rest and add v to In .
2. If there is a vertex v in Rest that is contained in an H -component C_H in $G[(\text{Out} \cup \{v\}) \setminus S]$ for $H \in \{L_1, L_2, R\}$ and $N(C_H) \cap D_H \neq \emptyset$, then remove v from Rest and add v to In .
3. If there is a vertex v in Rest that is contained in an R -component in $G[(\text{Out} \cup \{v\}) \setminus S]$ and that has degree at most one in $G[\text{Rest}]$, then remove v from Rest and add v to Out .
4. If there is a vertex v in Rest that is contained in an H -component in $G[(\text{Out} \cup \{v\}) \setminus S]$ for $H \in \{L_1, L_2\}$ and that has degree at most one in $G[\text{Rest}]$, then remove v from Rest and add v to Out .
5. If there is a vertex v in Rest that has degree one in $G[\text{Rest}]$ and the connected component C of $G[(\text{Out} \cup \{v\}) \setminus S]$ that contains v has the property that $N(V(C)) \cap \text{In} \neq \text{In}$, then remove v from Rest and add v to Out .
6. If there is a vertex v in Rest that has degree at most one in $G[\text{Rest}]$, then remove v from Rest and add v to Out .

Output In .

Since $G[\text{Rest}]$ is initially a forest and we never add new vertices to Rest , we have that as long as Rest is non-empty, at least one of the above four steps of Algorithm 3 is applicable. We can observe that the computation takes polynomial time.

► **Observation 6.5.** *Algorithm 2 runs in polynomial time.*

Correctness. Now we show that Algorithm 2 is correct. Assume there exists a slim S -nice tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G such that the following holds.

- \mathcal{T} contains a directed path of length at least three with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, or if $\hat{L}^S = \emptyset$ then \mathcal{T} contains an S -bottom node that has an S -child (called t_{\max}) with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, such that neither t_{\max} nor the parent of t_{\max} are contained in a full join tree.
- $X^S \cup D_R \cup D_{L_1} \cup D_{L_2} \subseteq X_{t^*}$ and $X_{t^*} \cap (S \setminus X^S) = \emptyset$,
- if t^* has a parent t_p , then $X_{t_p} = X_{t^*} \setminus D_R$,
- if t^* has one S -child t_1 , then $X_{t_1} = X_{t^*} \setminus D_{L_1}$, and
- if t^* has two S -children t_1 and t_2 , then $X_{t_1} = X_{t^*} \setminus D_{L_1}$ and $X_{t_2} = X_{t^*} \setminus D_{L_2}$.

Then we say that the top operation of ϕ contains the correct guess for \mathcal{T} . Now we show the following.

► **Proposition 6.6.** *Assume there exists a slim S -nice tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G with width k that contains a directed path of length at least three with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, or if $\hat{L}^S = \emptyset$ then \mathcal{T} contains an S -bottom node that has an S -child with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, such that neither t_{\max} nor the parent of t_{\max} are contained in a full join tree, and with target node t^* such that the following holds.*

- $|X_{t^*}|$ is minimal, that is, for all siblings \mathcal{T}' of \mathcal{T} such that all of the above holds and if \mathcal{T} is top-heavy for a node \hat{t} , then the subtrees rooted at \hat{t} in \mathcal{T} and \mathcal{T}' are the same, we have that $|X_{t^*}| \leq |X'_{t^*}|$,
- if $t_{d_1}^*$ and $t_{d_2}^*$, respectively, exist, then \mathcal{T} is top-heavy for $t_{d_1}^*$ and $t_{d_2}^*$, respectively,
- $X^S \cup D_R \cup D_{L_1} \cup D_{L_2} \subseteq X_{t^*}$ and $X_{t^*} \cap (S \setminus X^S) = \emptyset$, and
- ϕ contains the correct guess for \mathcal{T} .

Let $(X, X_R, X_{L_1}, X_{L_2})$ denote the output of $\text{smallbag}(R^S, X^S, L_1^S, L_2^S, D_R^\phi, D_{L_1}^\phi, D_{L_2}^\phi)$. Then there is a slim S -nice tree decomposition $\mathcal{T}' = (T', \{X'_t\}_{t \in V(T')})$ of G with width at most k such that the following holds.

- \mathcal{T}' is a sibling of \mathcal{T} ,
- if \mathcal{T} is top-heavy for a node \hat{t} such that \hat{t} is
 - not in $T_{t_a^*}$ and not an ancestor of t_a^* , or
 - a descendant of t^* ,
 then \mathcal{T}' is also top-heavy for \hat{t} and the subtree rooted at \hat{t} is the same as in \mathcal{T} ,
- \mathcal{T}' is top-heavy for t_a^* ,
- $X'_{t^*} = X$ and $X'_{t_a^*} = X_R$,
- if t^* has one S -child t_1 , then $X'_{t_1} = X_{L_1}$, and
- if t^* has two S -children t_1 and t_2 , then $X'_{t_1} = X_{L_1}$ and $X'_{t_2} = X_{L_2}$.

To show Proposition 6.6, in particular, we need to argue that the subrouting smallbagcandidate called in the beginning of Algorithm 2, that is, Algorithm 3 is correct. To this end, we have the following.

► **Lemma 6.7.** Assume there exists a slim S -nice tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G with width k that contains a directed path of length at least three with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, or if $\hat{L}^S = \emptyset$ then \mathcal{T} contains an S -bottom node that has an S -child with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, such that neither t_{\max} nor the parent of t_{\max} are contained in a full join tree, and with target node t^* such that the following holds.

- $X^S \cup D_R \cup D_{L_1} \cup D_{L_2} \subseteq X_{t^*}$ and $X_{t^*} \cap (S \setminus X^S) = \emptyset$,
 - if t^* has a parent t_p , then $X_{t_p} = X_{t^*} \setminus D_R$,
 - if t^* has one S -child t_1 , then $X_{t_1} = X_{t^*} \setminus D_{L_1}$, and
 - if t^* has two S -children t_1 and t_2 , then $X_{t_1} = X_{t^*} \setminus D_{L_1}$ and $X_{t_2} = X_{t^*} \setminus D_{L_2}$.
- Then there is a slim S -nice tree decomposition $\mathcal{T}' = (T', \{X'_t\}_{t \in V(T')})$ of G with width at most k such that the following holds.

- \mathcal{T}' is a sibling of \mathcal{T} ,
- if \mathcal{T} is top-heavy for a node \hat{t} such that \hat{t} is
 - not in $T_{t_a^*}$ and not an ancestor of t_a^* , or
 - a descendant of t^* ,
 then \mathcal{T}' is also top-heavy for \hat{t} and the subtree rooted at \hat{t} is the same as in \mathcal{T} ,
- $X'_{t^*} = \text{smallbagcandidate}(R^S, X^S, L_1^S, L_2^S, D_R, D_{L_1}, D_{L_2})$, and
- $|X'_{t^*}| < |X_{t^*}|$ or each of the following holds:
 - if t^* has a parent t'_p , then $X'_{t'_p} = X'_{t^*} \setminus D_R$,
 - if t^* has one S -child t'_1 , then $X'_{t'_1} = X'_{t^*} \setminus D_{L_1}$, and
 - if t^* has two S -children t'_1 and t'_2 , then $X'_{t'_1} = X'_{t^*} \setminus D_{L_1}$ and $X'_{t'_2} = X'_{t^*} \setminus D_{L_2}$.

We postpone the proof of Lemma 6.7 to the end of this subsection and first show how we can use it to prove Proposition 6.6.

Proof. TOPROVE 13

◀

Now we prove Lemma 6.7. To do this, we will essentially show that in each step of Algorithm 3, we do not make mistakes. This first lemma, intuitively, shows that Step 1 of Algorithm 3, if applicable, is always correct.

► **Lemma 6.8.** *Let $(\text{Rest}, \text{In}, \text{Out})$ be a 3-partition of V that appears during some iteration of Algorithm 3. If there exists a slim S -nice tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G with width k that contains a directed path of length at least three with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, or if $\hat{L}^S = \emptyset$ then \mathcal{T} contains an S -bottom node that has an S -child with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, such that neither t_{\max} nor the parent of t_{\max} are contained in a full join tree, and with target node t^* such that $\text{In} \subseteq X_{t^*}$ and $X_{t^*} \cap \text{Out} = \emptyset$, then the following holds. If there is a vertex $v \in \text{Rest}$ that is contained in a mixed component in $G[(\text{Out} \cup \{v\}) \setminus S]$, then $v \in X_{t^*}$.*

Proof. TOPROVE 14 ◀

Next, we show that the Step 2 of Algorithm 3, if applicable, is correct.

► **Lemma 6.9.** *Let $(\text{Rest}, \text{In}, \text{Out})$ be a 3-partition of V that appears during some iteration of Algorithm 3. Assume that there exists a slim S -nice tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G with width k that contains a directed path of length at least three with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, or if $\hat{L}^S = \emptyset$ then \mathcal{T} contains an S -bottom node that has an S -child with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, such that neither t_{\max} nor the parent of t_{\max} are contained in a full join tree, and with target node t^* such that the following holds.*

- $\text{In} \subseteq X_{t^*}$ and $X_{t^*} \cap \text{Out} = \emptyset$,
- if t^* has a parent t_p , then $X_{t_p} = X_{t^*} \setminus D_R$,
- if t^* has one S -child t_1 , then $X_{t_1} = X_{t^*} \setminus D_{L_1}$, and
- if t^* has two S -children t_1 and t_2 , then $X_{t_1} = X_{t^*} \setminus D_{L_1}$ and $X_{t_2} = X_{t^*} \setminus D_{L_2}$.

Then, if there is a vertex v in Rest that is contained in an H -component C_H in $G[(\text{Out} \cup \{v\}) \setminus S]$ for $H \in \{L_1, L_2, R\}$ and $N(C_H) \cap D_H \neq \emptyset$, we have that $v \in X_{t^}$.*

Proof. TOPROVE 15 ◀

Before we show that the remaining steps of the algorithm are correct, we first make the following observation. We have made essentially the same observation for Algorithm 1 in Lemma 6.2. The proof here is very similar, but since Algorithm 3 obviously behaves differently from Algorithm 1, some small adjustments need to be made.

► **Lemma 6.10.** *During every iteration of Algorithm 3, the following holds. For every connected component C in $G[\text{Out} \setminus S]$ we have that $|N(V(C)) \cap \text{Rest}| \leq 1$.*

Proof. TOPROVE 16 ◀

Now we show that Step 3 of Algorithm 3, if applicable, is correct.

► **Lemma 6.11.** *Let $(\text{Rest}, \text{In}, \text{Out})$ be a 3-partition of V that appears during some iteration of Algorithm 3. Assume there exists a slim S -nice tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G with width k that contains a directed path of length at least three with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, or if $\hat{L}^S = \emptyset$ then \mathcal{T} contains an S -bottom node that has an S -child with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, such that neither t_{\max} nor the parent of t_{\max} are contained in a full join tree, and with target node t^* such that the following holds.*

- $\text{In} \subseteq X_{t^*}$ and $X_{t^*} \cap \text{Out} = \emptyset$,
- if t^* has a parent t_p , then $X_{t_p} = X_{t^*} \setminus D_R$,
- if t^* has one S -child t_1 , then $X_{t_1} = X_{t^*} \setminus D_{L_1}$,
- if t^* has two S -children t_1 and t_2 , then $X_{t_1} = X_{t^*} \setminus D_{L_1}$ and $X_{t_2} = X_{t^*} \setminus D_{L_2}$,

- Steps 1 and 2 of Algorithm 3 do not apply, and
- there is a vertex $v \in \text{Rest}$ that is contained in an R -component in $G[(\text{Out} \cup \{v\}) \setminus S]$ in $G[(\text{Out} \cup \{v\}) \setminus S]$ and that has degree at most one in $G[\text{Rest}]$.

Then there is a slim S -nice tree decomposition $\mathcal{T}' = (T', \{X'_t\}_{t \in V(T')})$ of G with width at most k such that the following holds.

- \mathcal{T}' is a sibling of \mathcal{T} ,
- if \mathcal{T} is top-heavy for a node \hat{t} such that \hat{t} is
 - not in $T_{t_a}^*$ and not an ancestor of t_a^* , or
 - a descendant of t^* ,
 then \mathcal{T}' is also top-heavy for \hat{t} and the subtree rooted at \hat{t} is the same as in \mathcal{T} ,
- $\text{In} \subseteq X'_{t^*}$ and $X'_{t^*} \cap (\text{Out} \cup \{v\}) = \emptyset$, and
- $|X'_{t^*}| < |X_{t^*}|$ or each of the following holds:
 - $|X'_{t^*}| = |X_{t^*}|$,
 - if t^* has a parent t'_p , then $X'_{t'_p} = X'_{t^*} \setminus D_R$,
 - if t^* has one S -child t'_1 , then $X'_{t'_1} = X'_{t^*} \setminus D_{L_1}$, and
 - if t^* has two S -children t'_1 and t'_2 , then $X'_{t'_1} = X'_{t^*} \setminus D_{L_1}$ and $X'_{t'_2} = X'_{t^*} \setminus D_{L_2}$.

Proof. TOPROVE 17 ◀

Now we show that Step 4 of Algorithm 3, if applicable, is correct. This proof has many similarities with the one of Lemma 6.11, but there are some key differences.

► **Lemma 6.12.** *Let $(\text{Rest}, \text{In}, \text{Out})$ be a 3-partition of V that appears during some iteration of Algorithm 3. Assume there exists a slim S -nice tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G with width k that contains a directed path of length at least three with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, or if $\hat{L}^S = \emptyset$ then \mathcal{T} contains an S -bottom node that has an S -child with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, such that neither t_{\max} nor the parent of t_{\max} are contained in a full join tree, and with target node t^* such that the following holds.*

- $\text{In} \subseteq X_{t^*}$ and $X_{t^*} \cap \text{Out} = \emptyset$,
- if t^* has a parent t_p , then $X_{t_p} = X_{t^*} \setminus D_R$,
- if t^* has one S -child t_1 , then $X_{t_1} = X_{t^*} \setminus D_{L_1}$,
- if t^* has two S -children t_1 and t_2 , then $X_{t_1} = X_{t^*} \setminus D_{L_1}$ and $X_{t_2} = X_{t^*} \setminus D_{L_2}$,
- Steps 1, 2, and 3 of Algorithm 3 do not apply, and
- there is a vertex $v \in \text{Rest}$ that is contained in an H -component in $G[(\text{Out} \cup \{v\}) \setminus S]$ for $H \in \{L_1, L_2\}$ in $G[(\text{Out} \cup \{v\}) \setminus S]$ and that has degree at most one in $G[\text{Rest}]$.

Then there is a slim S -nice tree decomposition $\mathcal{T}' = (T', \{X'_t\}_{t \in V(T')})$ of G with width at most k such that the following holds.

- \mathcal{T}' is a sibling of \mathcal{T} ,
- if \mathcal{T} is top-heavy for a node \hat{t} such that \hat{t} is
 - not in $T_{t_a}^*$ and not an ancestor of t_a^* , or
 - a descendant of t^* ,
 then \mathcal{T}' is also top-heavy for \hat{t} and the subtree rooted at \hat{t} is the same as in \mathcal{T} ,
- $\text{In} \subseteq X'_{t^*}$ and $X'_{t^*} \cap (\text{Out} \cup \{v\}) = \emptyset$, and
- $|X'_{t^*}| < |X_{t^*}|$ or each of the following holds:
 - if t^* has a parent t'_p , then $X'_{t'_p} = X'_{t^*} \setminus D_R$,
 - if t^* has one S -child t'_1 , then $X'_{t'_1} = X'_{t^*} \setminus D_{L_1}$, and
 - if t^* has two S -children t'_1 and t'_2 , then $X'_{t'_1} = X'_{t^*} \setminus D_{L_1}$ and $X'_{t'_2} = X'_{t^*} \setminus D_{L_2}$.

Proof. TOPROVE 18 ◀

Next, we show that the Step 5 of Algorithm 3, if applicable, is correct. Again, this proof has many similarities to the previous ones.

► **Lemma 6.13.** *Let $(\text{Rest}, \text{In}, \text{Out})$ be a 3-partition of V that appears during some iteration of Algorithm 3. Assume there exists a slim S -nice tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G with width k that contains a directed path of length at least three with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, or if $\hat{L}^S = \emptyset$ then \mathcal{T} contains an S -bottom node that has an S -child with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, such that neither t_{\max} nor the parent of t_{\max} are contained in a full join tree, and with target node t^* such that the following holds.*

- $\text{In} \subseteq X_{t^*}$ and $X_{t^*} \cap \text{Out} = \emptyset$,
- if t^* has a parent t_p , then $X_{t_p} = X_{t^*} \setminus D_R$,
- if t^* has one S -child t_1 , then $X_{t_1} = X_{t^*} \setminus D_{L_1}$,
- if t^* has two S -children t_1 and t_2 , then $X_{t_1} = X_{t^*} \setminus D_{L_1}$ and $X_{t_2} = X_{t^*} \setminus D_{L_2}$,
- Steps 1, 2, 3, and 4 of Algorithm 3 do not apply, and
- there is a vertex $v \in \text{Rest}$ that has degree one in $G[\text{Rest}]$ and the connected component C of $G[(\text{Out} \cup \{v\}) \setminus S]$ that contains v has the property that $N(V(C)) \cap \text{In} \neq \text{In}$.

Then there is a slim S -nice tree decomposition $\mathcal{T}' = (T', \{X'_t\}_{t \in V(T')})$ of G with width at most k such that the following holds.

- \mathcal{T}' is a sibling of \mathcal{T} ,
- if \mathcal{T} is top-heavy for a node \hat{t} such that \hat{t} is
 - not in $T_{t_a}^*$ and not an ancestor of t_a^* , or
 - a descendant of t^* ,
 then \mathcal{T}' is also top-heavy for \hat{t} and the subtree rooted at \hat{t} is the same as in \mathcal{T} ,
- $\text{In} \subseteq X'_{t^*}$ and $X'_{t^*} \cap (\text{Out} \cup \{v\}) = \emptyset$, and
- $|X'_{t^*}| < |X_{t^*}|$ or each of the following holds:
 - if t^* has a parent t'_p , then $X'_{t'_p} = X'_{t^*} \setminus D_R$,
 - if t^* has one S -child t'_1 , then $X'_{t'_1} = X'_{t^*} \setminus D_{L_1}$, and
 - if t^* has two S -children t'_1 and t'_2 , then $X'_{t'_1} = X'_{t^*} \setminus D_{L_1}$ and $X'_{t'_2} = X'_{t^*} \setminus D_{L_2}$.

Proof. TOPROVE 19 ◀

Finally, we show that Step 6 of Algorithm 3 is correct.

► **Lemma 6.14.** *Let $(\text{Rest}, \text{In}, \text{Out})$ be a 3-partition of V that appears during some iteration of Algorithm 3. Assume there exists a slim S -nice tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G with width k that contains a directed path of length at least three with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, or if $\hat{L}^S = \emptyset$ then \mathcal{T} contains an S -bottom node that has an S -child with S -trace $(\hat{L}^S, \hat{X}^S, \hat{R}^S)$, such that neither t_{\max} nor the parent of t_{\max} are contained in a full join tree, and with target node t^* such that the following holds.*

- $\text{In} \subseteq X_{t^*}$ and $X_{t^*} \cap \text{Out} = \emptyset$,
- if t^* has a parent t_p , then $X_{t_p} = X_{t^*} \setminus D_R$,
- if t^* has one S -child t_1 , then $X_{t_1} = X_{t^*} \setminus D_{L_1}$,
- if t^* has two S -children t_1 and t_2 , then $X_{t_1} = X_{t^*} \setminus D_{L_1}$ and $X_{t_2} = X_{t^*} \setminus D_{L_2}$,
- Steps 1, 2, 3, 4, and 5 of Algorithm 3 do not apply, and
- there is a vertex $v \in \text{Rest}$ that has degree at most one in $G[\text{Rest}]$.

Then there is a slim S -nice tree decomposition $\mathcal{T}' = (T', \{X'_t\}_{t \in V(T')})$ of G with width at most k such that the following holds.

- \mathcal{T}' is a sibling of \mathcal{T} ,
- if \mathcal{T} is top-heavy for a node \hat{t} such that \hat{t} is
 - not in $T_{t_a}^*$ and not an ancestor of t_a^* , or

- a descendant of t^* ,
- then \mathcal{T}' is also top-heavy for \hat{t} and the subtree rooted at \hat{t} is the same as in \mathcal{T} ,
- $\text{In} \subseteq X'_{t^*}$ and $X'_{t^*} \cap (\text{Out} \cup \{v\}) = \emptyset$, and
- $|X'_{t^*}| < |X_{t^*}|$ or each of the following holds:
 - if t^* has a parent t'_p , then $X'_{t'_p} = X'_{t^*} \setminus D_R$,
 - if t^* has one S -child t'_1 , then $X'_{t'_1} = X'_{t^*} \setminus D_{L_1}$, and
 - if t^* has two S -children t'_1 and t'_2 , then $X'_{t'_1} = X'_{t^*} \setminus D_{L_1}$ and $X'_{t'_2} = X'_{t^*} \setminus D_{L_2}$.

Proof. TOPROVE 20 ◀

Lemma 6.7 now follows directly from Lemmas 6.8, 6.9, and 6.11–6.14.

6.3 Summary and Putting the Pieces Together

Recall from the beginning of Section 6 that we wish to compute the following.

- A candidate for the bag X_{\max}^ϕ of t_{\max} .
- A candidate for the bag X_p^ϕ of the parent of t_{\max} except for the case where $\tau_+ = \text{void}$ (in this case t_{\max} is the root of the tree decomposition).
- A candidate for the bag X_{\min}^ϕ of t_{\min} except for the case where $\tau_- = \text{void}$ (in this case t_{\min} is not defined).
- A candidate for the union of the bags X_c^ϕ of the S -children of t_{\min} except for the case where $\tau_- = \text{void}$ (in this case t_{\min} is not defined).
- A candidate for the set X_{path}^ϕ of vertices that we want to add to bags of the directed path or subtrees of the tree decomposition that are rooted at children of nodes in the directed path.

In Sections 6.1 and 6.2 we explained how to compute X_{\max}^ϕ , depending on whether the top operation τ_+ is a

1. **bigjoin** S -operation or $\tau_+ = \text{extendedforget}(v, d, \text{true}, \tau)$ and τ is a **bigjoin** S -operation, or
2. any of the remaining cases except $\tau_+ = \text{void}$.

In the first case, the bag X_{\max}^ϕ and the bag X_p^ϕ of the parent of t_{\max} can directly be determined via the additional information in the **bigjoin** S -operation. In the second case, Algorithm 2 outputs all necessary information to determine X_{\max}^ϕ and the bag of the parent of t_{\max} . Note that in both cases, we also have that $X_{\max}^\phi \subseteq X_p^\phi$ unless $\tau_+ = \text{extendedforget}(v, d, f, \tau)$, then we have that $X_{\max}^\phi = X_p^\phi \cup \{v\}$. In the case where $\tau_+ = \text{void}$, we set $X_{\max}^\phi = X^S$ and $X_p^\phi = \emptyset$.

In the case where we wish to determine X_{\min}^ϕ , we do the following. Note that the parent of the top node of the directed path corresponding to the S -trace of a predecessor state is t_{\min} . This means that we can apply our algorithms to a predecessor state ψ to determine X_{\min}^ϕ . More specifically, we have that $X_{\min}^\phi = X_p^\psi$. Since the computation of X_p^ψ is independent from the bottom operation of state ψ , we can pick an arbitrary predecessor state ψ of ϕ for the computation. In the case where $\tau_- = \text{void}$ and ϕ does not have any predecessor states, we set $X_{\min}^\phi = X^S$. Similarly, we have that $X_c^\phi = X_{\max}^\psi$ if $\tau_- \neq \text{void}$ and τ_- is neither **smalljoin** nor **bigjoin**. In the case where τ_- is a **smalljoin** or a **bigjoin** S -operation, there is a pair we pick an arbitrary pair ψ_1, ψ_2 of predecessor states (see Definition 4.14) and we have that $X_c^\phi = X_{\max}^{\psi_1} \cup X_{\max}^{\psi_2}$. If $\tau_- = \text{void}$, then we set $X_c^\phi = \emptyset$.

From the definition of X_{path}^ϕ in Section 5.3 it follows that, given X_{\max}^ϕ , X_p^ϕ , X_{\min}^ϕ , and X_c^ϕ , we can compute X_{path}^ϕ in polynomial time.

7 Computing the Local Treewidth

In this section we present a polynomial-time algorithm to compute the function LTW. Formally, we show the following.

► **Proposition 7.1.** *The function $\text{LTW} : \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ can be computed in polynomial time.*

By Definition 5.10, the task of the function LTW is to compute the treewidth of graph $G'[X_{\max}^\phi \cup X_p^\phi \cup X_{\min}^\phi \cup X_{\text{path}}^\phi]$, where G' is the graph obtained from the input graph G by adding edges between all pairs of vertices $u, v \in X_{\max}^\phi$ with $\{u, v\} \notin E$, all pairs of vertices $u, v \in X_p^\phi$ with $\{u, v\} \notin E$, and all pairs of vertices $u, v \in X_{\min}^\phi$ with $\{u, v\} \notin E$. Note that by definition we have $X_{\max}^\phi \subseteq X_p^\phi$ or $X_p^\phi \subseteq X_{\max}^\phi$ and that $G[X_{\max}^\phi \cup X_p^\phi \cup X_{\min}^\phi \cup X_{\text{path}}^\phi] - (X_{\max}^\phi \cap X_p^\phi \cap X_{\min}^\phi)$ is a forest. The latter will allow us to infer some important properties on the edge set $E(G'[X_{\max}^\phi \cup X_p^\phi \cup X_{\min}^\phi \cup X_{\text{path}}^\phi])$.

We will first show how to solve the following simpler problem: Given a graph $G = (V_1 \cup V_2, E)$ such that $G[V_1]$ is complete and $G[V_2]$ is a forest, compute an optimal tree decomposition for G . We show that this problem can be solved in polynomial time and we will use this as a subroutine in the algorithm for computing LTW. We can observe the following. Since $G[V_1]$ is complete, we have by Lemmas 3.2 and 3.3 that $\text{tw}(G) \geq |V_1| - 1$. Furthermore, since $G[V_2]$ is a forest, we have that $\text{tw}(G) \leq |V_1| + 1$, since we can obtain a tree decomposition for G by taking any tree decomposition for $G[V_2]$ (of width one) and adding V_1 to every bag. Hence, the task of the algorithm is to decide which of the three options applies, and to construct a tree decomposition of the corresponding width. Formally, we show the following.

► **Lemma 7.2.** *Let $G = (V_1 \cup V_2, E)$ be a graph such that $G[V_1]$ is complete and $G[V_2]$ is a forest. A tree decomposition for G with minimum width can be computed in polynomial time.*

Proof. TOPROVE 21 ◀

Now we prove Proposition 7.1 with the help of Lemma 7.2.

Proof. TOPROVE 22 ◀

We remark that the above described algorithm explicitly constructs a tree decomposition that it can output. We describe in Section 9 how to use this to construct a tree decomposition for the whole input graph G .

8 Legal Predecessor States

In this section, we formally define the functions legal_1 and legal_2 . We show that we can compute the functions using the algorithms introduced in Section 6. Recall that the functions legal_1 and legal_2 , intuitively, should check whether a state ϕ is compatible with its predecessor state(s). To this end, for a state $\phi = (\tau_-, L^S, X^S, R^S, \tau_+)$, we define three sets V_L^ϕ , V_X^ϕ , and V_R^ϕ which, intuitively, determine which vertices are in bags “below” t_{\min} , which vertices are in bags “between” t_{\max} and t_{\min} , and the remaining vertices. We use the five sets X_{\max}^ϕ , X_p^ϕ , X_{\min}^ϕ , X_c^ϕ , and X_{path}^ϕ to determine those sets. Using those sets, we define V_L^ϕ , V_X^ϕ , and V_R^ϕ as follows.

- $v \in V_L^\phi \Leftrightarrow v \in L^S \cup (X_c^\phi \setminus X_{\min}^\phi)$ or v is connected to some $u \in L^S \cup (X_c^\phi \setminus X_{\min}^\phi)$ in $G - X_{\min}^\phi$.

- $V_X^\phi = X_{\max}^\phi \cup X_p^\phi \cup X_{\min}^\phi \cup X_{\text{path}}^\phi$.
- $V_R^\phi = V \setminus (V_L^\phi \cup V_X^\phi)$.

Now we are ready to define legal_1 and legal_2 as follows.

► **Definition 8.1** (legal_1). *Let ϕ be a state and let $\psi \in \Psi(\phi)$. Then $\text{legal}_1(\phi, \psi) = \text{true}$ if and only if all of the following three conditions hold.*

1. $V_L^\psi \cup V_X^\psi \subseteq V_L^\phi \cup V_X^\phi$.
2. $N[V_R^\psi] \cap V_L^\phi = \emptyset$.
3. $(V_L^\psi \cup V_X^\psi) \cap V_X^\phi \subseteq X_{\min}^\phi$.

► **Definition 8.2** (legal_2). *Let ϕ be a state let $\psi_1 \in \Psi_1(\phi)$, and let $\psi_2 \in \Psi_2(\phi)$. Then $\text{legal}_2(\phi, \psi_1, \psi_2) = \text{true}$ if and only if all of the following conditions hold.*

1. $V_L^{\psi_1} \cup V_X^{\psi_1} \cup V_L^{\psi_2} \cup V_X^{\psi_2} \subseteq V_L^\phi \cup V_X^\phi$.
2. $N[V_R^{\psi_1}] \cap V_L^\phi \subseteq V_L^{\psi_2} \cup V_X^{\psi_2}$.
3. $N[V_R^{\psi_2}] \cap V_L^\phi \subseteq V_L^{\psi_1} \cup V_X^{\psi_1}$.
4. $(V_L^{\psi_1} \cup V_X^{\psi_1}) \cap V_X^\phi \subseteq X_{\min}^\phi$.
5. $(V_L^{\psi_2} \cup V_X^{\psi_2}) \cap V_X^\phi \subseteq X_{\min}^\phi$.
6. $(V_L^{\psi_1} \cup V_X^{\psi_1}) \cap (V_L^{\psi_2} \cup V_X^{\psi_2}) \subseteq X_{\min}^\phi$.

Note that we can clearly compute the sets V_L^ϕ , V_X^ϕ , and V_R^ϕ from the sets X_{\max}^ϕ , X_p^ϕ , X_{\min}^ϕ , and X_{path}^ϕ in polynomial time. The sets X_{\max}^ϕ , X_p^ϕ , and X_{\min}^ϕ can be computed in polynomial time by Observations 6.1 and 6.5. By the definition of X_{path}^ϕ given in Section 5.3 we have that we can compute X_{path}^ϕ from X_{\max}^ϕ , X_p^ϕ , and X_{\min}^ϕ in polynomial time. Hence, we have the following.

► **Observation 8.3.** *The functions $\text{legal}_1 : \mathcal{S} \times \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ and $\text{legal}_2 : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ can be computed in polynomial time.*

Now we show that if we have a slim S -nice tree decomposition of G with width k that witnesses a state $\phi = (\tau_-, L^S, X^S, R^S, \tau_+)$ and the predecessor state(s) of ϕ and is top-heavy for the parent of the parent of the top node of the directed path of S -trace (L^S, X^S, R^S) , then we have, depending on the number of predecessor states, that $\text{legal}_1(\phi, \psi) = \text{true}$ and $\text{legal}_2(\phi, \psi_1, \psi_2) = \text{true}$. To this end, we first prove the following.

► **Lemma 8.4.** *Let \mathcal{T} be a slim S -nice tree decomposition of G with width k that witnesses $\phi = (\tau_-, L^S, X^S, R^S, \tau_+)$. If $L^S \neq \emptyset$, then let t_{\min} and t_{\max} be the bottom node and the top node of the directed path of S -trace (L^S, X^S, R^S) , respectively. If $L^S = \emptyset$, then let t_{\max} be the S -child with S -trace (L^S, X^S, R^S) of the S -bottom node that admits S -operation τ_+ . Let t_p be the parent of t_{\max} (if it exists). The bags of t_{\min} and t_{\max} are X_{\min}^ϕ and X_{\max}^ϕ , respectively. If t_{\min} is not defined, then $X_{\min}^\phi = X^S$. If t_p exists, its bag is X_p^ϕ , otherwise $X_p^\phi = \emptyset$. Let the tree decomposition \mathcal{T} be top-heavy for the parent of t_p . It holds that $v \in V_L^\phi$ if and only if $L^S \neq \emptyset$ and t_{\min} has an S -child t such that $v \in V_t \setminus X_{\min}^\phi$.*

Proof. TOPROVE 23 ◀

► **Lemma 8.5.** *Let \mathcal{T} be a slim S -nice tree decomposition of G with width k that witnesses $\phi = (\tau_-, L^S, X^S, R^S, \tau_+)$. If $L^S \neq \emptyset$, then let t_{\min} and t_{\max} be the bottom node and the top node of the directed path of S -trace (L^S, X^S, R^S) , respectively. If $L^S = \emptyset$, then let t_{\max} be the S -child with S -trace (L^S, X^S, R^S) of the S -bottom node that admits S -operation τ_+ . Let t_p be the parent of t_{\max} (if it exists). The bags of t_{\min} and t_{\max} are X_{\min}^ϕ and X_{\max}^ϕ ,*

respectively. If t_{\min} is not defined, then $X_{\min}^\phi = X^S$. If t_p exists, its bag is X_p^ϕ , otherwise $X_p^\phi = \emptyset$. Let the tree decomposition \mathcal{T} be top-heavy for the parent of t_p . It holds that $v \in V_X^\phi$ if and only if $v \in X_p^\phi$ or both of the following conditions hold.

- $v \in V_{t_{\max}}$, and
- if $L^S \neq \emptyset$, then for each S -child t of t_{\min} it holds that $v \notin V_t \setminus X_{\min}^\phi$.

Proof. TOPROVE 24 ◀

► **Corollary 8.6.** Let \mathcal{T} be a slim S -nice tree decomposition of G with width k that witnesses $\phi = (\tau_-, L^S, X^S, R^S, \tau_+)$. If $L^S \neq \emptyset$, then let t_{\min} and t_{\max} be the bottom node and the top node of the directed path of S -trace (L^S, X^S, R^S) , respectively. If $L^S = \emptyset$, then let t_{\max} be the S -child with S -trace (L^S, X^S, R^S) of the S -bottom node that admits S -operation τ_+ . Let t_p be the parent of t_{\max} (if it exists). The bags of t_{\min} and t_{\max} are X_{\min}^ϕ and X_{\max}^ϕ , respectively. If t_{\min} is not defined, then $X_{\min}^\phi = X^S$. If t_p exists, its bag is X_p^ϕ , otherwise $X_p^\phi = \emptyset$. Let the tree decomposition \mathcal{T} be top-heavy for the parent of t_p . It holds that $V_{t_{\max}} \cup X_p^\phi = V_L^\phi \cup V_X^\phi$.

Now we are ready to prove that the requirements for $\text{legal}_1(\phi, \psi) = \text{true}$ and $\text{legal}_2(\phi, \psi_1, \psi_2) = \text{true}$, respectively, are given if there is a slim S -nice tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G with width k that witnesses ϕ and ψ or ψ_1 and ψ_2 respectively.

► **Lemma 8.7.** Let $\phi = (\tau_-, L^S, X^S, R^S, \tau_+)$ be a state and let $\psi = (\tau'_-, \hat{L}^S, \hat{X}^S, \hat{R}^S, \tau_-) \in \Psi(\phi)$ such that the following holds. There is a slim S -nice tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G with width k that witnesses ϕ and ψ . Let t_{\min} and t_{\max} be the bottom and top node, respectively, of the directed path of S -trace (L^S, X^S, R^S) . Let t_p be the parent of t_{\max} . The tree decomposition \mathcal{T} is top-heavy for the parent of t_p . Then we have the following.

1. $V_L^\psi \cup V_X^\psi \subseteq V_L^\phi \cup V_X^\phi$.
2. $N[V_R^\psi] \cap V_L^\phi = \emptyset$.
3. $(V_L^\psi \cup V_X^\psi) \cap V_X^\phi = X_{\min}^\phi$.

Proof. TOPROVE 25 ◀

► **Lemma 8.8.** Let $\phi = (\tau_-, L^S, X^S, R^S, \tau_+)$ be a state let $\psi_1 = (\tau'_-, \hat{L}^S, \hat{X}^S, \hat{R}^S, \tau_-) \in \Psi_1(\phi)$, and let $\psi_2 = (\tau''_-, \check{L}^S, \check{X}^S, \check{R}^S, \tau_-) \in \Psi_2(\phi)$ such that the following holds. There is a slim S -nice tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G with width k that witnesses ϕ , ψ_1 , and ψ_2 . Let t_{\min} and t_{\max} be the bottom and top node, respectively, of the directed path of S -trace (L^S, X^S, R^S) . Let t_p be the parent of t_{\max} . The tree decomposition \mathcal{T} is top-heavy for the parent of t_p . Then we have the following.

1. $V_L^{\psi_1} \cup V_X^{\psi_1} \cup V_L^{\psi_2} \cup V_X^{\psi_2} \subseteq V_L^\phi \cup V_X^\phi$.
2. $N[V_R^{\psi_1}] \cap V_L^\phi \subseteq V_L^{\psi_2} \cup V_X^{\psi_2}$.
3. $N[V_R^{\psi_2}] \cap V_L^\phi \subseteq V_L^{\psi_1} \cup V_X^{\psi_1}$.
4. $(V_L^{\psi_1} \cup V_X^{\psi_1}) \cap V_X^\phi = X_{\min}^\phi$.
5. $(V_L^{\psi_2} \cup V_X^{\psi_2}) \cap V_X^\phi = X_{\min}^\phi$.
6. $(V_L^{\psi_1} \cup V_X^{\psi_1}) \cap (V_L^{\psi_2} \cup V_X^{\psi_2}) = X_{\min}^\phi$.

Proof. TOPROVE 26 ◀

9 Correctness and Running Time Analysis

In this section, we finally prove that our overall dynamic programming algorithm is correct and give a running time analysis. The results from this section will imply Theorem 1.1. To this end, we first show that if $\text{PTW}(\phi) = \text{true}$ for some $\phi = (\tau_-, L^S, X^S, R^S, \tau_+)$, then we

can compute a partial tree decomposition for the input graph, intuitively, until the top node of the directed path of S -trace (L^S, X^S, R^S) . For the formal statement, we use the sets V_L^ϕ , V_X^ϕ , and V_R^ϕ defined in Section 8. We show the following.

► **Proposition 9.1.** *Let $\phi = (\tau_-, L^S, X^S, R^S, \tau_+)$ be a state such that $\tau_- = \text{void}$ if and only if $L^S = \emptyset$, and if $\tau_+ = \text{void}$ then $R^S = \emptyset$. If $\text{PTW}(\phi) = \text{true}$, then there exists a rooted tree decomposition \mathcal{T} of width at most k for $G[V_L^\phi \cup V_X^\phi]$ such that the root of \mathcal{T} has bag X_p^ϕ .*

Proof. TOPROVE 27 ◀

We remark that in the proof of Proposition 9.1, we explicitly construct a tree decomposition. Hence, we can save this tree decomposition in the dynamic programming table alongside the Boolean value that indicates whether the tree decomposition has width at most $k + 1$ or not. This allows the algorithm to output a tree decomposition for the whole input graph G .

For the other direction of the correctness, we prove the following. We first show that there exists a slim S -nice tree decomposition of optimal width where every S -bottom node admits an extended S -operation. Then, intuitively, for every directed path in that tree decomposition, there is a state that is witnessed, and via those states our algorithm will find this tree decomposition and output true. To this end, we do the following. Given a slim S -nice tree decomposition \mathcal{T} of optimal width, we take an arbitrary but fixed linear order to the S -traces that is a linearization of their processor relation on \mathcal{T} . Recall that by definition, siblings preserve the predecessor relation. By induction on this linear order, we show that we can transform \mathcal{T} into a slim S -nice tree decomposition \mathcal{T}' of the same or smaller width, that is a sibling of \mathcal{T} , such that each S -bottom node admits some extended S -operation. The following lemma gives us the induction basis and step.

► **Lemma 9.2.** *Assume there exists a slim S -nice tree decomposition \mathcal{T} for G with width at most k that has the following properties. Let (L^S, X^S, R^S) be an S -trace with $L^S \neq \emptyset$.*

- \mathcal{T} contains an S -bottom t node that has S -trace (L^S, X^S, R^S) .
- For each descendant t' of t that is an S -bottom node we have that t' admit some extended S -operation, and if t' is contained in a full join tree T , then \mathcal{T} is top-heavy for the parent of the root of T . Otherwise, \mathcal{T} is top-heavy for the parent of t' .

Then there exists a slim S -nice tree decomposition \mathcal{T}' for G with width at most k such that the following holds.

- \mathcal{T}' is a sibling of \mathcal{T} and t is the S -bottom node in \mathcal{T}' with S -trace (L^S, X^S, R^S) ,
- if \mathcal{T} is top-heavy for a node \hat{t} such that \hat{t} is the parent of an S -bottom node, and
 - not in T_t and not an ancestor of t , or
 - a descendant of t ,
 then \mathcal{T}' is also top-heavy for \hat{t} and the subtree rooted at \hat{t} is the same as in \mathcal{T} ,
- All descendants of t that are S -bottom nodes admit some extended S -operation.
- Node t admits some extended S -operation τ .
- If t is contained in a full join tree T in \mathcal{T}' , then \mathcal{T}' is top-heavy for the parent of the root of T . Otherwise, \mathcal{T}' is top-heavy for the parent of t .

Proof. TOPROVE 28 ◀

Now we use Lemma 9.2 to argue that we can obtain a slim S -nice tree decomposition \mathcal{T} for G such that every S -bottom node admits an extended S -operation.

► **Corollary 9.3.** *Assume G has treewidth at most k . Then there exists a slim S -nice tree decomposition \mathcal{T} for G with width at most k such that every S -bottom node admits an extended S -operation.*

Proof. TOPROVE 29 ◀

Corollary 9.3 allows us to assume that each S -bottom node admits some extended S -operation. Under this assumption we can prove the following in a straightforward way.

► **Proposition 9.4.** *Assume there exists a slim S -nice tree decomposition \mathcal{T} for G with width k such that every S -bottom node in \mathcal{T} admits an extended S -operation. If \mathcal{T} witnesses a state ϕ , then $\text{PTW}(\phi) = \text{true}$.*

Proof. TOPROVE 30 ◀

From Propositions 9.1 and 9.4 and the definition of V_L^ϕ , V_X^ϕ , and V_R^ϕ in Section 8 we get the following.

► **Corollary 9.5.** *There is a tree decomposition for G with width at most k if and only if*

$$\bigvee_{\phi \in \Phi} \text{PTW}(\phi) = \text{true},$$

where Φ contains all states of the form $(\tau_-, S \setminus X^S, X^S, \emptyset, \text{void})$ for some extended S -operation τ_- .

Proof. TOPROVE 31 ◀

Finally, we analyze the size of the dynamic programming table and the time needed to compute one entry.

► **Proposition 9.6.** *The dynamic programming table PTW has size $2^{\mathcal{O}(\text{fwn}(G))}$ and each entry can be computed in $2^{\mathcal{O}(\text{fwn}(G))}$ time.*

Proof. TOPROVE 32 ◀

Theorem 1.1 now follows from Corollary 9.5, Proposition 9.6, the fact that a minimum feedback vertex set can be computed in $2.7^{\text{fwn}(G)} \cdot n^{\mathcal{O}(1)}$ time (randomized) [63] or in $3.6^{\text{fwn}(G)} \cdot n^{\mathcal{O}(1)}$ deterministic time [58], and that we apply the kernelization algorithm by Bodlaender, Jansen, and Kratsch [24] to reduce the number of vertices in the input graph to $\mathcal{O}(\text{fwn}(G)^4)$ while leaving the minimum feedback vertex set size unchanged.

10 Conclusion

In this paper, we showed that a minimum tree decomposition for a graph G can be computed in single exponential time in the feedback vertex number of the input graph, that is, in $2^{\mathcal{O}(\text{fwn}(G))} \cdot n^{\mathcal{O}(1)}$ time. This improves the previously known result that a minimum tree decomposition for a graph G can be computed in $2^{\mathcal{O}(\text{vcn}(G))} \cdot n^{\mathcal{O}(1)}$ time [33, 47]. We believe that this can also be seen either as an important step towards a positive resolution of the open problem of finding a $2^{\mathcal{O}(\text{tw}(G))} \cdot n^{\mathcal{O}(1)}$ time algorithm for computing an optimal tree decomposition, or, if its answer is negative, then a mark of the tractability border of single exponential time algorithms for the computation of treewidth.

A natural future research direction is to explore to which extent our techniques can be used to obtain a single exponential time algorithm for treewidth computation for other

parameters that measure the vertex deletion distance to a graph class where the treewidth is small or can be computed in polynomial time. Natural candidates would be e.g. series-parallel graphs (graphs of treewidth two) or chordal graphs. We remark that our algorithm needs to know the deletion set, so in order to obtain a single exponential running time, we need to be able to compute the deletion set fast enough. For a minimum vertex deletion set to chordal graphs, it is currently not known whether it can be computed in single exponential time [31]. However, the chordal vertex deletion number is incomparable to the treewidth, and hence any polynomial time algorithm for graphs with a constant-size chordal vertex deletion set would be interesting.

References

- 1 Abhijin Adiga, Rajesh Hemant Chitnis, and Saket Saurabh. Parameterized algorithms for boxicity. In *Proceedings of the 21st International Symposium on Algorithms and Computation (ISAAC)*, volume 6506 of *LNCS*, pages 366–377. Springer, 2010.
- 2 Akanksha Agrawal, Daniel Lokshantov, Amer E Mouawad, and Saket Saurabh. Simultaneous feedback vertex set: A parameterized perspective. *ACM Transactions on Computation Theory (TOCT)*, 10(4):1–25, 2018.
- 3 Eyal Amir. Approximation algorithms for treewidth. *Algorithmica*, 56:448–479, 2010.
- 4 Stefan Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability—a survey. *BIT Numerical Mathematics*, 25(1):1–23, 1985.
- 5 Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k -tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.
- 6 Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12(2):308–340, 1991.
- 7 Stefan Arnborg and Andrzej Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k -trees. *Discrete Applied Mathematics*, 23(1):11–24, 1989.
- 8 Stefan Arnborg, Andrzej Proskurowski, and Derek G Corneil. Forbidden minors characterization of partial 3-trees. *Discrete Mathematics*, 80(1):1–19, 1990.
- 9 Ann Becker, Reuven Bar-Yehuda, and Dan Geiger. Randomized algorithms for the loop cutset problem. *Journal of Artificial Intelligence Research*, 12:219–234, 2000.
- 10 Mahdi Belbasi and Martin Fürer. Finding all leftmost separators of size $\leq k$. In *Proceedings of the 15th International Conference on Combinatorial Optimization and Applications (COCOA)*, volume 13135 of *LNCS*, pages 273–287. Springer, 2021.
- 11 Mahdi Belbasi and Martin Fürer. An improvement of Reed’s treewidth approximation. *Journal of Graph Algorithms and Applications*, 26(2):257–282, 2022.
- 12 Marshall W. Bern, Eugene L. Lawler, and Alice L. Wong. Linear-time computation of optimal subgraphs of decomposable graphs. *Journal of Algorithms*, 8(2):216–235, 1987.
- 13 Umberto Bertelè and Francesco Brioschi. *Nonserial dynamic programming*. Academic Press, Inc., 1972.
- 14 Hans L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In *Proceedings of the 15th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 317 of *LNCS*, pages 105–118. Springer, 1988.
- 15 Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11(1-2):1–21, 1993.
- 16 Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996.
- 17 Hans L. Bodlaender. Treewidth: Algorithmic techniques and results. In *Proceedings of the 22nd International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 1295 of *LNCS*, pages 19–36. Springer, 1997.
- 18 Hans L. Bodlaender. A partial k -arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209(1-2):1–45, 1998.

- 19 Hans L. Bodlaender. Discovering treewidth. In *Proceedings of the 31st Annual Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, volume 3381 of *LNCS*, pages 1–16. Springer, 2005.
- 20 Hans L. Bodlaender. Treewidth: Characterizations, applications, and computations. In *Proceedings of the 32nd International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, volume 4271 of *LNCS*, pages 1–14. Springer, 2006.
- 21 Hans L. Bodlaender. Treewidth: Structure and algorithms. In *Proceedings of the 14th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, volume 4474 of *LNCS*, pages 11–25. Springer, 2007.
- 22 Hans L. Bodlaender, Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. *Information and Computation*, 243:86–111, 2015.
- 23 Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshantov, and Michał Pilipczuk. A $c^k n$ 5-approximation algorithm for treewidth. *SIAM Journal on Computing*, 45(2):317–378, 2016.
- 24 Hans L. Bodlaender, Bart M.P. Jansen, and Stefan Kratsch. Preprocessing for treewidth: A combinatorial analysis through kernelization. *SIAM Journal on Discrete Mathematics*, 27(4):2108–2142, 2013.
- 25 Hans L. Bodlaender and Ton Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms*, 21(2):358–402, 1996.
- 26 Hans L. Bodlaender and Arie M.C.A. Koster. Safe separators for treewidth. *Discrete Mathematics*, 306(3):337–350, 2006.
- 27 Hans L. Bodlaender and Arie M.C.A. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2008.
- 28 Édouard Bonnet. Treewidth inapproximability and tight ETH lower bound. *CoRR*, abs/2406.11628, 2024. Accepted for publication in the proceedings of the 57th Annual ACM Symposium on Theory of Computing (STOC 2025). URL: <https://doi.org/10.48550/arXiv.2406.11628>.
- 29 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. *Algorithmica*, 7:555–581, 1992.
- 30 Yixin Cao. A naive algorithm for feedback vertex set. In *Proceedings of the 1st Symposium on Simplicity in Algorithms (SOSA)*, volume 61 of *OASIcs*, pages 1:1–1:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.
- 31 Yixin Cao and Dániel Marx. Chordal editing is fixed-parameter tractable. *Algorithmica*, 75:118–137, 2016.
- 32 Arnaud Casteigts, Anne-Sophie Himmel, Hendrik Molter, and Philipp Zschoche. Finding temporal paths under waiting time constraints. *Algorithmica*, 83(9):2754–2802, 2021.
- 33 Mathieu Chapelle, Mathieu Liedloff, Ioan Todinca, and Yngve Villanger. Treewidth and pathwidth parameterized by the vertex cover number. *Discrete Applied Mathematics*, 216:114–129, 2017.
- 34 Juhi Chaudhary and Meirav Zehavi. Parameterized results on acyclic matchings with implications for related problems. *Journal of Computer and System Sciences*, 148:103599, 2025.
- 35 Ernest J. Cockayne, Seymour E. Goodman, and Stephen T. Hedetniemi. A linear algorithm for the domination number of a tree. *Information Processing Letters*, 4(2):41–44, 1975.
- 36 Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, 1990.
- 37 Bruno Courcelle and Joost Engelfriet. *Graph structure and monadic second-order logic: a language-theoretic approach*, volume 138. Cambridge University Press, 2012.
- 38 Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshantov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

- 39 Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michał Pilipczuk, Johan M.M. Van Rooij, and Jakub Onufry Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. *ACM Transactions on Algorithms (TALG)*, 18(2):1–31, 2022.
- 40 Marek Cygan, Marcin Pilipczuk, Michał Pilipczuk, and Jakub Onufry Wojtaszczyk. Subset feedback vertex set is fixed-parameter tractable. *SIAM Journal on Discrete Mathematics*, 27(1):290–309, 2013.
- 41 David E. Daykin and C.P. Ng. Algorithms for generalized stability numbers of tree graphs. *Journal of the Australian Mathematical Society*, 6(1):89–100, 1966.
- 42 Reinhard Diestel. *Graph Theory, 5th Edition*, volume 173 of *Graduate Texts in Mathematics*. Springer, 2016.
- 43 Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Springer, 2013.
- 44 Jessica A. Enright, Kitty Meeks, and Hendrik Molter. Counting temporal paths. In *Proceedings of the 40th International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 254 of *LIPICs*, pages 30:1–30:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.
- 45 Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*, volume XIV of *Texts in Theoretical Computer Science. An EATCS Series*. Springer, 2006.
- 46 Till Fluschnik, Hendrik Molter, Rolf Niedermeier, Malte Renken, and Philipp Zschoche. As time goes by: Reflections on treewidth for temporal graphs. In *Treewidth, Kernels, and Algorithms - Essays Dedicated to Hans L. Bodlaender on the Occasion of His 60th Birthday*, volume 12160 of *LNCs*, pages 49–77. Springer, 2020.
- 47 Fedor V. Fomin, Mathieu Liedloff, Pedro Montealegre, and Ioan Todinca. Algorithms parameterized by vertex cover and modular width, through potential maximal cliques. *Algorithmica*, 80:1146–1169, 2018.
- 48 Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, and Meirav Zehavi. *Kernelization: Theory of Parameterized Preprocessing*. Cambridge University Press, 2019.
- 49 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
- 50 Jiong Guo, Jens Gramm, Falk Hüffner, Rolf Niedermeier, and Sebastian Wernicke. Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization. *Journal of Computer and System Sciences*, 72(8):1386–1396, 2006.
- 51 Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8:171–186, 1976.
- 52 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k -SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.
- 53 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001.
- 54 Bart M.P. Jansen and Hans L. Bodlaender. Vertex cover kernelization revisited: Upper and lower bounds for a refined parameter. *Theory of Computing Systems*, 53(2):263–299, 2013.
- 55 Lawqueen Kanesh, Soumen Maity, Komal Muluk, and Saket Saurabh. Parameterized complexity of fair feedback vertex set problem. *Theoretical Computer Science*, 867:1–12, 2021.
- 56 Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer, 1972.
- 57 Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *LNCs*. Springer, 1994.
- 58 Tomasz Kociumaka and Marcin Pilipczuk. Faster deterministic feedback vertex set. *Information Processing Letters*, 114(10):556–560, 2014.
- 59 Tuukka Korhonen. A single-exponential time 2-approximation algorithm for treewidth. *SIAM Journal on Computing*, 0(0):FOCS21–174–FOCS21–194, 2023.
- 60 Tuukka Korhonen and Daniel Lokshtanov. An improved parameterized algorithm for treewidth. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC)*, pages 528–541. ACM, 2023.

- 61 Stefan Kratsch and Pascal Schweitzer. Isomorphism for graphs of bounded feedback vertex set number. In *Proceedings of the 12th Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 6139 of *LNCS*, pages 81–92. Springer, 2010.
- 62 Jens Lagergren and Stefan Arnborg. Finding minimal forbidden minors using a finite congruence. In *Proceedings of the 18th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 510 of *LNCS*, pages 532–543. Springer, 1991.
- 63 Jason Li and Jesper Nederlof. Detecting feedback vertex sets of size k in $O^*(2.7^k)$ time. *ACM Transactions on Algorithms (TALG)*, 18(4):1–26, 2022.
- 64 Daniel Lokshtanov, M. S. Ramanujan, and Saket Saurabh. Linear time parameterized algorithms for subset feedback vertex set. *ACM Transactions on Algorithms (TALG)*, 14(1):7:1–7:37, 2018.
- 65 Neeldhara Misra, Geevarghese Philip, Venkatesh Raman, and Saket Saurabh. On parameterized independent feedback vertex set. *Theoretical Computer Science*, 461:65–75, 2012.
- 66 Neeldhara Misra, Geevarghese Philip, Venkatesh Raman, Saket Saurabh, and Somnath Sikdar. Fpt algorithms for connected feedback vertex set. *Journal of Combinatorial Optimization*, 24:131–146, 2012.
- 67 Sandra Mitchell and Stephen Hedetniemi. Linear algorithms for edge-coloring trees and unicyclic graphs. *Information Processing Letters*, 9(3):110–112, 1979.
- 68 Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.
- 69 Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.
- 70 Neil Robertson and Paul D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986.
- 71 Neil Robertson and Paul D. Seymour. Graph minors. XIII. The disjoint paths problem. *Journal of combinatorial theory, Series B*, 63(1):65–110, 1995.
- 72 Meirav Zehavi. Tournament fixing parameterized by feedback vertex set number is FPT. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence (AAAI)*, volume 37:5, pages 5876–5883, 2023.