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 $2^{\mathcal{O}(\mathsf{fvn}(G))} \cdot n^{\mathcal{O}(1)}$, where $\mathsf{fvn}(G)$ 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 $2^{\mathcal{O}(\mathsf{tw}(G)^2)} \cdot n^4$ time, where $\mathsf{tw}(G)$ is the treewidth of G. Our algorithm improves upon this result on graphs G where $\mathsf{fvn}(G) \in o(\mathsf{tw}(G)^2)$. On a different note, since $\mathsf{fvn}(G)$ is an upper bound on $\mathsf{tw}(G)$, 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

Digital Object Identifier 10.4230/LIPIcs.ICALP.2025.120

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://arxiv.org/abs/2504.18302` [41]

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 [11, 13, 15, 16, 17, 36, 44]. 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 [51, Chapter 10], Flum and Grohe [35, Chapters 11 & 12], Downey and Fellows [34, Chapters 10–14], Cygan et al. [30, Chapter 7], and Fomin et al. [38, 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 [27, 32, 39, 50]. 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 [2, 4, 5, 8, 10, 22]. Maybe the most impactful such result is Courcelle's theorem [24, 28, 29], 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 [9] 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 [2, 14]. In the meantime, the concept of treewidth was rediscovered numerous times, for example by Halin [40] and, more prominently, by Robertson and Seymour [52, 53, 54] 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 [3], many early works on such algorithms assumed that a tree decomposition (or an equivalent structure) is given as part of the input [2, 5, 8]. 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 $\mathcal{O}(n^{k+2})$ [3]. Shortly afterwards, an algorithm with a running time in $f(k) \cdot n^2$ was given by Robertson and Seymour [54]. It consists of two steps. In a first step, a tree decomposition of with at most $4k + 3$ is computed in $O(3^{2k} \cdot n^2)$ 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 [21][2], and Lagergren and Arnborg [48] independently discovered algorithms with a running time in $2^{\mathcal{O}(k^3)} \cdot n$ for the second step. Furthermore, Bodlaender [12] 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 [24, 28, 29] 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 [47] 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 [47].[3]

[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 [21] 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) [42, 43], the existence of algorithms with

For many other graph problems, however, single exponential time algorithms in the treewidth of the input graph are known [5, 10, 18, 22, 31, 34, 35, 51]. If we want to use those algorithms to solve a graph problem, computing a tree decomposition of sufficiently 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 [54] in 1995, and since then many improvements both in terms of running time and approximation factor have been made [1, 6, 7, 19, 46]. The current best algorithm by Korhonen [46] 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 [46].

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 [20] 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 $2^{\mathcal{O}(\mathsf{fvn}(G))} \cdot n^{\mathcal{O}(1)}$ time. Here, $\mathsf{fvn}(G)$ 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 $2^{\mathcal{O}(\mathsf{vcn}(G))} \cdot n^{\mathcal{O}(1)}$ time (if it exists) [26, 37], where $\mathsf{vcn}(G)$ 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. [37], who showed that an optimal tree decomposition can be computed in $2^{\mathcal{O}(\mathsf{mw}(G))} \cdot n^{\mathcal{O}(1)}$, where $\mathsf{mw}(G)$ 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

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

where $\mathsf{tw}(G)$ denotes the treewidth of G. On a different note, since $\mathsf{fvn}(G)$ is an upper bound on $\mathsf{tw}(G)$, our algorithm can also be seen either as an important step towards a positive resolution of the open problem of finding an $2^{\mathcal{O}(\mathsf{tw}(G))} \cdot n^{\mathcal{O}(1)}$ 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 running time in $2^{o(k)} \cdot n^{\mathcal{O}(1)}$ can be ruled out [23].

Our algorithm constitutes a significant extension of a dynamic programming algorithm by Chapelle et al. [26] for computing optimal tree decompositions which runs in $2^{\mathcal{O}(\mathsf{vcn}(G))} \cdot n^{\mathcal{O}(1)}$ 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 $2.7^{\mathsf{fvn}(G)} \cdot n^{\mathcal{O}(1)}$ time (randomized) [49] or in $3.6^{\mathsf{fvn}(G)} \cdot n^{\mathcal{O}(1)}$ deterministic time [45]. As a second step, we apply the kernelization algorithm by Bodlaender, Jansen, and Kratsch [20] to reduce the number of vertices in the input graph to $\mathcal{O}(\mathsf{fvn}(G)^4)$. 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 [20]. 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 [20], Li and Nederlof [49], nor Kociumaka and Pilipczuk [45] 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 $2^{\mathcal{O}(\mathsf{fvn}(G))} \cdot n^2$ time [25].

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.

- Chapelle et al. [26] 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 [21], 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.

Due to space constraints, we defer considerable amounts of technical details to a full version of this work [41]. In Section 2 we give an overview of the main parts of the algorithm. Afterwards, we introduce the most important concepts and give some details on how the dynamic programming table looks like. The correctness proof and the running time analysis is deferred to the full version. Moreover, the full version contains expanded related work and additional illustrations.

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.

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 $\mathcal{T}$ is a very specific way. Given a bag X_t of some node in 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. [26]. One can show that the nodes in a tree decomposition that share the same S-trace partition the tree decomposition into path segments [26]. 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. [26] 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* [21]) 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.

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 $\mathsf{tw}(G) + 1$, 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 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.

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 vertice 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.

3 Main Concepts for the Algorithm

We use standard notation and terminology from graph theory [33]. 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* $\mathsf{tw}(G)$ of a graph G is the minimum width over all tree decomposition of G.

We present several concepts introduced by Chapelle et al. [26] that we also use for our algorithm in Section 3. In particular, we give the definitions of *traces*, *directed paths* thereof [26], and several additional concepts. The rest of the concepts (most of which are deferred to the full version [41]) are novel. 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. Finally, we introduce slim S-nice tree decompositions. This type of tree decompositions has crucial additional properties that we need to show that our algorithm is correct.

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 3.1** ([26]) (S-Trace). *Let $G = (V, E)$ be a graph and tet $S \subseteq V$. 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.*

Chapelle et al. [26] 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 3.2** ([26]). *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 3.2 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 3.3** (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 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.

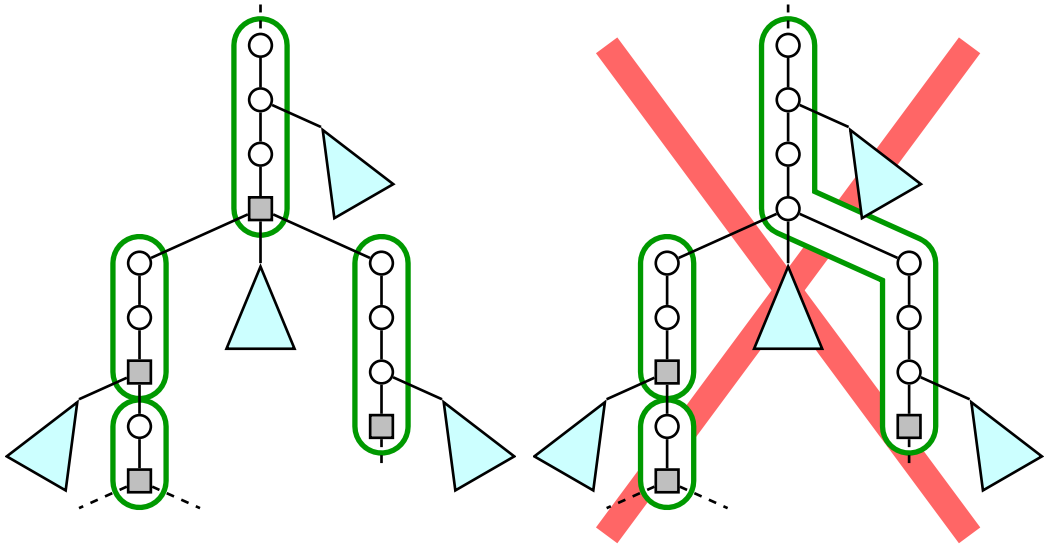
▶ **Definition 3.4** (S-Bottom Node). *Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a rooted tree decomposition of a graph $G = (V, E)$ and $S \subseteq V$. Let t be a node of $\mathcal{T}$. We say that t is an* S-bottom *node if there is an S-trace (L^S, X^S, R^S) such that $L^S \neq \emptyset$, 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.

We now define so-called *S-parent* and *S-children*. Informally, as 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 3.5** (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.*



(a) Illustration of an S-nice tree decomposition. **(b)** This cannot happen.

Figure 1 Illustrations for the concepts of S-traces (Definition 3.1) and their directed paths (Definition 3.3), S-bottom nodes (Definition 3.4), and S-nice tree decompositions (Definition 3.6). 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 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.

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 nice tree decompositions [21]) 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 $\mathsf{introduce}(v)$ for some $v \in X^S$, $\mathsf{forget}(v)$ for some $v \in S \setminus X^S$, or $\mathsf{join}(X^S, X_1^S, X_2^S, L_1^S, L_2^S)$ for some $X_1^S, X_2^S \subseteq X^S$ and $L_1^S, L_2^S \subseteq L^S$.

▶ **Definition 3.6** (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 $v \in S \setminus X_{t_1}$. Then we say that t admits the S-*operation $\mathsf{introduce}(v)$.
3. *Node t has exactly one S-child t_1. We have that $X_t^S = X_{t_1}^S \setminus \{v\}$ for some $v \in S \cap X_{t_1}$. Then we say that t admits the S-*operation $\mathsf{forget}(v)$.
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 $\mathsf{join}(X_t^S, X_{t_1}^S, X_{t_2}^S, L_{t_1}^S, L_{t_2}^S)$.

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 3.7.** *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.*

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 their parent.

Slim S-Nice Tree Decompositions. Now we introduce *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 3.8** (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, this most central definition to be presented in the extended abstract.

▶ **Definition 3.9** (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* 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) it 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 3.10.** *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.*

4 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, $V = \{v_1, v_2, \ldots, v_n\}$. 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 3.1) of a slim S-nice tree decomposition (Definitions 3.6 and 3.9) of G. 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 (which we also refer to as $t_{\max}$) 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 3.6. 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.

Notice that the bag $X_{\max}$ of node $t_{\max}$ 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 path of the S-trace. We give formal definitions of the extended S-operations in the next section.

Extended S-Operations. Now we describe how we extend the S-operations introduced in Definition 3.6. We first introduce a *big* version $\mathsf{bigjoin}$ 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.

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

In order to define which S-bottom nodes admit the above-described big S-operations, we introduce a function that "decodes" its bag. Let $\mathsf{bigbag} : 2^S \times \{0,1\}^{2\mathsf{fvn}(G)+1} \to 2^V$ be a function which takes as input 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. We additionally need to define a function that determines the bags of the children using the additional integers d_1 and d_2. Let $\mathsf{subbag} : 2^V \times \{0, \ldots, k+1\} \to 2^V$ 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 $\mathsf{bigjoin}$) and an additional integer that encodes which vertex is missing is the bag of a child (if the integer is zero, this will encode that the bag of the child is the same). Formally, $\mathsf{subbag}(X, 0) = X$ and $\mathsf{subbag}(X, i) = X \setminus \{v\}$, where v is at the ith ordinal position in X (recall that V is ordered in an arbitrary but fixed way).

▶ **Definition 4.1** (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* $\mathsf{join}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S)$. *Let t_1, t_2 be the two S-children of t and let t_0 be the parent of t. If $X_t = \mathsf{bigbag}(X_t^S, s)$ and for all $0 \leq i \leq 2$ we have $X_{t_i} = \mathsf{subbag}(X_t, d_i)$, then we say that node t admits S-operation* $\mathsf{bigjoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)$.

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 4.1 has, the function bigbag maps the X_t^S and the bit string s to that bag. Formally, we will show the following.

▶ **Lemma 4.2.** *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* $\mathsf{join}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S)$. *Then there exists $s \in \{0,1\}^{2\mathsf{fvn}(G)+1}$ and $d_0, d_1, d_2 \in \{0, \ldots, k+1\}$ such that node t admits S-operation* $\mathsf{bigjoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)$.

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 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.

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

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 $\mathsf{smallbag} : 2^S \times \mathbb{N} \to 2^V$ be a function which takes as input a subset of S and the number d that will guide the algorithm to a specific bag. 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 $\mathsf{nbag} : 2^V \times \mathbb{N} \times \{P, L, R\} \to 2^V$.

▶ **Definition 4.3** (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* $\mathsf{introduce}(v)$. *Let t_1 be the S-child of t and let t_0 be the parent of t. If $X_t = \mathsf{smallbag}(X_t^S, d)$, $X_{t_0} = \mathsf{nbag}(X_t, d, P)$, and $X_{t_1} = \mathsf{nbag}(X_t, d, L)$, then we say that node t admits S-operation* $\mathsf{smallintroduce}(v, d)$.

▶ **Definition 4.4** (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* $\mathsf{join}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S)$ *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 $X_t = \mathsf{smallbag}(X_t^S, d)$, $X_{t_0} = \mathsf{nbag}(X_t, d, P)$, $X_{t_1} = \mathsf{nbag}(X_t, d, L)$, and $X_{t_2} = \mathsf{nbag}(X_t, d, R)$, then we say that node t admits S-operation* $\mathsf{smalljoin}(X_t^S, X_1^S, X_2^S, L_1^S, L_2^S, d)$.

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 $\mathsf{forget}(v)$, 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 $\mathsf{forget}(v)$, 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 $\mathsf{forget}(v)$.

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

Similarly to the previously introduced extended S-operations, we define some auxiliary functions that serve analogous purposes. Let $\mathsf{smallbag}_{\mathsf{forget}} : 2^S \times \mathbb{N} \times \{\text{true}, \text{false}\} \times \Pi \to 2^V$, where Π denotes the set of all extended S-operations. Furthermore, let $\mathsf{nbag}_{\mathsf{forget}} : 2^V \times \mathbb{N} \times \{\text{true}, \text{false}\} \times \Pi \times \{P, L, R\} \to 2^V$.

▶ **Definition 4.5** ($\mathsf{extendedforget}$)**.** *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 $\mathsf{forget}(v)$. Let t' be the S-child of t. Let τ be an extended S-operation.*

- *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, $X_{t'} = \mathsf{smallbag}_{\mathsf{forget}}(X_t^S, d, \text{true}, \tau)$, $X_t = \mathsf{nbag}_{\mathsf{forget}}(X_t, d, \text{true}, \tau, P)$, and $X_{t''} = \mathsf{nbag}_{\mathsf{forget}}(X_t, d, \text{true}, \tau, L)$, then we say that node t admits S-operation $\mathsf{extendedforget}(v, d, \text{true}, \tau)$.*
- *Otherwise, if τ, $X_{t'} = \mathsf{smallbag}_{\mathsf{forget}}(X_t^S, d, \text{false}, \mathsf{void})$, $X_t = \mathsf{nbag}_{\mathsf{forget}}(X_t, d, \text{false}, \mathsf{void}, P)$, and $X_{t''} = \mathsf{nbag}_{\mathsf{forget}}(X_t, d, \text{false}, \mathsf{void}, L)$, then we say that node t admits S-operation $\mathsf{extendedforget}(v, d, \text{false}, \mathsf{void})$.*

Observe that if an S-bottom node admits an extended S-operation $\mathsf{extendedforget}(v, d, f, \tau)$, then $\tau \neq \mathsf{extendedforget}(v', d', f', \tau')$. This follows from the fact that the bag of an S-bottom node that admits S-operation $\mathsf{forget}(v)$ (and hence potentially an $\mathsf{extendedforget}$ S-operation) is never full.

▶ **Observation 4.6.** *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 $\mathsf{extendedforget}(v, d, f, \tau)$ for some $v \in S$, $d \in \{1, \ldots, (n+1)^3\}$, $f \in \{\text{true}, \text{false}\}$, and some S-operation τ. Then we have that $\tau \neq \mathsf{extendedforget}(v', d', f', \tau')$ for every $v' \in S$, $d' \in \{1, \ldots, (n+1)^3\}$, $f' \in \{\text{true}, \text{false}\}$, and every extended S-operation τ'.*

Finally, we remark that for the small S-operations, we do not have an analog to Lemma 4.2, 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 the full version [41].

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 4.7** (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 $\tau_+ = \mathsf{void}$, then the root of $\mathcal{T}$ has S-trace (L^S, X^S, R^S).*
- *If $L^S = \emptyset$, then $\tau_- = \mathsf{void}$ 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 $\tau_+ \neq \mathsf{void}$, 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 4.7 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 $\tau_- = \mathsf{void}$ 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 $\mathsf{PTW} : \mathcal{S} \to \{\text{true}, \text{false}\}$ (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 $\mathsf{LTW} : \mathcal{S} \to \{\text{true}, \text{false}\}$ (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 $\mathsf{PTW} : \mathcal{S} \to \{\text{true}, \text{false}\}$ will be somewhat similar to the one by Chapelle et al. [26], however, our definition of $\mathsf{LTW} : \mathcal{S} \to \{\text{true}, \text{false}\}$ will be significantly more sophisticated than the one used by Chapelle et al. [26].

Given a state $(\tau_-, L^S, X^S, R^S, \tau_+)$ of the dynamic program, by Definition 3.9 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. 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 $\tau_- = \mathsf{smallintroduce}(v, d)$ or $\tau_- = \mathsf{extendedforget}(v, d, f, \tau)$, the set $\Psi(\phi)$ of *preceding states* of ϕ is defined as follows.

- If $\tau_- = \mathsf{smallintroduce}(v, d)$, then
  $\Psi(\phi) = \{(\tau'_-, L^S, X^S \setminus \{v\}, R^S \cup \{v\}, \mathsf{smallintroduce}(v, d)) \mid \tau'_- \in \Pi\}$.
- If $\tau_- = \mathsf{extendedforget}(v, d, f, \tau)$, then
  $\Psi(\phi) = \{(\tau, L^S \setminus \{v\}, X^S \cup \{v\}, R^S, \mathsf{extendedforget}(v, d, f, \tau))\}$ if $f = \text{true}$, and
  $\Psi(\phi) = \{(\tau'_-, L^S \setminus \{v\}, X^S \cup \{v\}, R^S, \mathsf{extendedforget}(v, d, f, \tau)) \mid \tau'_- \in \Pi\}$ otherwise.

If $\tau_- = \mathsf{smalljoin}(X^S, X_1^S, X_2^S, L_1^S, L_2^S, d)$ or $\tau_- = \mathsf{bigjoin}(X^S, X_1^S, X_2^S, L_1^S, L_2^S, s, d_0, d_1, d_2)$, then the sets $\Psi_1(\phi)$ and $\Psi_2(\phi)$ of *preceding states* of ϕ are defined in an analogous way. We give all details in the full version [41].

We can show 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 $\mathsf{PTW}(\phi)$ for some state ϕ. However, the above definition 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 $\mathsf{legal}_1 : \mathcal{S} \times \mathcal{S} \to \{\text{true}, \text{false}\}$ and $\mathsf{legal}_2 : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \to \{\text{true}, \text{false}\}$ 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. 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. Formally, we define the dynamic programming table as follows.

▶ **Definition 4.8** (Dynamic Programming Table PTW). *The dynamic programming table is a recursive function* $\mathsf{PTW} : \mathcal{S} \to \{\text{true}, \text{false}\}$ *which is defined as follows. Let* $\mathsf{LTW} : \mathcal{S} \to \{\text{true}, \text{false}\}$*, let* $\mathsf{legal}_1 : \mathcal{S} \times \mathcal{S} \to \{\text{true}, \text{false}\}$*, and let* $\mathsf{legal}_2 : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \to \{\text{true}, \text{false}\}$*. 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* $\tau_- = \mathsf{void}$*, then*

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

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

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

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

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

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.

- $X^\phi_{\max}$ 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^ϕ_p 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 $X^\phi_p = \emptyset$.
- $X^\phi_{\min}$ 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^ϕ_{path} 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 the third condition of the definition of tree decompositions (see beginning of Section 3), vertex v needs to be in the bags of both $t_{\max}$ and $t_{\min}$ and also every bag between them. The set X^ϕ_{path}, 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^ϕ_{path}, 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^ϕ_c be the union of the bags of the S-children of $t_{\min}$ and the empty set if $L^S = \emptyset$. Computing $X^\phi_{\max}$, X^ϕ_p, $X^\phi_{\min}$, and X^ϕ_c for a given state ϕ is the main challenge here. We give a formal definition of these three sets and algorithms to compute them in the full version [41]. Given $X^\phi_{\max}$, X^ϕ_p, and $X^\phi_{\min}$, we define X^ϕ_{path} as follows.

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

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

▶ **Definition 4.9** (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 $u, v \in X^\phi_{\max}$ with $\{u, v\} \notin E$, all pairs of vertices $u, v \in X^\phi_p$ with $\{u, v\} \notin E$, and all pairs of vertices $u, v \in X^\phi_{\min}$ with $\{u, v\} \notin E$. Then*

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

5 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 $2^{\mathcal{O}(\mathsf{fvn}(G))} \cdot n^{\mathcal{O}(1)}$ time. This improves the previously known result that a minimum tree decomposition for a graph G can be computed in $2^{\mathcal{O}(\mathsf{vcn}(G))} \cdot n^{\mathcal{O}(1)}$ time [26, 37]. We believe that this can also be seen either as a critical step towards a positive resolution of the open problem of finding a $2^{\mathcal{O}(\mathsf{tw}(G))} \cdot n^{\mathcal{O}(1)}$ 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.

References

1. Eyal Amir. Approximation algorithms for treewidth. *Algorithmica*, 56:448–479, 2010. doi:10.1007/S00453-008-9180-4.
2. Stefan Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability—a survey. *BIT Numerical Mathematics*, 25(1):1–23, 1985.
3. 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.
4. Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12(2):308–340, 1991. doi:10.1016/0196-6774(91)90006-K.
5. 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. doi:10.1016/0166-218X(89)90031-0.
6. 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.
7. Mahdi Belbasi and Martin Fürer. An improvement of Reed's treewidth approximation. *Journal of Graph Algorithms and Applications*, 26(2):257–282, 2022. doi:10.7155/JGAA.00593.
8. 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. doi:10.1016/0196-6774(87)90039-3.

9 Umberto Bertelè and Francesco Brioschi. *Nonserial dynamic programming*. Academic Press, Inc., 1972.

10 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. doi:10.1007/3-540-19488-6_110.

11 Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11(1-2):1–21, 1993. URL: https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/3417.

12 Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996. doi:10.1137/S0097539793251219.

13 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. doi:10.1007/BFB0029946.

14 Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209(1-2):1–45, 1998. doi:10.1016/S0304-3975(97)00228-4.

15 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. doi:10.1007/978-3-540-30577-4_1.

16 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. doi:10.1007/11917496_1.

17 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. doi:10.1007/978-3-540-72951-8_3.

18 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. doi:10.1016/J.IC.2014.12.008.

19 Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michał Pilipczuk. A $c^k n$ 5-approximation algorithm for treewidth. *SIAM Journal on Computing*, 45(2):317–378, 2016. doi:10.1137/130947374.

20 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. doi:10.1137/120903518.

21 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. doi:10.1006/JAGM.1996.0049.

22 Hans L. Bodlaender and Arie M.C.A. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2008. doi:10.1093/COMJNL/BXM037.

23 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). doi:10.48550/arXiv.2406.11628.

24 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. doi:10.1007/BF01758777.

25 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. doi:10.4230/OASICS.SOSA.2018.1.

26 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. doi:10.1016/J.DAM.2014.12.012.

27 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. doi:10.1016/0020-0190(75)90011-3.

28 Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, 1990. doi:10.1016/0890-5401(90)90043-H.

29 Bruno Courcelle and Joost Engelfriet. *Graph structure and monadic second-order logic: a language-theoretic approach*, volume 138. Cambridge University Press, 2012.

30 Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015. doi:10.1007/978-3-319-21275-3.

31 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. doi:10.1145/3506707.

32 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.

33 Reinhard Diestel. *Graph Theory, 5th Edition*, volume 173 of *Graduate Texts in Mathematics*. Springer, 2016.

34 Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Springer, 2013. doi:10.1007/978-1-4471-5559-1.

35 Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*, volume XIV of *Texts in Theoretical Computer Science. An EATCS Series*. Springer, 2006. doi:10.1007/3-540-29953-X.

36 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. doi:10.1007/978-3-030-42071-0_6.

37 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. doi:10.1007/S00453-017-0297-1.

38 Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, and Meirav Zehavi. *Kernelization: Theory of Parameterized Preprocessing*. Cambridge University Press, 2019.

39 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

40 Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8:171–186, 1976.

41 Amit Zivan Hendrik Molter, Meirav Zehavi. Treewidth parameterized by feedback vertex number. *CoRR*, abs/2504.18302, 2025. URL: https://arxiv.org/abs/2504.18302.

42 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001. doi:10.1006/JCSS.2000.1727.

43 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001. doi:10.1006/JCSS.2001.1774.

44 Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *LNCS*. Springer, 1994. doi:10.1007/BFB0045375.

45 Tomasz Kociumaka and Marcin Pilipczuk. Faster deterministic feedback vertex set. *Information Processing Letters*, 114(10):556–560, 2014. doi:10.1016/J.IPL.2014.05.001.

46 Tuukka Korhonen. A single-exponential time 2-approximation algorithm for treewidth. *SIAM Journal on Computing*, 0(0):FOCS21–174–FOCS21–194, 2023.

47 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. doi:10.1145/3564246.3585245.

48 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. doi:10.1007/3-540-54233-7_161.

49 Jason Li and Jesper Nederlof. Detecting feedback vertex sets of size k in $O^\star(2.7^k)$ time. *ACM Transactions on Algorithms (TALG)*, 18(4):1–26, 2022. doi:10.1145/3504027.

50 Sandra Mitchell and Stephen Hedetniemi. Linear algorithms for edge-coloring trees and unicyclic graphs. *Information Processing Letters*, 9(3):110–112, 1979. doi:10.1016/0020-0190(79)90049-8.

51 Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006. doi:10.1093/ACPROF:OSO/9780198566076.001.0001.

52 Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984. doi:10.1016/0095-8956(84)90013-3.

53 Neil Robertson and Paul D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986. doi:10.1016/0196-6774(86)90023-4.

54 Neil Robertson and Paul D. Seymour. Graph minors. XIII. The disjoint paths problem. *Journal of combinatorial theory, Series B*, 63(1):65–110, 1995. doi:10.1006/JCTB.1995.1006.