A $c^k n$ 5-Approximation Algorithm for Treewidth

Hans L. Bodlaender* Pål Grønås Drange† Markus S. Dregi† Fedor V. Fomin†
Daniel Lokshtanov† Michał Pilipczuk†

July 4, 2013

Abstract

We give an algorithm that for an input n-vertex graph G and integer $k > 0$, in time $O(c^k n)$ either outputs that the treewidth of G is larger than k, or gives a tree decomposition of G of width at most $5k + 4$. This is the first algorithm providing a constant factor approximation for treewidth which runs in time single-exponential in k and linear in n.

Treewidth based computations are subroutines of numerous algorithms. Our algorithm can be used to speed up many such algorithms to work in time which is single-exponential in the treewidth and linear in the input size.

1 Introduction

Since its invention in the 1980s, the notion of treewidth has come to play a central role in an enormous number of fields, ranging from very deep structural theories to highly applied areas. An important (but not the only) reason for the impact of the notion is that many graph problems that are intractable on general graphs become efficiently solvable when the input is a graph of bounded treewidth. In most cases, the first step of an algorithm is to find a tree decomposition of small width and the second step is to perform a dynamic programming procedure on the tree decomposition.

In particular, if a graph on n vertices is given together with a tree decomposition of width k, many problems can be solved by dynamic programming in time $2^{O(k)}n$, i.e., single-exponential in the treewidth and linear in n. Many of the problems admitting such algorithms have been known for over thirty years [6] but new algorithmic techniques on graphs of bounded treewidth [11, 21] as well as new problems motivated by various applications (just a few of many examples are [1, 27, 31, 38]) continue to be discovered. While a reasonably good tree decomposition can be derived from the properties of the problem sometimes, in most of the applications, the computation of a good tree decomposition is a challenge. Hence the natural question here is what can be done when no tree decomposition is given. In other words, is there an algorithm that for a given graph G and integer k, in time $2^{O(k)}n$ either correctly reports that the treewidth of G is at least k, or finds an optimal solution to our favorite problem (finds a maximum independent set, computes the chromatic number, decides if G is Hamiltonian, etc.)? To answer this question it would be sufficient to have an algorithm that in time $2^{O(k)}n$ either reports correctly that the treewidth of G is more that k, or construct a tree decomposition of width at most ck for some constant c.

*Department of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, the Netherlands. h.l.bodlaender@uu.nl

†Department of Informatics, Univerity of Bergen, Norway. {Pal.Drange, Markus.Dregi, fomin, Daniel.Lokshtanov, Michal.Pilipczuk}@ii.uib.no

However, the lack of such algorithms has been a bottleneck, both in theory and in practical applications of the treewidth concept. The existing approximation algorithms give us the choice of running times of the form $2^{O(k)}n^2$, $2^{O(k\log k)}n\log n$, or $k^{O(k^3)}n$, see Table 1. Remarkably, the newest of these current record holders is now almost 20 years old. This "newest record holder" is the linear time algorithm of Bodlaender [7, 9] that given a graph G, decides if the treewidth of G is at most k, and if so, gives a tree decomposition of width at most k in $O(k^{O(k^3)}n)$ time. The improvement by Perković and Reed [36] is only a factor polynomial in k faster (but also, if the treewidth is larger than k, it gives a subgraph of treewidth more than k with a tree decomposition of width at most k, leading to an $O(n^2)$ algorithm for the fundamental disjoint paths problem). Recently, a version running in logarithmic space was found by Elberfeld et al. [24], but its running time is not linear.

Reference	Approximation	$f(k)$	$g(n)$
Arnborg et al. [4]	exact	$O(1)$	$O(n^{k+2})$
Robertson & Seymour [40]	$4k+3$	$O(3^{3k})$	n^2
Lagergren [32]	$8k+7$	$2^{O(k\log k)}$	$n\log^2 n$
Reed [37]	$8k+O(1)$[1]	$2^{O(k\log k)}$	$n\log n$
Bodlaender [9]	exact	$k^{O(k^3)}$	n
Amir [3]	$4.5k$	$O(2^{3k}k^{3/2})$	n^2
Amir [3]	$(3+2/3)k$	$O(2^{3.6982k}k^3)$	n^2
Amir [3]	$O(k\log k)$	$O(k\log k)$	n^4
Feige et al. [25]	$O(k\cdot\sqrt{\log k})$	$O(1)$	$n^{O(1)}$
This paper	$3k+4$	$2^{O(k)}$	$n\log n$
This paper	$5k+4$	$2^{O(k)}$	n

Table 1: Overview of treewidth algorithms. Here k is the treewidth and n is the number of vertices of an input graph G. Each of the algorithms outputs in time $f(k)\cdot g(n)$ a decomposition of width given in the Approximation column.

In this paper, we give the first constant factor approximation algorithm for the treewidth graph such that its running time is single exponential in treewidth and linear in the size of the input graph. Our main result is the following theorem.

Theorem I. *There exists an algorithm, that given an n-vertex graph G and an integer k, in time $2^{O(k)}n$ either outputs that the treewidth of G is larger than k, or constructs a tree decomposition of G of width at most $5k+4$.*

Of independent interest are a number of techniques that we use to obtain the result and the intermediate result of an algorithm that either tells that the treewidth is larger than k or outputs a tree decomposition of width at most $3k+4$ in time $2^{O(k)}n\log n$.

Related results and techniques. The basic shape of our algorithm is along the same lines as most of the treewidth approximation algorithms [3, 13, 25, 32, 37, 40], i.e., a specific scheme of repeatedly finding separators. If we ask for polynomial time approximation algorithms for treewidth, the currently best result is that of [25] that gives in polynomial (but not linear) time a tree decomposition of width $O(k\cdot\sqrt{\log k})$ where k is the treewidth of the graph. Their work also gives a polynomial time approximation algorithm with ratio $O(|V_H|^2)$ for H-minor free graphs.

[1] Reed [37] does not state the approximation ratio of his algorithm explicitly. However, a careful analysis of his manuscript show that the algorithm can be implemented to give a tree decomposition of width at most $8k+O(1)$.

By Austrin et al. [5], assuming the Small Set Expansion Conjecture, there is no polynomial time approximation algorithm for treewidth with a constant performance ratio.

An important element in our algorithms is the use of a data structure that allows to perform various queries in time $O(c^k \log n)$ each, for some constant c. This data structure is obtained by adding various new techniques to old ideas from the area of dynamic algorithms for graphs of bounded treewidth [8, 17, 18, 19, 29].

A central element in the data structure is a tree decomposition of the input graph of bounded (but too large) width such that the tree used in the tree decomposition is binary and of logarithmic depth. To obtain this tree decomposition, we combine the following techniques: following the scheme of the exact linear time algorithms [9, 36], but replacing the call to the dynamic programming algorithm of Bodlaender and Kloks [15] by a recursive call to our algorithm, we obtain a tree decomposition of G of width at most $10k+9$ (or $6k+9$, in the case of the $O(c^k n \log n)$ algorithm of Section 4.) Then, we use a result by Bodlaender and Hagerup [14] that this tree decomposition can be turned into a tree decomposition with a logarithmic depth binary tree in linear time, or more precisely, in $O(\log n)$ time and $O(n)$ operations on an EREW PRAM. The cost of this transformation is increasing the width of the decomposition roughly three times. The latter result is an application of classic results from parallel computing for solving problems on trees, in particular Miller-Reif tree contraction [34, 35].

Using the data structure to "implement" the algorithm of Robertson and Seymour [40] already gives an $O(c^k n \log n)$ 3-approximation for treewidth (Section 4). Additional techniques are needed to speed this algorithm up. We build a series of algorithms, with running times of the forms $O(c^k n \log\log n)$, $O(c^k n \log\log\log n)$, ..., etc. Each algorithm "implements" Reed's algorithm [37], but with a different procedure to find balanced separators of the subgraph at hand, and stops when the subgraph at hand has size $O(\log n)$. In the latter case, we call the previous algorithm of the series on this subgraph.

Finally, to obtain a linear time algorithm, we consider two cases, one case for when n is "small" (with respect to k), and one case when n is "large", where we consider n to be small if

$$n \leq 2^{2^{c_0 k^3}}, \text{ for some constant } c_0.$$

For small values of n, we apply the $O(c^k n \log^{(2)} n)$ algorithm from Section 5. This will yield a linear running time in n since $\log^{(2)} n \leq k$. For larger values of n, we show that the linear time algorithms of [9] or [36] can be implemented in truly linear time, without any overhead depending on k. This seemingly surprising result can be roughly obtained as follows. We explicitly construct the finite state tree automaton of the dynamic programming algorithm in sublinear time before processing the graph, and then view the dynamic programming routine as a run of the automaton, where productions are implemented as constant time table lookups. Viewing a dynamic programming algorithm on a tree decomposition as a finite state automaton traces back to early work by Fellows and Langston [26], see also [2]. Our algorithm assumes the RAM model of computation [41], and the only aspect of the RAM model which is exploited by our algorithm is the ability to look up an entry in a table in constant time, independently of the size of the table. This capability is crucially used in almost every linear time graph algorithm including breadth first search and depth first search.

Overview of the paper. In Section 2 we give the outline of the main algorithms, focusing on explaining main intuitions rather than formal details of the proofs. Some concluding remarks and open questions are made in Section 3. Sections 4, 5 and 6 give the formal descriptions of the main algorithms: first the $O(c^k n \log n)$ algorithm, then the series of $O(c^k n \log^{(\alpha)} n)$ algorithms, before describing the $O(c^k n)$ algorithm. Each of the algorithms described in these sections use queries to a data structure which is described in Section 7.

Notation. We give some basic definitions and notation, used throughout the paper. For $\alpha \in \mathbb{N}$, the function $\log^{(\alpha)} n$ is defined as

$$\log^{\alpha} n = \begin{cases} \log n & \text{if } \alpha = 1 \\ \log(\log^{\alpha-1} n) & \text{otherwise.} \end{cases}$$

For the presentation of our results, it is more convenient when we regard tree decompositions as rooted. This yields the following definition of tree decompositions.

Definition 1.1. A *tree decomposition* of a graph $G = (V, E)$ is a pair $\mathcal{T} = (\{B_i \mid i \in I\}, T = (I, F))$ where $T = (I, F)$ is a rooted tree, and $\{B_i \mid i \in I\}$ is a family of subsets of V, such that

- $\bigcup_{i \in I} B_i = V$,
- for all $\{v, w\} \in E$, there exists an $i \in I$ with $v, w \in B_i$, and
- for all $v \in V$, $I_v = \{i \in I \mid v \in B_i\}$ induces a subtree of T.

The *width* of $\mathcal{T} = (\{B_i \mid i \in I\}, T = (I, F))$, denoted $\mathrm{w}(\mathcal{T})$ is $\max_{i \in I} |B_i| - 1$. The *treewidth* of a graph G, denoted by $\mathrm{tw}(G)$, is the minimum width of a tree decomposition of G.

For each $v \in V$, the tree induced by I_v is denoted by T_v; the root of this tree, i.e., the node in the tree with smallest distance to the root of T is denoted by r_v.

The sets B_i are called the *bags* of the decomposition. For each node $i \in I$, we define V_i to be the union of bags contained in the subtree of T rooted in i, including B_i. Moreover, we denote $W_i = V_i \setminus B_i$ and $G_i = G[V_i]$, $H_i = G[W_i]$. Note that by the definition of tree decomposition, B_i separates W_i from $V \setminus V_i$.

2 Proof outline

Our algorithm builds on the constant factor approximation algorithm for treewidth described in Graph Minors XIII [40] with running time $O(c^k n^2)$. We start with a brief explanation of a variant of this algorithm.

2.1 The $O(3^{3k} n^2)$ time 4-approximation algorithm from Graph Minors XIII

The engine behind the algorithm is a lemma that states that graphs of treewidth k have balanced separators of size $k + 1$. In particular, for any way to assign non-negative weights to the vertices there exists a set X of size at most $k + 1$ such that the total weight of any connected component of $G \setminus X$ is at most half of the total weight of G. We will use the variant of the lemma where some vertices have weight 1 and some have weight 0.

Lemma 2.1 (Graph Minors II [39])**.** *If $\mathrm{tw}(G) \leq k$ and $S \subseteq V(G)$, then there exists $X \subseteq V(G)$ with $|X| \leq k + 1$ such that every component of $G \setminus X$ has at most $\frac{1}{2}|S|$ vertices which are in S.*

We note that the original version of this lemma is somewhat stronger: it gives a bound $\frac{1}{2}|S \setminus X|$ instead of $\frac{1}{2}|S|$. However, we do not need this stronger version and we find it more convenient to work with the weaker. The set X with properties ensured by Lemma 2.1 will be called a *balanced S-separator*, or a *$\frac{1}{2}$-balanced S-separator*. More generally, for a *β-balanced S-separator* X, every connected component of $G \setminus X$ contains at most $\beta|S|$ vertices of S. If we omit the set S, i.e., talk about separators instead of S-separators, we mean $S = V(G)$ and balanced separators of the entire vertex set.

The proof of Lemma 2.1 is not too hard; start with a tree decomposition of G with width at most k and orient every edge of the decomposition tree towards the side which contains the larger part of the set S. Two edges of the decomposition can not point "in different directions", since then there would be disjoint parts of the tree, both containing more than half of S. Thus there has to be a node in the decomposition tree such that all edges of the decomposition are oriented towards it. The bag of the decomposition corresponding to this node is exactly the set X of at most $k+1$ vertices whose deletion leaves connected components with at most $\frac{1}{2}|S|$ vertices of S each.

The proof of Lemma 2.1 is constructive if one has access to a tree decomposition of G of width less than k. The algorithm does not have such a decomposition at hand, after all we are trying to compute a decomposition of G of small width. Thus we have to settle for the following algorithmic variant of Robertson and Seymour [39].

Lemma 2.2 ([40])**.** *There is an algorithm that given a graph G, a set S and a $k \in \mathbb{N}$ either concludes that $\mathrm{tw}(G) > k$ or outputs a set X of size at most $k+1$ such that every component of $G \setminus X$ has at most $\frac{2}{3}|S|$ vertices which are in S and runs in time $O(3^{|S|}k^{O(1)}(n+m))$.*

Proof sketch. By Lemma 2.1 there exists a set X' of size at most $k+1$ such that every component of $G \setminus X'$ has at most $\frac{1}{2}|S|$ vertices which are in S. A simple packing argument shows that the components can be assigned to left or right such that at most $\frac{2}{3}|S|$ vertices of S go left and at most $\frac{2}{3}|S|$ go right. Let S_X be $S \cap X'$ and let S_L and S_R be the vertices of S that were put left and right respectively. By trying all partitions of S in three parts the algorithm correctly guesses S_X, S_L and S_R. Now X separates S_L from S_R and so the minimum vertex cut between S_L and S_R in $G \setminus S_X$ is at most $|X \setminus S_X| \leq (k+1) - |S_X|$. The algorithm finds using max-flow a set Z of size at most $(k+1) - |S_X|$ that separates S_L from S_R in $G \setminus S_X$. Since we are only interested in a set Z of size at most $k - |S_X|$ one can run max-flow in time $O((n+m)k^{O(1)})$. Having found S_L, S_R, S_X and Z the algorithm sets $X = S_X \cup Z$, L to contain all components of $G \setminus X$ that contain vertices of S_L and R to contain all other vertices. Since every component C of $G \setminus X$ is fully contained in L or R, the bound on $|C \cap S|$ follows.

If no partition of S into S_L, S_R, S_X yielded a cutset Z of size $\leq (k+1) - |S_X|$, this means that $\mathrm{tw}(G) > k$, which the algorithm reports. □

The algorithm takes as input G, S, k, with S a set with at most $3k+3$ vertices, and either concludes that the treewidth of G is larger than k or finds a tree decomposition of width at most $4k+3$ such that the top bag of the decomposition contains S.

On input G, S, k the algorithm starts by ensuring that $|S| = 3k+3$. If $|S| < 3k+3$ the algorithm just adds arbitrary vertices to S until equality is obtained. Then the algorithm applies Lemma 2.2 and finds a set X of size at most $k+1$ such that each component C_i of $G \setminus X$ satisfies $|C_i \cap S| \leq \frac{2|S|}{3} \leq 2k+2$. Thus for each C_i we have $|(S \cap C_i) \cup X| \leq 3k+3$. For each component C_i of $G \setminus X$ the algorithm runs itself recursively on $(G[C_i \cup X], (S \cap C_i) \cup X, k)$.

If either of the recursive calls returns that the treewidth is more than k then the treewidth of G is more than k as well. Otherwise we have for every component C_i a tree decomposition of $G[C_i \cup X]$ of width at most $4k+3$ such that the top bag contains $(S \cap C_i) \cup X$. To make a tree decomposition of G we make a new root node with bag $X \cup S$, and connect this bag to the roots of the tree decompositions of $G[C_i \cup X]$ for each component C_i. It is easy to verify that this is indeed a tree decomposition of G. The top bag contains S, and the size of the top bag is at most $|S| + |X| \leq 4k+4$, and so the width of the decomposition is at most $4k+3$ as claimed.

The running time of the algorithm is governed by the recurrence

$$T(n,k) \leq O(3^{|S|}k^{O(1)}(n+m)) + \sum_{C_i} T(|C_i \cup X|, k) \tag{1}$$

which solves to $T(n,k) \leq (3^{3k}k^{O(1)}n(n+m))$ since $|S| = 3k+3$ and there always are at least two non-empty components of $G \setminus X$. Finally, we use the following observation about the number of edges in a graph of treewidth k.

Lemma 2.3 ([12]). *Let $G = (V,E)$ be a graph with treewidth at most k. Then $|E| \leq |V|k$.*

Thus if $|E| > nk$ the algorithm can safely output that $\mathrm{tw}(G) > k$. After this, running the algorithm above takes time $O(3^{3k}k^{O(1)}n(n+m)) = O(3^{3k}k^{O(1)}n^2)$.

2.2 The $O(k^{O(k)}n \log n)$ time approximation algorithm of Reed

Reed [37] observed that the running time of the algorithm of Robertson and Seymour [40] can be sped up from $O(n^2)$ for fixed k to $O(n \log n)$ for fixed k, at the cost of a worse (but still constant) approximation ratio, and a $k^{O(k)}$ dependence on k in the running time, rather than the 3^{3k} factor in the algorithm of Robertson and Seymour. We remark here that Reed [37] never states explicitly the dependence on k of his algorithm, but a careful analysis shows that this dependence is in fact of order $k^{O(k)}$. The main idea of this algorithm is that the recurrence in Equation 1 only solves to $O(n^2)$ for fixed k if one of the components of $G \setminus X$ contains almost all of the vertices of G. If one could ensure that each component C_i of $G \setminus X$ had at most $c \cdot n$ vertices for some fixed $c < 1$, the recurrence in Equation 1 solves to $O(n \log n)$ for fixed k. To see that this is true we simply consider the recursion tree. The total amount of work done at any level of the recursion tree is $O(n)$ for a fixed k. Since the size of the components considered at one level is always a constant factor smaller than the size of the components considered in the previous level, the number of levels is only $O(\log n)$ and we have $O(n \log n)$ work in total.

By using Lemma 2.1 with $S = V(G)$ we see that if G has treewidth $\leq k$, then there is a set X of size at most $k+1$ such that each component of $G \setminus X$ has size at most $\frac{n}{2}$. Unfortunately if we try to apply Lemma 2.2 to *find* an X which splits G in a balanced way using $S = V(G)$, the algorithm of Lemma 2.2 takes time $O(3^{|S|}k^{O(1)}(n+m)) = O(3^n n^{O(1)})$, which is exponential in n. Reed [37] gave an algorithmic variant of Lemma 2.1 especially tailored for the case where $S = V(G)$.

Lemma 2.4 ([37]). *There is an algorithm that given G and k, runs in time $O(k^{O(k)}n)$ and either concludes that $\mathrm{tw}(G) > k$ or outputs a set X of size at most $k+1$ such that every component of $G \setminus X$ has at most $\frac{3}{4}|S|$ vertices which are in S.*

Let us remark that Lemma 2.4 as stated here is never explicitly proved in [37], but it follows easily from the arguments given there.

Having Lemmata 2.2 and 2.4 at hand, we show how to obtain an 8-approximation of the treewidth of G in time $O(k^{O(k)}n \log n)$. The algorithm takes as input G, S, k, where S is a set of at most $6k+6$ vertices, and either concludes that $\mathrm{tw}(G) > k$, or finds a tree decomposition of width at most $8k+7$ where the top bag of the decomposition contains S.

On input G, S, k the algorithm starts by ensuring that $|S| = 6k+6$. If $|S| < 6k+6$ the algorithm just adds vertices to S until equality is obtained. Then the algorithm applies Lemma 2.2 and finds a set X_1 of size at most $k+1$ such that each component C_i of $G \setminus X_1$ satisfies $|C_i \cap S| \leq \frac{2}{3}|S| \leq 4k+4$. Now the algorithm applies Lemma 2.4 and finds a set X_2 of size at most $k+1$ such that each component C_i of $G \setminus X_2$ satisfies $|C_i| \leq \frac{3}{4}|V(G)| \leq \frac{3}{4}n$. Set $X = X_1 \cup X_2$. For each component C_i of $G \setminus X$ we have that $|(S \cap C_i) \cup X| \leq 6k+6$. For each component C_i of $G \setminus X$ the algorithm runs itself recursively on $(G[C_i \cup X], (S \cap C_i) \cup X, k)$.

If either of the recursive calls returns that the treewidth is more than k then the treewidth of G is more than k as well. Otherwise we have for every component C_i a tree decomposition of $G[C_i \cup X]$ of width at most $8k+7$ such that the top bag contains $(S \cap C_i) \cup X$. Similarly

as before, to make a tree decomposition of G we make a new root node with bag $X \cup S$, and connect this bag to the roots of the tree decompositions of $G[C_i \cup X]$ for each component C_i. It is easy to verify that this is indeed a tree decomposition of G. The top bag contains S, and the size of the top bag is at most $|S| + |X| \leq |S| + |X_1| + |X_2| \leq 6k + 6 + 2k + 2 = 8k + 8$, and the width of the decomposition is at most $8k + 7$ as claimed.

The running time of the algorithm is governed by the recurrence

$$T(n,k) \leq O\left(k^{O(k)}(n+m)\right) + \sum_{C_i} T(|C_i \cup X|, k) \tag{2}$$

which solves to $T(n,k) \leq O(k^{O(k)}(n+m)\log n)$ since each C_i has size at most $\frac{3}{4}n$. By Lemma 2.3 we have $m \leq kn$ and so the running time of the algorithm is upper bounded by $O(k^{O(k)} n \log n)$.

2.3 A new $O(c^k n \log n)$ time 3-approximation algorithm

The goal of this section is to sketch a proof of the following theorem. A full proof of Theorem II can be found in Section 4.

Theorem II. *There exists an algorithm which given a graph G and an integer k, either computes a tree decomposition of G of width at most $3k+4$ or correctly concludes that $\mathrm{tw}(G) > k$, in time $O(c^k \cdot n \log n)$ for some $c \in \mathbb{N}$.*

The algorithm employs the same recursive compression scheme which is used in Bodlaender's linear time algorithm [7, 9] and the algorithm of Perković and Reed [36]. The idea is to solve the problem recursively on a smaller instance, expand the obtained tree decomposition of the smaller graph to a "good, but not quite good enough" tree decomposition of the instance in question, and then use this tree decomposition to either conclude that $\mathrm{tw}(G) > k$ or find a decomposition of G which is good enough. A central concept in the recursive approach of Bodlaender [9] is the definition of an improved graph:

Definition 2.5. Given a graph $G = (V, E)$ and an integer k, the *improved* graph of G, denoted G_I, is obtained by adding an edge between each pair of vertices with at least $k+1$ common neighbors of degree at most k in G.

Intuitively, adding the edges during construction of the improved graph cannot spoil any tree decomposition of G of width at most k, as the pairs of vertices connected by the new edges will need to be contained together in some bag anyway. This is captured in the following lemma.

Lemma 2.6. *Given a graph G and an integer $k \in \mathbb{N}$, $\mathrm{tw}(G) \leq k$ if and only if $\mathrm{tw}(G_I) \leq k$.*

If $|E| = O(kn)$, which is the case in graphs of treewidth at most k, the improved graph can be computed in $O(k^{O(1)} \cdot n)$ time using radix sort [9].

A vertex $v \in G$ is *simplicial* if its neighborhood is a clique, and is called *I-simplicial*, if it is simplicial in the improved graph G_I. The intuition behind I-simplicial vertices is as follows: all the neighbors of an I-simplicial vertex must be simultaneously contained in some bag of any tree decomposition of G_I of width at most k, so we can safely remove such vertices from the improved graph, compute the tree decomposition, and reintroduce the removed I-simplicial vertices. The crucial observation is that if no large set of I-simplicial vertices can be found, then one can identify a large matching, which can be also used for a robust recursion step. The following lemma, which follows from the work of Bodlaender [9], encapsulates all the main ingredients that we will use.

Lemma 2.7. *There is an algorithm running in $O(k^{O(1)}n)$ time that, given a graph $G = (V, E)$ and an integer k, either*

(i) returns a maximal matching in G of cardinality at least $\frac{|V|}{O(k^6)}$,

(ii) returns a set of at least $\frac{|V|}{O(k^6)}$ I-simplicial vertices, or

(iii) correctly concludes that the treewidth of G is larger than k.

Moreover, if a set X of at least $\frac{|V|}{O(k^6)}$ I-simplicial vertices is returned, and the algorithm is in addition provided with some tree decomposition $\mathcal{T}_I$ of $G_I \setminus X$ of width at most k, then in $O(k^{O(1)} \cdot n)$ time one can turn $\mathcal{T}_I$ into a tree decomposition $\mathcal{T}$ of G of width at most k, or conclude that the treewidth of G is larger than k.

Lemma 2.7 allows us to reduce the problem to a *compression* variant where we are given a graph G, an integer k and a tree decomposition of G of width $O(k)$, and the goal is to either conclude that the treewidth of G is at least k or output a tree decomposition of G of width at most $3k + 4$. The proof of Theorem II has two parts: an algorithm for the compression step and an algorithm for the general problem that uses the algorithm for the compression step together with Lemma 2.7 as black boxes. We now state the properties of our algorithm for the compression step in the following lemma.

Lemma 2.8. *There exists an algorithm which on input $G, k, S_0, \mathcal{T}_{apx}$, where (i) $S_0 \subseteq V(G)$, $|S_0| \leq 2k + 3$, (ii) G and $G \setminus S_0$ are connected, and (iii) $\mathcal{T}_{apx}$ is a tree decomposition of G with $\mathrm{w}(\mathcal{T}_{apx}) = O(k)$, in $O(c^k \cdot n \log n)$ time for some $c \in \mathbb{N}$ either computes a tree decomposition $\mathcal{T}$ of G with $\mathrm{w}(\mathcal{T}) \leq 3k + 4$ and S_0 as the root bag, or correctly concludes that $\mathrm{tw}(G) > k$.*

We now give a proof of Theorem II, assuming the correctness of Lemma 2.8. The correctness of the lemma will be argued for in Sections 2.3.1 and 2.3.2.

Proof of Theorem II. Our algorithm will in fact solve a slightly more general problem. Here we are given a graph G, an integer k and a set S_0 on at most $2k + 3$ vertices, with the property that $G \setminus S_0$ is connected. The algorithm will either conclude that $\mathrm{tw}(G) > k$ or output a tree decomposition of width at most $3k + 4$ such that S_0 is the root bag. To get a tree decomposition of any (possibly disconnected) graph it is sufficient to run this algorithm on each connected component with $S_0 = \emptyset$. The algorithm proceeds as follows. It first applies Lemma 2.7 on $(G, 3k + 4)$. If the algorithm of Lemma 2.7 concludes that $\mathrm{tw}(G) > 3k + 4$ the algorithm reports that $\mathrm{tw}(G) > 3k + 4 > k$.

If the algorithm finds a matching M in G with at least $\frac{|V|}{O(k^6)}$ edges, it contracts every edge in M and obtains a graph G'. Since G' is a minor of G we know that $\mathrm{tw}(G') \leq \mathrm{tw}(G)$. The algorithm runs itself recursively on $(G', k, \emptyset)$, and either concludes that $\mathrm{tw}(G') > k$ (implying $\mathrm{tw}(G) > k$) or outputs a tree decomposition of G' of width at most $3k + 4$. Uncontracting the matching in this tree decomposition yields a tree decomposition $\mathcal{T}_{\mathrm{apx}}$ of G of width at most $6k + 9$ [9]. Now we can run the algorithm of Lemma 2.8 on $(G, k, S_0, \mathcal{T}_{\mathrm{apx}})$ and either obtain a tree decomposition of G of width at most $3k + 4$ and S_0 as the root bag, or correctly conclude that $\mathrm{tw}(G) > k$.

If the algorithm finds a set X of at least $\frac{|V|}{O(k^6)}$ I-simplicial vertices, it constructs the improved graph G_I and runs itself recursively on $(G_I \setminus X, k, \emptyset)$. If the algorithm concludes that $\mathrm{tw}(G_I \setminus X) > k$ then $\mathrm{tw}(G_I) > k$ implying $\mathrm{tw}(G) > k$ by Lemma 2.6. Otherwise we obtain a tree decomposition of $G_I \setminus X$ of width at most $3k + 4$. We may now apply Lemma 2.7 and obtain a tree decomposition $\mathcal{T}_{\mathrm{apx}}$ of G with the same width. Note that we can not just output

$\mathcal{T}_{\text{apx}}$ directly, since we can not be sure that S_0 is the top bag of $\mathcal{T}_{\text{apx}}$. However we can run the algorithm of Lemma 2.8 on $(G, k, S_0, \mathcal{T}_{\text{apx}})$ and either obtain a tree decomposition of G of width at most $3k+4$ and S_0 as the root bag, or correctly conclude that $\text{tw}(G) > k$.

It remains to analyze the running time of the algorithm. Suppose the algorithm takes time at most $T(n,k)$ on input (G, k, S_0) where $n = |V(G)|$. Running the algorithm of Lemma 2.7 takes $O(k^{O(1)}n)$ time. Then the algorithm either halts, or calls itself recursively on a graph with at most $n - \frac{n}{O(k^6)} = n(1 - \frac{1}{O(k^6)})$ vertices taking time $T(n(1 - \frac{1}{O(k^6)}), k)$. Then the algorithm takes time $O(k^{O(1)}n)$ to either conclude that $\text{tw}(G) > k$ or to construct a tree decomposition $\mathcal{T}_{\text{apx}}$ of G of width $O(k)$. In the latter case we finally run the algorithm of Lemma 2.8, taking time $O(c^k \cdot n \log n)$. This gives the following recurrence:

$$T(n,k) \leq O\left(c^k \cdot n \log n\right) + T\left(n\left(1 - \frac{1}{O(k^6)}\right), k\right)$$

The recurrence leads to a geometric series and solves to $T(n,k) \leq O(c^k k^{O(1)} \cdot n \log n)$, completing the proof. For a thorough analysis of the recurrence, see Equations 3 and 4 in Section 4. Pseudocode for the algorithm described here is given in Algorithm 1 in Section 4. □

2.3.1 A compression algorithm

We now proceed to give a sketch of a proof for a slightly weakened form of Lemma 2.8. The goal is to give an algorithm that given as input a graph G, an integer k, a set S_0 of size at most $6k+6$ such that $G \setminus S_0$ is connected, and a tree decomposition $\mathcal{T}_{\text{apx}}$ of G, runs in time $O(c^k n \log n)$ and either correctly concludes that $\text{tw}(G) > k$ or outputs a tree decomposition of G of width at most $8k+7$. The paper does not contain a full proof of this variant of Lemma 2.8 — we will discuss the proof of Lemma 2.8 in Section 2.3.2. The aim of this section is to demonstrate that the recursive scheme of Section 2.3 together with a nice trick for finding balanced separators is already sufficient to obtain a factor 8 approximation for treewidth running in time $O(c^k n \log n)$. A variant of the trick used in this section for computing balanced separators turns out to be useful in our final $O(c^k n)$ time 5-approximation algorithm.

The route we follow here is to apply the algorithm of Reed described in Section 2.2, but instead of using Lemma 2.4 to find a set X of size $k+1$ such that every connected component of $G \setminus X$ is small, finding X by dynamic programming over the tree decomposition $\mathcal{T}_{\text{apx}}$ in time $O(c^k n)$. There are a few technical difficulties with this approach.

The most serious issue is that, to the best of our knowledge, the only known dynamic programming algorithms for balanced separators in graphs of bounded treewidth take time $O(c^k n^2)$ rather than $O(c^k n)$: in the state, apart from a partition of the bag, we also need to store the cardinalities of the sides which gives us another dimension of size n. We now explain how it is possible to overcome this issue. We start by applying the argument in the proof of Lemma 2.1 on the tree decomposition $\mathcal{T}_{\text{apx}}$ and get in time $O(k^{O(1)}n)$ a partition of $V(G)$ into L_0, X_0 and R_0 such that there are no edges between L_0 and R_0, $\max(|L_0|, |R_0|) \leq \frac{3}{4}n$ and $|X_0| \leq \text{w}(\mathcal{T}_{\text{apx}}) + 1$. For every way of writing $k+1 = k_L + k_X + k_R$ and every partition of X_0 into $X_L \cup X_X \cup X_R$ with $|X_X| = k_X$ we do the following:

In $O(c^k n)$ time using dynamic programming over the tree decomposition $\mathcal{T}_{\text{apx}}$, we

- first find a partition of $L_0 \cup X_0$ into $\hat{L}_L \cup \hat{R}_L \cup \hat{X}_L$ such that there are no edges from $\hat{L}_L$ to $\hat{R}_L$, $|\hat{X}_L| \leq k_L + k_X$, $X_X \subseteq \hat{X}_L$, $X_R \subseteq \hat{R}_L$ and $X_L \subseteq \hat{L}_L$ and the size $|\hat{L}_L|$ is maximized, and
- then find a partition of $R_0 \cup X_0$ into $\hat{L}_R \cup \hat{R}_R \cup \hat{X}_R$ such that there are no edges from $\hat{L}_R$ to $\hat{R}_R$, $|\hat{X}_R| \leq k_R + k_X$, $X_X \subseteq \hat{X}_R$, $X_R \subseteq \hat{R}_R$ and $X_L \subseteq \hat{L}_R$ and the size $|\hat{R}_R|$ is maximized.

Let $L = L_L \cup L_R$, $R = R_L \cup R_R$ and $X = X_L \cup X_R$. The sets L, X, R form a partition of $V(G)$ with no edges from L to R and $|X| \leq k_L + k_X + k_R + k_X - k_X \leq k + 1$.

It is possible to show using a combinatorial argument (see Lemma 7.5 in Section 7) that if $\mathrm{tw}(G) \leq k$ then there exists a choice of k_L, k_X, k_R such that $k+1 = k_L + k_X + k_R$ and partition of X_0 into $X_L \cup X_X \cup X_R$ with $|X_X| = k_X$ such that the above algorithm will output a partition of $V(G)$ into X, L and R such that $\max(|L|, |R|) \leq \frac{8n}{9}$. Thus we have an algorithm that in time $O(c^k n)$ either finds a set X of size at most $k+1$ such that each connected component of $G \setminus X$ has size at most $\frac{8n}{9}$ or correctly concludes that $\mathrm{tw}(G) > k$.

The second problem with the approach is that the algorithm of Reed is an 8-approximation algorithm rather than a 3-approximation. Thus, even the sped up version does not quite prove Lemma 2.8. It does however yield a version of Lemma 2.8 where the compression algorithm is an 8-approximation. In the proof of Theorem II there is nothing special about the number 3 and so one can use this weaker variant of Lemma 2.8 to give a 8-approximation algorithm for treewidth in time $O(c^k n \log n)$. We will not give complete details of this algorithm, as we will shortly describe a proof of Lemma 2.8 using a quite different route.

It looks difficult to improve the algorithm above to an algorithm with running time $O(c^k n)$. The main hurdle is the following: both the algorithm of Robertson and Seymour [40] and the algorithm of Reed [37] find a separator X and proceed recursively on the components of $G \setminus X$. If we use $O(c^k \cdot n)$ time to find the separator X, then the total running time must be at least $O(c^k \cdot n \cdot d)$ where d is the depth of the recursion tree of the algorithm. It is easy to see that the depth of the tree decomposition output by the algorithms equals (up to constant factors) the depth of the recursion tree. However there exist graphs of treewidth k such that no tree decomposition of depth $o(\log n)$ has width $O(k)$ (take for example powers of paths). Thus the depth of the constructed tree decompositions, and hence the recursion depth of the algorithm must be at least $\Omega(\log n)$.

Even if we somehow managed to reuse computations and find the separator X in time $O(c^k \cdot \frac{n}{\log n})$ on average, we would still be in trouble since we need to pass on the list of vertices of the connected components of $G \setminus X$ that we will call the algorithm on recursively. At a first glance this has to take $O(n)$ time and then we are stuck with an algorithm with running time $O((c^k \cdot \frac{n}{\log n} + n) \cdot d)$, where d is the recursion depth of the algorithm. For $d = \log n$ this is still $O(c^k n + n \log n)$ which is slower than what we are aiming at. In Section 2.3.2 we give a proof of Lemma 2.8 that *almost* overcomes these issues.

2.3.2 A better compression algorithm

We give a sketch of the proof of Lemma 2.8. The goal is to give an algorithm that given as input a connected graph G, an integer k, a set S_0 of size at most $2k+3$ such that $G \setminus S_0$ is connected, and a tree decomposition $\mathcal{T}_{\mathrm{apx}}$ of G, runs in time $O(c^k n \log n)$ and either correctly concludes that $\mathrm{tw}(G) > k$ or outputs a tree decomposition of G of width at most $3k+4$ with top bag S_0.

Our strategy is to implement the $O(c^k n^2)$ time 4-approximation algorithm described in Section 2.1, but make some crucial changes in order to (a) make the implementation run in $O(c^k n \log n)$ time, and (b) make it a 3-approximation rather than a 4-approximation. We first turn to the easier of the two changes, namely making the algorithm a 3-approximation.

To get an algorithm that satisfies all of the requirements of Lemma 2.8, but runs in time $O(c^k n^2)$ rather than $O(c^k n \log n)$ we run the algorithm described in Section 2.1 setting $S = S_0$ in the beginning. Instead of using Lemma 2.2 to find a set X such that every component of $G \setminus X$ has at most $\frac{2}{3}|S|$ vertices which are in S, we apply Lemma 2.1 to show the *existence* of an X such that every component of $G \setminus X$ has at most $\frac{1}{2}|S|$ vertices which are in S, and do dynamic programming over the tree decomposition $\mathcal{T}_{\mathrm{apx}}$ in time $O(c^k n)$ in order to find such an

X. Going through the analysis of Section 2.1 but with X satisfying that every component of $G \setminus X$ has at most $\frac{1}{2}|S|$ vertices which are in S shows that the algorithm does in fact output a tree decomposition with width $3k+4$ and top bag S_0 whenever $\mathrm{tw}(G) \leq k$.

It is somewhat non-trivial to do dynamic programming over the tree decomposition $\mathcal{T}_{\mathrm{apx}}$ in time $O(c^k n)$ in order to find an X such that every component of $G \setminus X$ has at most $\frac{2}{3}|S|$ vertices which are in S. The problem is that $G \setminus X$ could potentially have many components and we do not have time to store information about each of these components individually. The following lemma, whose proof can be found in Section 7.4.2, shows how to deal with this problem.

Lemma 2.9. *Let G be a graph and $S \subseteq V(G)$. Then a set X is a balanced S-separator if and only if there exists a partition (M_1, M_2, M_3) of $V(G) \setminus X$, such that there is no edge between M_i and M_j for $i \neq j$, and $|M_i \cap S| \leq \frac{1}{2}|S|$ for $i = 1, 2, 3$.*

Lemma 2.9 shows that when looking for a balanced S-separator we can just look for a partition of G into four sets X, M_1, M_2, M_3 such that there is no edge between M_i and M_j for $i \neq j$, and $|M_i \cap S| \leq \frac{1}{2}|S|$ for $i = 1, 2, 3$. This can easily be done in time $O(c^k n)$ by dynamic programming over the tree decomposition $\mathcal{T}_{\mathrm{apx}}$. This yields the promised algorithm that satisfies all of the requirements of Lemma 2.8, but runs in time $O(c^k n^2)$ rather than $O(c^k n \log n)$.

We now turn to the most difficult part of the proof of Lemma 2.8, namely how to improve the running time of the algorithm above from $O(c^k n^2)$ to $O(c^k n \log n)$ in a way that gives hope of a further improvement to running time $O(c^k n)$. The $O(c^k n \log n)$ time algorithm we describe now is based on the following observations; (a) In any recursive call of the algorithm above, the considered graph is an induced subgraph of G. Specifically, the considered graph is always $G[C \cup S]$ where S is a set with at most $2k+3$ vertices and C is a connected component of $G \setminus S$. (b) The only computationally hard step, finding the balanced S-separator X, is done by dynamic programming over the tree decomposition $\mathcal{T}_{\mathrm{apx}}$ of G. The observations (a) and (b) give some hope that one can reuse the computations done in the dynamic programming when finding a balanced S-separator for G during the computation of balanced S-separators in induced subgraphs of G. This plan can be carried out in a surprisingly clean manner and we now give a rough sketch of how it can be done.

We start by preprocessing the tree decomposition using an algorithm of Bodlaender and Hagerup [14]. This algorithm is a parallel algorithm and here we state its sequential form.

Proposition 2.10 (Bodlaender and Hagerup [14])**.** *There is an algorithm that, given a tree decomposition of width k with $O(n)$ nodes of a graph G, finds a rooted binary tree decomposition of G of width at most $3k+2$ with depth $O(\log n)$ in $O(kn)$ time.*

Proposition 2.10 lets us assume without loss of generality that the tree decomposition $\mathcal{T}_{\mathrm{apx}}$ has depth $O(\log n)$.

In Section 7 we will describe a data structure with the following properties. The data structure takes as input a graph G, an integer k and a tree decomposition $\mathcal{T}_{\mathrm{apx}}$ of width $O(k)$ and depth $O(\log n)$. After an initialization step which takes $O(c^k n)$ time the data structure allows us to do certain operations and queries. At any point of time the data structure is in a certain *state*. The operations allow us to change the state of the data structure. Formally, the state of the data structure is a quadruple (S, X, F, π), where S, X and F are subsets of $V(G)$ and π is a vertex called the "pin", with the restriction that $\pi \notin S$. The initial state of the data structure is that $S = S_0$, $X = F = \emptyset$, and π is an arbitrary vertex of $G \setminus S_0$. The data structure allows operations that change S, X or F by inserting/deleting a specified vertex, and move the pin to a specified vertex in time $O(c^k \log n)$.

For a fixed state of the data structure, the *active component* U is the component of $G \setminus S$ which contains π. Given S and $\pi \notin S$, the set U is uniquely defined. The data structure allows

the query findSSeparator, which outputs in time $O(c^k \log n)$ either a balanced S-separator $\hat{X}$ of $G[U \cup S]$ of size at most $k+1$, or $\perp$, which means that $\mathrm{tw}(G[S \cup U]) > k$.

The algorithm of Lemma 2.8 runs the $O(c^k n^2)$ time algorithm described above, but uses the *data structure* to find the balanced S-separator in time $O(c^k \log n)$ instead of doing dynamic programming over $\mathcal{T}_{\mathrm{apx}}$. All we need to make sure is that the S in the state of the data structure is always equal to the set S for which we want to find the balanced separator, and that the active component U is set such that $G[U \cup S]$ is equal to the induced subgraph we are working on. Since we always maintain that $|S| \leq 2k+3$ we can change the set S to anywhere in the graph (and specifically into the correct position) by doing $k^{O(1)}$ operations taking $O(c^k \log n)$ time each.

At a glance, it appears that if we assume the data structure as a black box, this is sufficient to obtain the desired $O(c^k n \log n)$ time algorithm. However, we haven't even used the sets X and F in the state of the data structure, or described what they mean! The reason for this is of course that there is a complication. In particular, after the balanced S-separator $\hat{X}$ is found — how can we recurse into the connected components of $G[S \cup U] \setminus (S \cup \hat{X})$? We need to move the pin into each of these components one at a time, but if we want to use $O(c^k \log n)$ time in each recursion step, we cannot afford to spend $O(|S \cup U|)$ time to compute the connected components of $G[S \cup U] \setminus (S \cup \hat{X})$. We resolve this issue by pushing the problem into the data structure, and showing that the appropriate queries can be implemented there. This is where the sets X and F in the state of the data structure come in.

The rôle of X in the data structure is that when queries to the data structure depending on X are called, X equals the set $\hat{X}$, i.e., the balanced S-separator found by the query findSSeparator. The set F is a set of "finished pins" whose intention is the following: when the algorithm calls itself recursively on a component U' of $G[S \cup U] \setminus (S \cup \hat{X})$ after it has finished computing a tree decomposition of $G[U' \cup N(U')]$ with $N(U')$ as its top bag, it selects an arbitrary vertex of U' and inserts it into F.

The query findNextPin finds a new pin π' in a component U' of $G[S \cup U] \setminus (S \cup \hat{X})$ that does not contain any vertices of F. And finally, the query findNeighborhood allows us to find the neighborhood $N(U')$, which in turn allows us to call the algorithm recursively in order to find a tree decomposition of $G[U' \cup N(U')]$ with $N(U')$ as its top bag.

At this point it is possible to convince oneself that the $O(c^k n^2)$ time algorithm described in the beginning of this section can be implemented using $O(k^{O(1)})$ calls to the data structure in each recursive step, thus spending only $O(c^k \log n)$ time in each recursive step. Pseudocode for this algorithm can be found in Algorithm 3. The recurrence bounding the running time of the algorithm then becomes

$$T(n,k) \leq O(c^k \log n) + \sum_{U_i} T(|U_i \cup \hat{X}|, k).$$

Here $U_1, \ldots, U_q$ are the connected components of $G[S \cup U] \setminus (S \cup \hat{X})$. This recurrence solves to $O(c^k n \log n)$, proving Lemma 2.8. A full proof of Lemma 2.8 assuming the data structure as a black box may be found in Section 4.2.

2.4 The data structure

We sketch the main ideas in the implementation of the data structure. The goal is to set up a data structure that takes as input a graph G, an integer k and a tree decomposition $\mathcal{T}_{\mathrm{apx}}$ of width $O(k)$ and depth $O(\log n)$, and initializes in time $O(c^k n)$. The *state* of the data structure is a quadruple (S, X, F, π) where S, X and F are vertex sets in G and $\pi \in V(G) \setminus S$. The initial state of the data structure is $(S_0, \emptyset, \emptyset, v)$ where v is an arbitrary vertex in $G \setminus S_0$. The data

structure should support operations that insert (delete) a single vertex to (from) S, X and F, and an operation to change the pin π to a specified vertex. These operations should run in time $O(c^k \log n)$. For a given state of the data structure, set U to be the component of $G \setminus S$ containing π. The data structure should also support the following queries in time $O(c^k \log n)$.

- findSSeparator: Assuming that $|S| \leq 2k+3$, return a set $\hat{X}$ of size at most $k+1$ such that every component of $G[S \cup U] \setminus \hat{X}$ contains at most $\frac{1}{2}|S|$ vertices of S, or conclude that $\mathrm{tw}(G) > k$.
- findNextPin: Return a vertex π' in a component U' of $G[S \cup U] \setminus (S \cup \hat{X})$ that does not contain any vertices of F.
- findNeighborhood: Return $N(U) \cap S$ if $|N(U) \cap S| < 2k+3$ and a marker '$\boxtimes$' otherwise.

Suppose for now that we want to set up a much simpler data structure. Here the state is just the set S and the only query we want to support is findSSeparator which returns a set $\hat{X}$ such that every component of $G \setminus (S \cup \hat{X})$ contains at most $\frac{1}{2}|S|$ vertices of S, or conclude that $\mathrm{tw}(G) > k$. At our disposal we have the tree decomposition $\mathcal{T}_{\mathrm{apx}}$ of width $O(k)$ and depth $O(\log n)$. To set up the data structure we run a standard dynamic programming algorithm for finding $\hat{X}$ given S. Here we use Lemma 2.9 and search for a partition of $V(G)$ into (M_1, M_2, M_3, X) such that $|X| \leq k+1$, there is no edge between M_i and M_j for $i \neq j$, and $|M_i \cap S| \leq \frac{1}{2}|S|$ for $i = 1, 2, 3$. This can be done in time $O(c^k k^{O(1)} n)$ and the tables stored at each node of the tree decomposition have size $O(c^k k^{O(1)})$. This finishes the initialization step of the data structure. The initialization step took time $O(c^k k^{O(1)} n)$.

We will assume without loss of generality that the top bag of the decomposition is empty. The data structure will maintain the following invariant: after every change has been performed the tables stored at each node of the tree decomposition correspond to a valid execution of the dynamic programming algorithm on input (G, S). If we are able to maintain this invariant, then answering findSSeparator queries is easy: assuming that each cell of the dynamic programming table also stores solution sets (whose size is at most $k+1$) we can just output in time $k^{O(1)}$ the content of the top bag of the decomposition!

But how to maintain the invariant and support changes in time $O(c^k \log n)$? It turns out that this is not too difficult: the content of the dynamic programming table of a node t in the tree decomposition depends only on S and the dynamic programming tables of t's children. Thus, when the dynamic programming table of the node t is changed, this will only affect the dynamic programming tables of the $O(\log n)$ ancestors of t. If the dynamic program is done carefully, one can ensure that adding or removing a vertex to/from S will only affect the dynamic programming tables for a single node t in the decomposition, together with all of its $O(\log n)$ ancestors. Performing the changes amounts to recomputing the dynamic programming tables for these nodes, and this takes time $O(c^k k^{O(1)} \log n)$.

It should now be plausible that the idea above can be extended to work also for the more complicated data structure with the more advanced queries. Of course there are several technical difficulties, the main one is how to ensure that the computation is done in the connected component U of $G \setminus S$ without having to store "all possible ways the vertices in a bag could be connected below the bag". We omit the details of how this can be done in this outline. The full exposition of the data structure can be found in Section 7.

2.5 Approximating treewidth in $O(c^k n \log^{(\alpha)} n)$ time

We now sketch how the algorithm of the previous section can be sped up, at the cost of increasing the approximation ratio from 3 to 5. In particular we give a proof outline for the following theorem.

Theorem III. *For every $\alpha \in \mathbb{N}$, there exists an algorithm which, given a graph G and an integer k, in $O(c^k \cdot n \log^{(\alpha)} n)$ time for some $c \in \mathbb{N}$ either computes a tree decomposition of G of width at most $5k+3$ or correctly concludes that $\mathrm{tw}(G) > k$.*

The algorithm of Theorem II satisfies the conditions of Theorem III for $\alpha = 1$. We will show how one can use the algorithm for $\alpha = 1$ in order to obtain an algorithm for $\alpha = 2$. In particular we aim at an algorithm which given a graph G and an integer k, in $O(c^k \cdot n \log\log n)$ time for some $c \in \mathbb{N}$ either computes a tree decomposition of G of width at most $5k+3$ or correctly concludes that $\mathrm{tw}(G) > k$.

We inspect the $O(c^k n \log n)$ algorithm for the compression step described in Section 2.3.2. It uses the data structure of Section 2.4 in order to find balanced separators in time $O(c^k \log n)$. The algorithm uses $O(c^k \log n)$ time on each recursive call regardless of the size of the induced subgraph of G it is currently working on. When the subgraph we work on is big this is very fast. However, when we get down to induced subgraphs of size $O(\log\log n)$ the algorithm of Robertson and Seymour described in Section 2.1 would spend $O(c^k (\log\log n)^2)$ time in each recursive call, while our presumably fast algorithm still spends $O(c^k \log n)$ time. This suggests that there is room for improvement in the recursive calls where the considered subgraph is very small compared to n.

The overall structure of our $O(c^k \log\log n)$ time algorithm is identical to the structure of the $O(c^k \log n)$ time algorithm of Theorem II. The only modifications happen in the compression step. The compression step is also similar to the $O(c^k \log n)$ algorithm described in Section 2.3.2, but with the following caveat. The data structure query findNextPin finds the *largest* component where a new pin can be placed, returns a vertex from this component, and also returns the size of this component. If a call of findNextPin returns that the size of the largest yet unprocessed component is less than $\log n$ the algorithm does not process this component, nor any of the other remaining components in this recursive call. This ensures that the algorithm is never run on instances where it is slow. Of course, if we do not process the small components we do not find a tree decomposition of them either. A bit of inspection reveals that what the algorithm will do is either conclude that $\mathrm{tw}(G) > k$ or find a tree decomposition of an induced subgraph of G' of width at most $3k+4$ such that for each connected component C_i of $G \setminus V(G')$, (a) $|C_i| \leq \log n$, (b) $|N(C_i)| \leq 2k+3$, and (c) $N(C_i)$ is fully contained in some bag of the tree decomposition of G'.

How much time does it take the algorithm to produce this output? Each recursive call takes $O(c^k \log n)$ time and adds a bag to the tree decomposition of G' that contains some vertex which was not yet in $V(G')$. Thus the total time of the algorithm is upper bounded by $O(|V(G')| \cdot c^k \log n)$. What happens if we run this algorithm, then run the $O(c^k n \log n)$ time algorithm of Theorem II on each of the connected components of $G \setminus V(G')$? If either of the recursive calls return that the treewidth of the component is more than k then $\mathrm{tw}(G) > k$. Otherwise we have a tree decomposition of each of the connected components with width $3k+4$. With a little bit of extra care we find tree decompositions of the same width of $C_i \cup N(C_i)$ for each component C_i, such that the top bag of the decomposition contains $N(C_i)$. Then all of these decompositions can be glued together with the decomposition of G' to yield a decomposition of width $3k+4$ for the entire graph G.

The running time of the above algorithm can be bounded as follows. It takes $O(|V(G')| \cdot c^k \log n)$ time to find the partial tree decomposition of G', and

$$O(\sum_i c^k |C_i| \log |C_i|) \leq O(c^k \log\log n \cdot \sum_i |C_i|) \leq O(c^k n \log\log n)$$

time to find the tree decompositions of all the small components. Thus, if $|V(G')| \leq O(\frac{n}{\log n})$

the running time of the first part would be $O(c^k n)$ and the total running time would be $O(c^k n \log\log n)$.

How big can $|V(G')|$ be? In other words, if we inspect the algorithm described in Section 2.1, how big part of the graph does the algorithm see before all remaining parts have size less than $\log n$? The bad news is that the algorithm could see almost the entire graph. Specifically if we run the algorithm on a path it could well be building a tree decomposition of the path by moving along the path and only terminating when reaching the vertex which is $\log n$ steps away from from the endpoint. The good news is that the algorithm of Reed described in Section 2.2 will get down to components of size $\log n$ after decomposing only $O(\frac{n}{\log n})$ vertices of G. The reason is that the algorithm of Reed also finds balanced separators of the considered subgraph, ensuring that the size of the considered components drop by a constant factor for each step down in the recursion tree.

Thus, if we augment the algorithm that finds the tree decomposition of the subgraph G' such that it also finds balanced separators of the active component and adds them to the top bag of the decomposition before going into recursive calls, this will ensure that $|V(G')| \leq O(\frac{n}{\log n})$ and that the total running time of the algorithm described in the paragraphs above will be $O(c^k n \log\log n)$. The algorithm of Reed described in Section 2.2 has a worse approximation ratio than the algorithm of Robertson and Seymour described in Section 2.1. The reason is that we also need to add the balanced separator to the top bag of the decomposition. When we augment the algorithm that finds the tree decomposition of the subgraph G' in a similar manner, the approximation ratio also gets worse. If we are careful about how the change is implemented we can still achieve an algorithm with running time $O(c^k n \log\log n)$ that meets the specifications of Theorem III for $\alpha = 2$.

The approach to improve the running time from $O(c^k n \log n)$ to $O(c^k n \log\log n)$ also works for improving the running time from $O(c^k \cdot n \log^{(\alpha)} n)$ to $O(c^k \cdot n \log^{(\alpha+1)} n)$. Running the algorithm that finds in $O(c^k n)$ time the tree decomposition of the subgraph G' such that all components of $G \setminus V(G')$ have size $\log n$ and running the $O(c^k \cdot n \log^{(\alpha)} n)$ time algorithm on each of these components yields an algorithm with running time $O(c^k \cdot n \log^{(\alpha+1)} n)$.

In the above discussion we skipped over the following issue. How can we compute a small balanced separator for the active component in time $O(c^k \log n)$? It turns out that also this can be handled by the data structure. The main idea here is to consider the dynamic programming algorithm used in Section 2.3.1 to find balanced separators in graphs of bounded treewidth, and show that this algorithm can be turned into an $O(c^k \log n)$ time data structure query. We would like to remark here that the implementation of the trick from Section 2.3.1 is significantly more involved than the other queries: we need to use the approximate tree decomposition not only for fast dynamic programming computations, but also to locate the separation (L_0, X_0, R_0) on which the trick is employed. A detailed explanation of how this is done can be found at the end of Section 7.4.4. This completes the proof sketch of Theorem III. A full proof can be found in Section 5.

2.6 5-approximation in $O(c^k n)$ time

The algorithms of Theorem 5 are in fact already $O(c^k n)$ algorithms unless n is astronomically large compared to k. If, for example, $n \leq 2^{2^{2^k}}$ then $\log^{(3)} n \leq k$ and so $O(c^k n \log^{(3)} n) \leq O(c^k k n)$. Thus, to get an algorithm which runs in $O(c^k n)$ it is sufficient to consider the cases when n is *much greater than* k. The recursive scheme of Section 2.3 allows us to only consider the case where (a) n is much greater than k and (b) we have at our disposal a tree decomposition $\mathcal{T}_{\text{apx}}$ of G of width $O(k)$.

For this case, consider the dynamic programming algorithm of Bodlaender and Kloks [15] that

given G and a tree decomposition $\mathcal{T}_{\text{apx}}$ of G of width $O(k)$ either computes a tree decomposition of G of width k or concludes that $\text{tw}(G) > k$ in time $O(2^{O(k^3)}n)$. The dynamic programming algorithm can be turned into a tree automata based algorithm [2, 26] with running time $O(2^{2^{O(k^3)}} + n)$ if one can inspect an arbitrary entry of a table of size $O(2^{2^{O(k^3)}})$ in constant time. If $n \geq \Omega(2^{2^{O(k^3)}})$ then inspecting an arbitrary entry of a table of size $O(2^{2^{O(k^3)}})$, means inspecting an arbitrary entry of a table of size $O(n)$, which one can do in constant time in the RAM model. Thus, when $n \geq \Omega(2^{2^{O(k^3)}})$ we can find an optimal tree decomposition in time $O(n)$. When $n \leq O(2^{2^{O(k^3)}})$ the $O(c^k n \log^{(3)} n)$ time algorithm of Theorem 5 runs in time $O(c^k kn)$. This concludes the outline of the proof of Theorem I. A full explanation of how to handle the case where n is much greater than k can be found in Section 6.

3 Conclusions

In this paper we have presented an algorithm that gives a constant factor approximation (with a factor 5) of the treewidth of a graph, which runs in single exponential time in the treewidth and linear in the number of vertices of the input graph. Here we give some consequences of the result, possible improvements and open problems.

3.1 Consequences, corollaries and future work

A large number of computational results use the following overall scheme: first find a tree decomposition of bounded width, and then run a dynamic programming algorithm on it. Many of these results use the linear-time exact algorithm of Bodlaender [9] for the first step. If we aim for algorithms whose running time dependency on treewidth is single exponential, however, then our algorithm is preferable over the exact algorithm of Bodlaender [9]. Indeed, many classical problems like Dominating Set and Independent Set are easily solvable in time $O(c^k \cdot n)$ when a tree decomposition of width k is provided. Furthermore, there is an on-going work on finding new dynamic programming routines with such a running time for problems seemingly not admitting so robust solutions; the fundamental examples are Steiner Tree, Traveling Salesman and Feedback Vertex Set [11]. The results of this paper show that for all these problems we may also claim an $O(c^k \cdot n)$ running time even if the decomposition is not given to us explicitly, as we may find its constant factor approximation within the same complexity bound.

Our algorithm is also compatible with the celebrated Courcelle's theorem [20], which states that every graph problem expressible in monadic second-order logic (MSOL) is solvable in time $f(k, ||\varphi||) \cdot n$ when a tree decomposition of width k is provided, where φ is the formula expressing the problem and f is some function. Again, the first step of applying Courcelle's theorem is usually computing an optimal tree decomposition using the linear-time algorithm of Bodlaender [9]. Using the results of this paper, this step can be substituted with finding an approximate decomposition in $O(c^k n)$ time. For many problems, in the overall running time analysis we may thus significantly reduce the factor dependent on the treewidth of the graph, while keeping the linear dependence on n at the same time.

It seems that the main novel idea of this paper, namely treating the tree decomposition as a data structure on which logarithmic-time queries can be implemented, can be similarly applied to all the problems expressible in MSOL. Extending our results in this direction seems like a thrilling perspective for future work.

Concrete examples where the results of this paper can be applied, can be found also within the framework of *bidimensionality theory* [22, 23]. In all parameterized subexponential algorithms

obtained within this framework, the polynomial dependence of n in the running time becomes linear if our algorithm is used. For instance, it follows immediately that every parameterized minor bidimensional problem with parameter k, solvable in time $2^{O(t)}n$ on graphs of treewidth t, is solvable in time $2^{O(\sqrt{k})}n$ on graphs excluding some fixed graph as a minor.

3.2 Improvements and open problems

Our result is mainly of theoretical importance due to the large constant c at the base of the exponent. One immediate open problem is to obtain a constant factor approximation algorithm for treewidth with running time $O(c^k n)$, where c is a *small* constant.

Another open problem is to find more efficient *exact* FPT algorithms for treewidth. Bodlaender's algorithm [9] and the version of Perković and Reed [36] both use $O(k^{O(k^3)}n)$ time; the dominant term being a call to the dynamic programming algorithm of [15]. In fact, no exact FPT algorithm for treewidth is known whose running time as a function of the parameter k is asymptotically smaller than this; testing the treewidth by verifying the forbidden minors can be expected to be significantly slower. Thus, it would be very interesting to have an exact algorithm for testing if the treewidth of a given graph is at most k in $2^{o(k^3)}n^{O(1)}$ time.

Currently, the best approximation ratio for treewidth for algorithms whose running time is polynomial in n and single exponential in the treewidth is the 3-approximation algorithm from Section 4. What is the best approximation ratio for treewidth that can be obtained in this running time? Is it possible to give lower bounds?

4 An $O(c^k n \log n)$ 3-approximation algorithm for treewidth

In this section, we provide formal details of the proof of Theorem II:

Theorem IV (Theorem II, restated)**.** *There exists an algorithm which, given a graph G and an integer k, in $O(c^k \cdot n \log n)$ time for some $c \in \mathbb{N}$ either computes a tree decomposition of G of width at most $3k+4$ or correctly concludes that* $\mathrm{tw}(G) > k$.

In fact, the algorithm that we present, is slightly more general. The main procedure, $\texttt{Alg}_1$, takes as input a connected graph G, an integer k, and a subset of vertices S_0 such that $|S_0| \leq 2k+3$. Moreover, we have a guarantee that not only G is connected, but $G \setminus S_0$ as well. $\texttt{Alg}_1$ runs in $O(c^k \cdot n \log n)$ time for some $c \in \mathbb{N}$ and either concludes that $\mathrm{tw}(G) > k$, or returns a tree decomposition of G of width $\leq 3k+4$, such that S_0 is the root bag. Clearly, to prove Theorem II, we can run $\texttt{Alg}_1$ on every connected component of G separately using $S_0 = \emptyset$. Note that computation of the connected components takes $O(|V| + |E|) = O(kn)$ time, since if $|E| > kn$, then we can safely output that $\mathrm{tw}(G) > k$.

The presented algorithm $\texttt{Alg}_1$ uses two subroutines. As described in Section 2, $\texttt{Alg}_1$ uses the reduction approach developed by the first author [9]; in short words, we can either apply a reduction step, or find an approximate tree decomposition of width $O(k)$ on which a compression subroutine $\texttt{Compress}_1$ can be employed. In this compression step we are either able to find a refined, compressed tree decomposition of width at most $3k+4$, or again conclude that $\mathrm{tw}(G) > k$.

The algorithm $\texttt{Compress}_1$ starts by initializing the data structure (see Section 2 for an intuitive description of the role of the data structure), and then calls a subroutine `FindTD`. This subroutine resembles the algorithm of Robertson and Seymour (see Section 2): it divides the graph using balanced separators, recurses on the different connected components, and combines the subtrees obtained for the components into the final tree decomposition.

4.1 The main procedure $\texttt{Alg}_1$

Algorithm $\texttt{Alg}_1$, whose layout is proposed as Algorithm 1, runs very similarly to the algorithm of the first author [9]; we provide here all the necessary details for the sake of completeness, but we refer to [9] for a wider discussion.

First, we apply Lemma 2.7 on graph G for parameter $3k+4$. We either immediately conclude that $\mathrm{tw}(G) > 3k+4$, find a set of I-simplicial vertices of size at least $\frac{n}{O(k^6)}$, or a matching of size at least $\frac{n}{O(k^6)}$. Note that in the application of Lemma 2.7 we ignore the fact that some of the vertices are distinguished as S_0.

If a matching M of size at least $\frac{n}{O(k^6)}$ is found, we employ a similar strategy as in [9]. We first contract the matching M to obtain G'; note that if G had treewidth at most k then so does G'. Then we apply $\texttt{Alg}_1$ recursively to obtain a tree decomposition $\mathcal{T}'$ of G' of width at most $3k+4$, and having achieved this we decontract the matching M to obtain a tree decomposition $\mathcal{T}$ of G of width at most $6k+9$: every vertex in the contracted graph is replaced by at most two vertices before the contraction. Finally, we call the sub-procedure $\texttt{Compress}_1$, which given G, S_0, k and the decomposition $\mathcal{T}$ (of width $O(k)$), either concludes that $\mathrm{tw}(G) > k$, or provides a tree decomposition of G of width at most $3k+4$, with S_0 as the root bag. $\texttt{Compress}_1$ is given in details in the next section.

In case of obtaining a large set X of I-simplicial vertices, we proceed similarly as in [9]. We compute the improved graph, remove X from it, apply $\texttt{Alg}_1$ on $G_I \setminus X$ recursively to obtain its tree decomposition $\mathcal{T}'$ of width at most $3k+4$, and finally reintroduce the missing vertices of X to obtain a tree decomposition $\mathcal{T}$ of G of width at most $3k+4$ (recall that reintroduction can fail, and in this case we may conclude that $\mathrm{tw}(G) > k$). Observe that the decomposition $\mathcal{T}$ satisfies all the needed properties, with the exception that we have not guaranteed that S_0 is the root bag. However, to find a decomposition that has S_0 as the root bag, we may again make use of the subroutine $\texttt{Compress}_1$, running it on input G, S_0, k and the tree decomposition $\mathcal{T}$. Lemma 2.7 ensures that all the described steps, apart from the recursive calls to $\texttt{Alg}_1$ and $\texttt{Compress}_1$, can be performed in $O(k^{O(1)} \cdot n)$ time. Note that the I-simplicial vertices can safely be reintroduced since we used Lemma 2.7 for parameter $3k+4$ instead of k.

Let us now analyze the running time of the presented algorithm, provided that the running time of the subroutine $\texttt{Compress}_1$ is $O(c^k \cdot n \log n)$ for some $c \in \mathbb{N}$. Since all the steps of the algorithm (except for calls to subroutines) can be performed in $O(k^{O(1)} \cdot n)$ time, the time complexity satisfies the following recurrence relation:

$$T(n) \leq O(k^{O(1)} \cdot n) + O(c^k \cdot n \log n) + T\left(\left(1 - \frac{1}{C \cdot k^6}\right) n\right); \tag{3}$$

Here C is the constant hidden in the O-notation in Lemma 2.7. By unraveling the recurrence into a geometric series, we obtain that

$$\begin{aligned} T(n) &\leq \sum_{i=0}^{\infty} \left(1 - \frac{1}{Ck^6}\right)^i O(k^{O(1)} \cdot n + c^k \cdot n \log n) \\ &= Ck^6 \cdot O(k^{O(1)} \cdot n + c^k \cdot n \log n) = O(c_1^k \cdot n \log n), \end{aligned} \tag{4}$$

for some $c_1 > c$.

4.2 Compression

In this section we provide the details of the implementation of the subroutine $\texttt{Compress}_1$. The main goal is encapsulated in the following lemma.

Input: A connected graph G, an integer k, and $S_0 \subseteq V(G)$ s.t. $|S_0| \leq 2k+3$ and $G \setminus S_0$ is connected.

Output: A tree decomposition $\mathcal{T}$ of G with $\mathrm{w}(\mathcal{T}) \leq 3k+4$ and S_0 as the root bag, or conclusion that $\mathrm{tw}(G) > k$.

Run algorithm of Lemma 2.7 for parameter $3k+4$

if *Conclusion that* $\mathrm{tw}(G) > 3k+4$ **then**
  return $\perp$
end

if *G has a matching M of cardinality at least* $\frac{n}{O(k^6)}$ **then**
  Contract M to obtain G'.
  $\mathcal{T}' \leftarrow \mathtt{Alg}_1(G', k)$ /* $\mathrm{w}(\mathcal{T}') \leq 3k+4$ */
  if $\mathcal{T}' = \perp$ **then**
    return $\perp$
  else
    Decontract the edges of M in $\mathcal{T}'$ to obtain $\mathcal{T}$.
    return $\mathtt{Compress}_1(G, k, \mathcal{T})$
  end
end

if *G has a set X of at least* $\frac{n}{O(k^6)}$ *I-simplicial vertices* **then**
  Compute the improved graph G_I and remove X from it.
  $\mathcal{T}' \leftarrow \mathtt{Alg}_1(G_I \setminus X, k)$ /* $\mathrm{w}(\mathcal{T}') \leq 3k+4$ */
  if $\mathcal{T}' = \perp$ **then**
    return $\perp$
  end
  Reintroduce vertices of X to $\mathcal{T}'$ to obtain $\mathcal{T}$.
  if *Reintroduction failed* **then**
    return $\perp$
  else
    return $\mathtt{Compress}_1(G, k, \mathcal{T})$
  end
end

Algorithm 1: $\mathtt{Alg}_1$

Lemma 4.1 (Lemma 2.8, restated)**.** *There exists an algorithm which on input $G, k, S_0, \mathcal{T}_{apx}$, where (i) $S_0 \subseteq V(G)$, $|S_0| \leq 2k+3$, (ii) G and $G \setminus S_0$ are connected, and (iii) $\mathcal{T}_{apx}$ is a tree decomposition of G of width at most $O(k)$, in $O(c^k \cdot n \log n)$ time for some $c \in \mathbb{N}$ either computes a tree decomposition $\mathcal{T}$ of G with $\mathrm{w}(\mathcal{T}) \leq 3k+4$ and S_0 as the root bag, or correctly concludes that $\mathrm{tw}(G) > k$.*

The subroutine's layout is given as Algorithm 2. Shortly speaking, we first initialize the data structure $\mathcal{DS}$, with $G, k, S_0, \mathcal{T}$ as input, and then run a recursive algorithm `FindTD` that constructs the decomposition itself given access to the data structure. The decomposition is returned by a pointer to the root bag. The data structure interface will be explained in the following paragraphs, and its implementation is given in Section 7. We refer to Section 2 for a brief, intuitive outline.

The initialization of the data structure takes $O(c^k n)$ time (see Lemma 7.1). The time complexity of `FindTD`, given in Section 4.3, is $O(c^k \cdot n \log n)$.

Input: Connected graph G, $k \in \mathbb{N}$, a set S_0 s.t. $|S_0| \leq 2k+3$ and $G \setminus S_0$ is connected, and a tree decomposition $\mathcal{T}_{\text{apx}}$ with $\mathrm{w}(\mathcal{T}_{\text{apx}}) \leq O(k)$

Output: Tree decomposition of G of width at most $3k+4$ with S_0 as the root bag, or conclusion that $\mathrm{tw}(G) > k$.

Initialize data structure $\mathcal{DS}$ with $G, k, S_0, \mathcal{T}_{\text{apx}}$
return `FindTD`()

Algorithm 2: $\texttt{Compress}_1(G, k, \mathcal{T})$.

4.3 The recursive algorithm `FindTD`

Subroutine `FindTD` works on the graph G with two disjoint vertex sets S and U distinguished. Intuitively, S is small (of size at most $2k+3$) and represents the root bag of the tree decomposition under construction. U in turn, stands for the part of the graph to be decomposed below the bag containing S, and is always one of the connected components of $G \setminus S$. As explained in Section 2, we cannot afford storing U explicitly. Instead, we represent U in the data structure by an arbitrary vertex π (called the *pin*) belonging to it, and implicitly define U to be the connected component of $G \setminus S$ that contains π. Formally, behavior of the subroutine `FindTD` is encapsulated in the following lemma:

Lemma 4.2. *There exists an algorithm that, given access to the data structure $\mathcal{DS}$ in a state such that $|S| \leq 2k+3$, computes a tree decomposition $\mathcal{T}$ of $G[U \cup S]$ of width $\leq 3k+4$ with S as a root bag, or correctly reports that that $\mathrm{tw}(G[U \cup S]) > k$. If the algorithm is run on $S = \emptyset$ and $U = V(G)$, then its running time is $O(c^k \cdot n \log n)$ for some $c \in \mathbb{N}$.*

The data structure is initialized with $S = S_0$ and π set to an arbitrary vertex of $G \setminus S_0$; as we have assumed that $G \setminus S_0$ is connected, this gives $U = V(G) \setminus S_0$ after initialization. Therefore, Lemma 4.2 immediately yields Lemma 2.8.

A gentle introduction to the data structure Before we proceed to the implementation of the subroutine `FindTD`, we give a quick description of the interface of the data structure $\mathcal{DS}$: what kind of queries and updates it supports, and what is the running time of their execution. The details of the data structure implementation will be given in Section 7.

The state of the data structure is, in addition to $G, k, \mathcal{T}$, three subsets of vertices, S, X and F, and the pin π with the restriction that $\pi \notin S$. S and π uniquely imply the set U, defined as the connected component of $G \setminus S$ that contains π. The intuition behind these sets and the pin is the following:

- S is the set that will serve as a root bag for some subtree,
- π is a vertex which indicates the current active component,
- U is the current active component, the connected component of $G \setminus S$ containing π,
- X is a balanced S-separator (of $G[S \cup U]$) and
- F is a set of vertices marking the connected components of $G[S \cup U] \setminus (S \cup X)$ as "finished".

The construction of the data structure $\mathcal{DS}$ is heavily based on the fact that we are provided with some tree decomposition of width $O(k)$. Given this tree decomposition, the data structure can be initialized in $O(c^k \cdot n)$ time for some $c \in \mathbb{N}$. At the moment of initialization we set $S = X = F = \emptyset$ and π to be an arbitrary vertex of G. During the run of the algorithm, the following updates can be performed on the data structure:

- insert/remove a vertex to/from S, X, or F;
- mark/unmark a vertex as a pin π.

All of these updates will be performed in $O(c^k \cdot \log n)$ time for some $c \in \mathbb{N}$.

The data structure provides a number of queries that are used in the subroutine `FindTD`. The running time of each query is $O(c^k \cdot \log n)$ for some $c \in \mathbb{N}$, and in many cases it is actually much smaller. We find it more convenient to explain the needed queries while describing the algorithm itself.

Implementation of `FindTD` The pseudocode of the algorithm `FindTD` is given as Algorithm 3. Its correctness is proven as Claim 4.3, and its time complexity is proven as Claim 4.4. The subroutine is provided with the data structure $\mathcal{DS}$, and the following invariants hold at each time the subroutine is called and exited:

- $S \subseteq V(G)$, $|S| \leq 2k+3$,
- π exists, is unique and $\pi \notin S$,
- $X = F = \emptyset$ and
- the state of the data structure is the same on exit as it was when the function was called.

The latter means that when we return, be it a tree decomposition or $\perp$, the algorithm that called `FindTD` will have S, X, F and π as they were before the call.

We now describe the consecutive steps of the algorithm `FindTD`; the reader is encouraged to follow these steps in the pseudocode, in order to be convinced that all the crucial, potentially expensive computations are performed by calls to the data structure.

First we apply query findSSeparator, which either finds a $\frac{1}{2}$-balanced S-separator in $G[S \cup U]$ of size at most $k+1$, or concludes that $\mathrm{tw}(G) > k$. The running time of this query is $k^{O(1)}$. If no such separator can be found, by Lemma 2.1 we infer that $\mathrm{tw}(G[S \cup U]) > k$ and we can terminate the procedure. Otherwise we are provided with such a separator `sep`, which we add to X in the data structure. Moreover, for a technical reason, we also add the pin π to `sep` (and thus also to X), so we end up with having $|\texttt{sep}| \leq k+2$.

The next step is a loop through the connected components of $G[S \cup U] \setminus (S \cup \texttt{sep})$. This part is performed using the query findNextPin. Query findNextPin, which runs in constant time, either finds an arbitrary vertex u of a connected component of $G[S \cup U] \setminus (S \cup X)$ that does not contain any vertex from F, or concludes that each of these components contains some vertex of F. After finding u, we mark u by putting it to F and proceed further, until all the components are marked. Having achieved this, we have obtained a list `pins`, containing exactly one vertex from each connected component of $G[S \cup U] \setminus (S \cup \texttt{sep})$. We remove all the vertices on this list from F, thus making F again empty.

It is worth mentioning that the query findNextPin not only returns some vertex u of a connected component of $G[S \cup U] \setminus (S \cup \texttt{sep})$ that does not contain any vertex from F, but also provides the size of this component as the second coordinate of the return value. Moreover, the components are being found in decreasing order with respect to sizes. In this algorithm we do not exploit this property, but it will be crucial for the linear-time algorithm.

The set X will no longer be used, so we remove all the vertices of `sep` from X, thus making it again empty. On the other hand, we add all the vertices from `sep` to S. The new set S obtained in this manner will constitute the new bag, of size at most $|S| + |\texttt{sep}| \leq 3k+5$. We are left with computing the tree decompositions for the connected components below this bag, which are pinpointed by vertices stored in the list `pins`.

We iterate through the list `pins` and process the components one by one. For each vertex $u \in$ `pins`, we set u as the new pin by unmarking the old one and marking u. Note that the set U gets redefined and now is the connected component containing considered u. First, we find the neighborhood of U in S. This is done using query findNeighborhood, which in $O(k)$ time returns either this neighborhood, or concludes that its cardinality is larger than $2k+3$. However, as X was a $\frac{1}{2}$-balanced S-separator, it follows that this neighborhood will always be of size at most $2k+3$ (a formal argument is contained in the proof of correctness). We continue with $S \cap N(U)$ as our new S and recursively call `FindTD` in order to decompose the connected component under consideration, with its neighborhood in S as the root bag of the constructed tree decomposition. `FindTD` either provides a decomposition by returning a pointer to its root bag, or concludes that no decomposition can be found. If the latter is the case, we may terminate the algorithm providing a negative answer.

After all the connected components are processed, we merge the obtained tree decompositions. For this, we use the function `build`(S, X, C) which, given sets of vertices S and X and a set of pointers C, constructs two bags $B = S$ and $B' = S \cup X$, makes C the children of B', B' the child of B and returns a pointer to B. This pointer may be returned from the whole subroutine, after doing a clean-up of the data structure.

Invariants Now we show that the stated invariants indeed hold. Initially $S = X = F = \emptyset$ and $\pi \in V(G)$, so clearly the invariants are satisfied. If no S-separator is found, the algorithm returns without changing the data structure and hence the invariants trivially hold in this case. Since both X and F are empty or cleared before return or recursing, $X = F = \emptyset$ holds. Furthermore, as S is reset to old_S (consult Algorithm 3 for the variable names used) and the pin to old_π before returning, it follows that the state of the data structure is reverted upon returning.

The size of $S = \emptyset$ is trivially less than $2k+3$ when initialized. Assume that for some call to `FindTD` we have that $|\text{old}_S| \leq 2k+3$. When recursing, S is the neighborhood of some component C of $G[\text{old}_S \cup U] \setminus (\text{old}_S \cup \texttt{sep})$ (note that we refer to U before resetting the pin). This component is contained in some component C' of $G[\text{old}_S \cup U] \setminus \texttt{sep}$, and all the vertices of old_S adjacent to C must be contained in C'. Since `sep` is a $\frac{1}{2}$-balanced old_S-separator, we know that C' contains at most $\frac{1}{2}|\text{old}_S|$ vertices of old_S. Hence, when recursing we have that $|S| \leq \frac{1}{2}|\text{old}_S| + |\texttt{sep}| = \frac{1}{2}(2k+3) + k + 2 = 2k + \frac{7}{2}$ and, since $|S|$ is an integer, it follows that $|S| \leq 2k+3$.

Finally, we argue that the pin π is never contained in S. When we obtain the elements of `pins` (returned by query findNextPin) we know that $X =$ `sep` and the data structure guarantees that the pins will be from $G[\text{old}_S \cup U] \setminus (\text{old}_S \cup \texttt{sep})$. When recursing, $S = b \subseteq (\text{old}_S \cup \texttt{sep})$ and $\pi \in$ `pins`, so it follows that $\pi \notin S$. Assuming $\pi \notin \text{old}_S$, it follows that π is not in S when returning, and our argument is complete. From here on we will safely assume that the invariants indeed hold.

Correctness

Claim 4.3. *The algorithm* `FindTD` *is correct, that is*

(a) if $\text{tw}(G) \leq k$, `FindTD` *returns a valid tree decomposition of* $G[S \cup U]$ *of width at most* $3k+4$ *and*

(b) if `FindTD` *returns* $\perp$ *then* $\text{tw}(G) > k$.

Proof. We start by proving (b). Suppose the algorithm returns $\perp$. This happens when at some point we are unable to find a balanced S-separator for an induced subgraph $G' = G[S \cup U]$. By Lemma 2.1 the treewidth of G' is more than k. Hence $\text{tw}(G) > k$ as well.

Data: Data structure $\mathcal{DS}$
Output: Tree decomposition of width at most $3k+4$ of $G[S \cup U]$ with S as root bag or conclusion that $\mathrm{tw}(G) > k$.

$\mathrm{old_S} \leftarrow \mathcal{DS}.\mathrm{get}_S()$
$\mathrm{old}_\pi \leftarrow \mathcal{DS}.\mathrm{get}_\pi()$
$\mathtt{sep} \leftarrow \mathcal{DS}.\mathrm{findSSeparator}()$
if $\mathtt{sep} = \bot$ **then**
  return $\bot$ /* safe to return: the state not changed */
end
$\mathcal{DS}.\mathrm{insert}_X(\mathtt{sep})$
$\mathcal{DS}.\mathrm{insert}_X(\pi)$
$\mathtt{pins} \leftarrow \emptyset$
while $(u, l) \leftarrow \mathcal{DS}.\mathrm{findNextPin}() \neq \bot$ **do**
  $\mathtt{pins}.\mathit{append}(u)$
  $\mathcal{DS}.\mathrm{insert}_F(u)$
end
$\mathcal{DS}.\mathrm{clear}_X()$
$\mathcal{DS}.\mathrm{clear}_F()$
$\mathcal{DS}.\mathrm{insert}_S(\mathtt{sep})$
$\mathtt{bags} \leftarrow \emptyset$
for $u \in \mathtt{pins}$ **do**
  $\mathcal{DS}.\mathrm{set}_\pi(u)$
  $\mathtt{bags}.\mathit{append}(\mathcal{DS}.\mathrm{findNeighborhood}())$
end
$\mathrm{children} \leftarrow \emptyset$
for $u, b \in \mathtt{pins}, \mathtt{bags}$ **do**
  $\mathcal{DS}.\mathrm{set}_\pi(u)$
  $\mathcal{DS}.\mathrm{clear}_S()$
  $\mathcal{DS}.\mathrm{insert}_S(b)$
  $\mathrm{children}.\mathit{append}(\mathtt{FindTD}())$
end
$\mathcal{DS}.\mathrm{clear}_S()$
$\mathcal{DS}.\mathrm{insert}_S(\mathrm{old_S})$
$\mathcal{DS}.\mathrm{set}_\pi(\mathrm{old}_\pi)$
if $\bot \in \mathrm{children}$ **then**
  return $\bot$ /* postponed because of rollback of S and π */
end
return $\mathtt{build}(\mathrm{old_S}, \mathtt{sep}, \mathrm{children})$

Algorithm 3: `FindTD`

To show (a) we proceed by induction. In the induction we prove that the algorithm creates a tree decomposition, and we therefore argue that the necessary conditions are satisfied, namely

- the bags have size at most $3k+5$,
- every vertex and every edge is contained in some bag and
- for each $v \in V(G)$ the subtree of bags containing v is connected.

The base case is at the leaf of the obtained tree decomposition, namely when $U \subseteq S \cup \mathtt{sep}$. Then we return a tree decomposition containing two bags, B and B' where $B = \{S\}$ and

$B' = \{S \cup \texttt{sep}\}$. Clearly, every edge and every vertex of $G[S \cup U] = G[S \cup \texttt{sep}]$ is contained in the tree decomposition. Furthermore, since the tree has size two, the connectivity requirement holds and finally, since $|S| \leq 2k+3$ (invariant) and $\texttt{sep} \leq k+2$ it follows that $|S \cup \texttt{sep}| \leq 3k+5$. Note that due to the definition of the base case, the algorithm will find no pins and hence it will not recurse further.

The induction step is as follows. Since $U \not\subseteq S \cup \texttt{sep}$, the algorithm have found some pins $\pi_1, \pi_2, \ldots, \pi_d$ and the corresponding components $C_1, C_2, \ldots, C_d$ in $G[S \cup U] \setminus (S \cup \texttt{sep})$. Let $N_i = N(C_i) \cap (S \cup \texttt{sep})$. By the induction hypothesis the algorithm gives us valid tree decompositions $\mathcal{T}_i$ of $G[N_i \cup C_i]$. Note that the root bag of $\mathcal{T}_i$ consists of the vertices in N_i. By the same argument as for the base case, the two bags $B = S$ and $B' = S \cup \texttt{sep}$ that we construct have appropriate sizes.

Let v be an arbitrary vertex of $S \cup U$. If $v \in S \cup \texttt{sep}$, then it is contained in B'. Otherwise there exists a unique i such that $v \in C_i$. It then follows from the induction hypothesis that v is contained in some bag of $\mathcal{T}_i$.

It remains to show that the edge property and the connectivity property hold. Let uv be an arbitrary edge of $G[S \cup U]$. If u and v both are in $S \cup \texttt{sep}$, then the edge is contained in B'. Otherwise, assume without loss of generality that u is in some component C_i. Then u and v are in $N_i \cup C_i$ and hence they are in some bag of $\mathcal{T}_i$ by the induction hypothesis.

Finally, for the connectivity property, let v be some vertex in $S \cup U$. If $v \notin S \cup \texttt{sep}$, then there is a unique i such that $v \in C_i$, hence we can apply the induction hypothesis. So assume that $v \in S \cup \texttt{sep} = B'$. Let A be some bag of $\mathcal{T}$ containing v. We will complete the proof by proving that there is a path of bags containing v from A to B'. If A is B or B', then this follows directly from the construction. Otherwise there exists a unique i such that A is a bag in $\mathcal{T}_i$. Observe that v is in N_i as it is in $S \cup \texttt{sep}$. By the induction hypothesis the bags containing v in $\mathcal{T}_i$ are connected and hence there is a path of bags containing v from A to the root bag R_i of $\mathcal{T}_i$. By construction B' contains v and the bags B' and R_i are adjacent. Hence there is a path of bags containing v from A to B' and as A was arbitrary chosen, this proves that the bags containing v form a connected subtree of the decomposition. This concludes the proof of Claim 4.3. □

Complexity

Claim 4.4. *The invocation of* `FindTD` *in the algorithm* $\texttt{Compress}_1$ *runs in* $O(c^k \cdot n \log n)$ *time for some* $c \in \mathbb{N}$.

Proof. We will prove the complexity of the algorithm by first arguing that the constructed tree decomposition contains at most $2n$ bags. Then we will partition the used running time between the bags, charging each bag with at most $O(c^k \cdot \log n)$ time. It then follows that that `FindTD` runs in $O(c^k \cdot n \log n)$ time.

To bound the number of bags we simply observe that at each recursion step, we add the previous pin to S and create two bags. Since a vertex can only be added to S one time during the entire process, at most $2n$ bags are created.

It remains to charge the bags. For a call $\mathcal{C}$ to `FindTD`, let B and B' be as previously and let R_i be the root bag of $\mathcal{T}_i$. We will charge B' and $R_1, \ldots, R_d$ for the time spent on $\mathcal{C}$. Notice that as R_i will correspond to B in the next recursion step, each bag will only be charged by one call to `FindTD`. We charge B' with everything in $\mathcal{C}$ not executed in the two loops iterating through the components, plus with the last call to findNextPin that returned $\perp$. Since every update and query in the data structure is executed in $O(c^k \cdot \log n)$ time, and there is a constant number of queries charged to B', it follows that B' is charged with $O(c^k \cdot \log n)$ time. For each iteration in one of the loops we consider the corresponding π_i and charge the bag R_i with the time spent on

this iteration. As all the operations in the loops can be performed in $O(c^k \cdot \log n)$ time, each R_i is charged with at most $O(c^k \cdot \log n)$ time. Since our tree decomposition has at most $2n$ bags and each is charged with at most $O(c^k \cdot \log n)$ time, it follows that `FindTD` runs in $O(c^k \cdot n \log n)$ time and the proof is complete. □

5 $O(c^k n \log^{(\alpha)} n)$ 5-approximation algorithm for treewidth

In this section we provide formal details of the proof of Theorem III

Theorem V (Theorem III, restated)**.** *For every $\alpha \in \mathbb{N}$, there exists an algorithm which, given a graph G and an integer k, in $O(c^k \cdot n \log^{(\alpha)} n)$ time for some $c \in \mathbb{N}$ either computes a tree decomposition of G of width at most $5k+4$ or correctly concludes that* $\mathrm{tw}(G) > k$.

In the proof we give a sequence of algorithms $\texttt{Alg}_\alpha$ for $\alpha = 2, 3, \ldots$; $\texttt{Alg}_1$ has been already presented in the previous section. Each $\texttt{Alg}_\alpha$ in fact solves a slightly more general problem than stated in Theorem III, in the same manner as $\texttt{Alg}_1$ solved a more general problem than the one stated in Theorem II. Namely, every algorithm $\texttt{Alg}_\alpha$ gets as input a connected graph G, an integer k and a subset of vertices S_0 such that $|S_0| \leq 4k+3$ and $G \setminus S_0$ is connected, and either concludes that $\mathrm{tw}(G) > k$ or constructs a tree decomposition of width at most $5k+4$ with S_0 as the root bag. The running time of $\texttt{Alg}_\alpha$ is $O(c^k \cdot n \log^{(\alpha)} n)$ for some $c \in \mathbb{N}$; hence, in order to prove Theorem III we can again apply $\texttt{Alg}_\alpha$ to every connected component of G separately, using $S_0 = \emptyset$.

The algorithms $\texttt{Alg}_\alpha$ are constructed inductively; by that we mean that $\texttt{Alg}_\alpha$ will call $\texttt{Alg}_{\alpha-1}$, which again will call $\texttt{Alg}_{\alpha-2}$, and all the way until $\texttt{Alg}_1$, which was given in the previous section. Let us remark that a closer examination of our algorithms in fact shows that the constants c in the bases of the exponents of consecutive algorithms can be bounded by some universal constant. However, of course the constant factor hidden in the O-notation depends on α.

In the following we present a quick outline of what will be given in this section. For $\alpha = 1$, we refer to the previous section, and for $\alpha > 1$, $\texttt{Alg}_\alpha$ and $\texttt{Compress}_\alpha$ are described in this section, in addition to the subroutine `FindPartialTD`.

- $\texttt{Alg}_\alpha$ takes as input a graph G, an integer k and a vertex set S_0 with similar assumptions as in the previous section, and returns a tree decomposition $\mathcal{T}$ of G of width at most $5k+4$ with S_0 as the root bag. The algorithm is almost exactly as $\texttt{Alg}_1$ given as Algorithm 1, except that it uses $\texttt{Compress}_\alpha$ for the compression step.

- $\texttt{Compress}_\alpha$ is an advanced version of $\texttt{Compress}_1$ (see Algorithm 2), it allows S_0 to be of size up to $4k+3$ and gives a tree decomposition of width at most $5k+4$ in time $O(c^k \cdot n \log^{(\alpha)} n)$. It starts by initializing the data structure, and then it calls `FindPartialTD`, which returns a tree decomposition $\mathcal{T}'$ of an induced subgraph $G' \subseteq G$. The properties of G' and $\mathcal{T}'$ are as follows. All the connected components $C_1, \ldots C_p$ of $G \setminus V(G')$ are of size less than $\log n$. Furthermore, for every connected component C_j, the neighborhood $N(C_j)$ in G is contained in a bag of $\mathcal{T}'$. Intuitively, this ensures that we are able to construct a tree decomposition of C_j and attach it to $\mathcal{T}'$ without blowing up the width of $\mathcal{T}'$. More precisely, for every connected component C_j, the algorithm constructs the induced subgraph $G_j = G[C_j \cup N(C_j)]$ and calls $\texttt{Alg}_{\alpha-1}$ on G_j, k, and $N(C_j)$. The size of $N(C_j)$ will be bounded by $4k+3$, making the recursion valid with respect to the invariants of $\texttt{Alg}_\alpha$. If this call returned a tree decomposition $\mathcal{T}_j$ with a root bag $N(C_j)$, we can conveniently attach $\mathcal{T}_j$ to $\mathcal{T}'$; otherwise we conclude that $\mathrm{tw}(G[C_j \cup N(C_j)]) > k$ so $\mathrm{tw}(G) > k$ as well.

- `FindPartialTD` differs from `FindTD` in two ways. First, we use the fact that when enumerating the components separated by the separator using query findNextPin, these components are identified in the descending order of cardinalities. We continue the construction of partial tree decomposition in the identified components only as long as they are of size at least $\log n$, and we terminate the enumeration when we encounter the first smaller component. It follows that all the remaining components are smaller then $\log n$; these remainders are exactly the components $C_1, \ldots C_p$ that are left not decomposed by $\texttt{Alg}_\alpha$, and on which $\texttt{Alg}_{\alpha-1}$ is run.

  The other difference is that the data structure has a new *flag*, `whatsep`, which is set to either u or s and is alternated between calls. If $\texttt{whatsep} = s$, we use the same type of separator as `FindTD` did, namely findSSeparator, but if $\texttt{whatsep} = u$, then we use the (new) query findUSeparator. Query findUSeparator, instead of giving a balanced S-separator, provides a $\frac{8}{9}$-balanced U-separator, that is, a separator that splits the whole set U of vertices to be decomposed in a balanced way. Using the fact that on every second level of the decomposition procedure the whole set of available vertices shrinks by a constant fraction, we may for example observe that the resulting partial tree decomposition will be of logarithmic depth. More importantly, it may be shown that the total number of constructed bags is at most $O(n/\log n)$ and hence we can spend $O(c^k \cdot \log n)$ time constructing each bag and still obtain running time linear in n.

In all the algorithms that follow we assume that the cardinality of the edge set is at most k times the cardinality of the vertex set, because otherwise we may immediately conclude that treewidth of the graph under consideration is larger than k and terminate the algorithm.

5.1 The main procedure $\texttt{Alg}_\alpha$

The procedure $\texttt{Alg}_\alpha$ works exactly as $\texttt{Alg}_1$, with the exception that it applies Lemma 2.7 for parameter $5k+4$ instead of $3k+4$, and calls recursively $\texttt{Alg}_\alpha$ and $\texttt{Compress}_\alpha$ instead of $\texttt{Alg}_1$ and $\texttt{Compress}_1$. The running time analysis is exactly the same, hence we omit it here.

5.2 Compression algorithm

The following lemma explains the behavior of the compression algorithm $\texttt{Compress}_\alpha$.

Lemma 5.1. *For every integer $\alpha \geq 1$ there exists an algorithm, which on input $G, k, S_0, \mathcal{T}_{apx}$, where (i) $S_0 \subseteq V(G)$, $|S_0| \leq 4k+3$, (ii) G and $G \setminus S_0$ are connected, and (iii) $\mathcal{T}_{apx}$ is a tree decomposition of G of width at most $O(k)$, in $O(c^k \cdot n \log^{(\alpha)} n)$ time for some $c \in \mathbb{N}$ either computes a tree decomposition $\mathcal{T}$ of G with $\mathrm{w}(\mathcal{T}) \leq 5k+4$ and S_0 as the root bag, or correctly concludes that $\mathrm{tw}(G) > k$.*

The outline of the algorithm $\texttt{Compress}_\alpha$ for $\alpha > 1$ is given as Algorithm 4. Having initialized the data structure using $\mathcal{T}_{\mathrm{apx}}$, the algorithm asks `FindPartialTD` for a partial tree decomposition $\mathcal{T}'$, and then the goal is to decompose the remaining small components and attach the resulting tree decompositions in appropriate places of $\mathcal{T}'$.

First we traverse $\mathcal{T}'$ in linear time and store information on where each vertex appearing in $\mathcal{T}'$ is forgotten in $\mathcal{T}'$. More precisely, we compute a map $\texttt{forgotten} : V(G) \to V(\mathcal{T}') \cup \{\bot\}$, where for every vertex v of G we either store $\bot$ if it is not contained in $\mathcal{T}'$, or we remember the top-most bag B_i of $\mathcal{T}'$ such that $v \in B_i$ (the connectivity requirement of the tree decomposition ensures that such B_i exists and is unique). The map `forgotten` may be very easily computed via a DFS traversal of the tree decomposition: when accessing a child node i from a parent i',

we put $\texttt{forgotten}(v) = i$ for each $v \in B_i \setminus B_{i'}$. Moreover, for every $v \in B_r$, where r is the root node, we put $\texttt{forgotten}(v) = r$. Clearly, all the vertices not assigned a value in `forgotten` in this manner, are not contained in any bag of $\mathcal{T}'$, and we put value $\perp$ for them. Let W be the set of vertices contained in $\mathcal{T}'$, i.e., $W = \bigcup_{i \in V(\mathcal{T}')} B_i$.

Before we continue, let us show how the map `forgotten` will be used. Suppose that we have some set $Y \subseteq W$, and we have a guarantee that there exists a node i of $\mathcal{T}'$ such that B_i contains the whole Y. We claim the following: then one of the bags associated with $\texttt{forgotten}(v)$ for $v \in Y$ contains the whole Y. Indeed, take the path from i to the root of the tree decomposition $\mathcal{T}'$, and consider the last node i' of this path whose bag contains the whole Y. It follows that $i' = \texttt{forgotten}(v)$ for some $v \in Y$ and $Y \subseteq B_{i'}$, so the claim follows. Hence, we can locate the bag containing Y in $O(k^{O(1)} \cdot |Y|)$ time by testing each of $|Y|$ candidate nodes $\texttt{forgotten}(v)$ for $v \in Y$.

The next step of the algorithm is locating the vertices which has not been accounted for, i.e., those assigned $\perp$ by `forgotten`. The reason each of these vertices has not been put into the tree decomposition, is precisely because the size of its connected component C of $G \setminus W$, is smaller than $\log n$. The neighborhood of this component in G is $N(C)$, and this neighborhood is guaranteed to be of size at most $4k + 3$ and contained in some bag of $\mathcal{T}'$ (a formal proof of this fact will be given when presenting the algorithm `FindPartialTD`, i.e., in Lemma 5.2).

Let $C_1, C_2, \ldots, C_p$ be all the connected components of $G \setminus W$, i.e., the connected components *outside* the obtained partial tree decomposition $\mathcal{T}'$. To complete the partial tree decomposition into a tree decomposition, for every connected component C_j, we construct a graph $G_j = G[C_j \cup N(C_j)]$ that we then aim to decompose. These graphs may be easily identified and constructed in $O(k^{O(1)} \cdot n)$ time using depth-first search as follows.

We iterate through the vertices of G, and for each vertex v such that $\texttt{forgotten}(v) = \perp$ and v was not visited yet, we apply a depth-first search on v to identify its component C. During this depth-first search procedure, we terminate searching and return from a recursive call whenever we encounter a vertex from W. In this manner we identify the whole component C, and all the visited vertices of W constitute exactly $N(C)$. Moreover, the edges traversed while searching are exactly those inside C or between C and $N(C)$. To finish the construction of G_j, it remains to identify edges between vertices of $N(C)$. Recall that we have a guarantee that $N(C) \subseteq W$ and $N(C)$ is contained in some bag of $\mathcal{T}'$. Using the map `forgotten` we can locate some such bag in $O(k^{O(1)})$ time, and in $O(k^{O(1)})$ time check which vertices of $N(C)$ are adjacent in it, thus finishing the construction of G_j. Observe that during the presented procedure we traverse each edge of the graph at most once, and for each of at most n components C we spend $O(k^{O(1)})$ time on examination of $N(C)$. It follows that the total running time is $O(k^{O(1)} \cdot n)$.

Having constructed G_j, we run the algorithm $\texttt{Alg}_{\alpha-1}$ on G_j using $S_0 = N(C_j)$. Note that in this manner we have that both G_j and $G_j \setminus S_0$ are connected, which are requirements of the algorithm $\texttt{Alg}_{\alpha-1}$. If $\texttt{Alg}_{\alpha-1}$ concluded that $\mathrm{tw}(G_j) > k$, then we can consequently answer that $\mathrm{tw}(G) > k$ since G_j is an induced subgraph of G. On the other hand, if $\texttt{Alg}_{\alpha-1}$ provided us with a tree decomposition $\mathcal{T}_j$ of G_j having $N(C_j)$ as the root bag, then we may simply attach this root bag as a child of the bag of $\mathcal{T}'$ that contains the whole $N(C_j)$. Any such bag can be again located in $O(k^{O(1)})$ time using the map `forgotten`.

5.2.1 Correctness and complexity

In this section we prove Lemma 5.1 and Theorem III, and we proceed by induction on α. To this end we will assume the correctness of Lemma 5.2, which will be proved later, and which describes behavior of the subroutine `FindPartialTD`().

For the base case, $\alpha = 1$, we use $\texttt{Compress}_1$ given as Algorithm 2. When its correctness was

Input: Connected graph G, $k \in \mathbb{N}$, a set S_0 s.t. $|S_0| \leq 4k+3$ and $G \setminus S_0$ is connected, and a tree decomposition $\mathcal{T}_{\text{apx}}$ with $\text{w}(\mathcal{T}_{\text{apx}}) \leq O(k)$

Output: Tree decomposition of G of width at most $5k+4$ with S_0 as the root bag, or conclusion that $\text{tw}(G) > k$.

Initialize data structure $\mathcal{DS}$ with $G, k, S_0, \mathcal{T}_{\text{apx}}$
$\mathcal{T}' \leftarrow$ `FindPartialTD`()
if $\mathcal{T}' = \bot$ **then**
  return $\bot$
end
Create the map `forgotten` : $V(G) \to V(\mathcal{T}')$ using a DFS traversal of $\mathcal{T}'$
Construct components $C_1, C_2, \ldots, C_p$ of $G \setminus W$, and graphs $G_j = G[C_j \cup N(C_j)]$ for $j = 1, 2, \ldots, p$
for $j = 1, 2, \ldots, p$ **do**
  $\mathcal{T}_j \leftarrow \texttt{Alg}_{\alpha-1}$ on $G_j, k, N(C_j)$
  if $\mathcal{T}_j = \bot$ **then**
    return $\bot$
  end
  Locate a node i of $\mathcal{T}'$ s.t. $N(C_j) \subseteq B_i$, by checking `forgotten`(v) for each $v \in N(C_j)$
  Attach the root of $\mathcal{T}_j$ as a child of i
end
return $\mathcal{T}'$

Algorithm 4: $\texttt{Compress}_\alpha$

proved we assumed $|S_0| \leq 2k+3$ and this is no longer the case. However, if $\texttt{Alg}_1$ is applied with $|S_0| \leq 4k+3$ it will conclude that $\text{tw}(G) > k$ or give a tree decomposition of width at most $5k+4$. The reason is as follows; Assume that `FindTD` is applied with the invariant $|S| \leq 4k+3$ instead of $2k+3$. By the same argument as in the original proof this invariant will hold, since $\frac{1}{2}(4k+3)+k+2 \leq 4k+3$. The only part of the correctness (and running time analysis) affected by this change is the width of the returned decomposition, and when the algorithm adds the separator to S it creates a bag of size at most $(4k+3)+(k+2) = 5k+5$ and hence our argument for the base case is complete. For the induction step, suppose that the theorem and lemma hold for $\alpha - 1$. We show that $\texttt{Compress}_\alpha$ is correct and runs in $O(c^k \cdot n \log^{(\alpha)} n)$ time. This immediately implies correctness and complexity of $\texttt{Alg}_\alpha$, in the same manner as in Section 4.

To prove correctness of $\texttt{Compress}_\alpha$, suppose that $\mathcal{T}'$ is a valid tree decomposition for some $G' \subseteq G$ that we have obtained from `FindPartialTD`. Observe that if $\mathcal{T}' = \bot$, then $\text{tw}(G) > k$ by Lemma 5.2. Otherwise, let $C_1, \ldots, C_p$ be the connected components of $G \setminus W$, and let $G_j = G[C_j \cup N(C_j)]$ for $j = 1, 2, \ldots, p$. Let $\mathcal{T}_j$ be the tree decompositions obtained from application of the algorithm $\texttt{Alg}_{\alpha-1}$ on graphs G_j. If $\mathcal{T}_j = \bot$ for any j, we infer that $\text{tw}(G_j) > k$ and, consequently, $\text{tw}(G) > k$. Assume then that for all the components we have indeed obtained valid tree decompositions, with $N(C_j)$ as root bags. It can be easily seen that since $N(C_j)$ separates C_j from the rest of G, then attaching the root of $\mathcal{T}_j$ as a child of any bag containing the whole $N(C_j)$ gives a valid tree decomposition; the width of this tree decomposition is the maximum of widths of $\mathcal{T}'$ and $\mathcal{T}_j$, which is at most $5k+3$. Moreover, if we perform this operation for all the components C_j, then all the vertices and edges of the graph will be contained in some bag of the obtained tree decomposition.

We now proceed to the time complexity of $\texttt{Compress}_\alpha$. The first thing done by the algorithm is the initialization of the data structure and running `FindPartialTD` to obtain $\mathcal{T}'$. Application of `FindPartialTD` takes $O(c^k n)$ time by Lemma 5.2, and so does initialization of the data

structure (see Section 7). As discussed, creation of the `forgotten` map and construction of the graphs G_j takes $O(k^{O(1)} \cdot n)$ time.

Now, the algorithm applies $\texttt{Alg}_{\alpha-1}$ to each graph G_j. Let n_j be the number of vertices of G_j. Note that

$$\sum_{j=1}^{p} n_j = \sum_{j=1}^{p} |C_j| + \sum_{j=1}^{p} |N(C_j)| \leq n + p \cdot (4k+3) \leq (5k+3)n.$$

Moreover, as $n_j \leq \log n + (4k+3)$, it follows from concavity of $t \to \log^{(\alpha-1)} t$ that

$$\log^{(\alpha-1)} n_j \leq \log^{(\alpha-1)}(\log n + (4k+3)) \leq \log^{(\alpha)} n + \log^{(\alpha-1)}(4k+3).$$

By the induction hypothesis, the time complexity of $\texttt{Alg}_{\alpha-1}$ on G_j is $O(c^k \cdot n_j \log^{(\alpha-1)} n_j)$, hence we spend $O(c^k \cdot n_j \log^{(\alpha-1)} n_j)$ time for G_j. Attaching each decomposition $\mathcal{T}_j$ to $\mathcal{T}'$ can be done in $O(k^{O(1)})$ time.

Let C_α denote the complexity of $\texttt{Compress}_\alpha$ and A_α the complexity of $\texttt{Alg}_\alpha$. By applying the induction hypothesis we analyze the complexity of $\texttt{Compress}_\alpha$ (we use a constant $c_1 > c$ to hide polynomial factors depending on k):

$$\begin{aligned}
C_\alpha(n,k) &= O(c^k \cdot n) + \sum_{j=1}^{n} A_{\alpha-1}(n_j, k) \\
&= O(c^k \cdot n) + \sum_{j=1}^{p} O(c^k \cdot n_j \log^{(\alpha-1)} n_j) \\
&\leq O(c^k \cdot n) + \sum_{j=1}^{p} O(c^k \cdot n_j (\log^{(\alpha)} n + \log^{(\alpha-1)}(4k+3))) \\
&= O(c_1^k \cdot n) + \sum_{j=1}^{p} O(c^k \cdot n_j \log^{(\alpha)} n) \\
&\leq O(c_1^k \cdot n) + (5k+3)n \cdot O(c^k \cdot \log^{(\alpha)} n) = O(c_1^k \cdot n \log^{(\alpha)} n).
\end{aligned}$$

We conclude that $\texttt{Compress}_\alpha$ is both correct and that it runs in $O(c^k \cdot n \log^{(\alpha)} n)$ time for some $c \in \mathbb{N}$. The correctness and time complexity $O(c^k n \log^{(\alpha)} n)$ of $\texttt{Alg}_\alpha$ follow in the same manner as in the previous section. And hence our induction step is complete and the correctness of Lemma 2.8 and Theorem III follows. The only assumption we made was that of the correctness of Lemma 5.2, which will be given immediately.

5.3 The algorithm `FindPartialTD`

The following lemma describes behavior of the subroutine `FindPartialTD`.

Lemma 5.2. *There exists an algorithm that, given data structure $\mathcal{DS}$ in a state such that $|S| \leq 4k+3$ if $\texttt{whatsep} = s$ or $|S| \leq 3k+2$ if $\texttt{whatsep} = u$, in time $O(c^k n)$ either concludes that $\mathrm{tw}(G[U \cup S]) > k$, or give a tree decomposition $\mathcal{T}'$ of $G' \subseteq G[U \cup S]$ such that*

- *the width of the decomposition is at most $5k+4$ and S is its root bag;*
- *for every connected component C of $G[U \cup S] \setminus V(G')$, the size of the component is less than $\log n$, its neighborhood is of size at most $4k+3$, and there is a bag in the decomposition $\mathcal{T}'$ containing this whole neighborhood.*

The pseudocode of `FindPartialTD` is presented as Algorithm 5. The algorithm proceeds very similarly to the subroutine `FindTD`, given in Section 4. The main differences are the following.

- We alternate usage of findSSeparator and findUSeparator between the levels of the recursion to achieve that the resulting tree decomposition is also balanced. A special flag in the data structure, `whatsep`, that can be set to s or u, denotes whether we are currently about to use findSSeparator or findUSeparator, respectively. When initializing the data structure we set $\texttt{whatsep} = s$, so we start with finding a balanced S-separator.
- When identifying the next components using query findNextPin, we stop when a component of size less than $\log n$ is discovered. The remaining components are left without being decomposed.

The new query findUSeparator, provided that we have the data structure with S and π distinguished, gives a $\frac{8}{9}$-balanced separator of U in $G[U]$ of size at most $k+1$. That is, it returns a subset Y of vertices of U, with cardinality at most $k+1$, such that every connected component of $G[U] \setminus Y$ has at most $\frac{8}{9}|U|$ vertices. If such a separator cannot be found (which is signalized by $\perp$), we may safely conclude that $\mathrm{tw}(G[U]) > k$ and, consequently $\mathrm{tw}(G) > k$. The running time of query findUSeparator is $O(c^k \cdot \log n)$.

We would like to remark that the usage of balanced U-separators make it not necessary to add the pin to the obtained separator. Recall that this was a technical trick that was used in Section 4 to ensure that the total number of bags of the decomposition was linear.

5.3.1 Correctness

The invariants of Algorithm 5 are as for Algorithm 3, except for the size of S, in which case we distinguish whether `whatsep` is s or u. In the case of s the size of S is at most $4k+3$ and for u the size of S is at most $3k+2$.

If $\texttt{whatsep} = u$ then, since $|S| \leq 3k+2$ and we add an U-separator of size at most $k+1$ and make this our new S, the size of the new S will be at most $4k+3$ and we set $\texttt{whatsep} = s$. For every component C on which we recurse, the cardinality of its neighborhood (S at the moment of recursing) is therefore bounded by $4k+3$. So the invariant holds when $\texttt{whatsep} = u$.

We now show that the invariant holds when $\texttt{whatsep} = s$. Now $|\mathrm{old}_S| \leq 4k+3$. We find $\frac{1}{2}$-balanced S-separator `sep` of size at most $k+1$. When recursing, the new S is the neighborhood of some component C of $G[\mathrm{old}_S \cup U] \setminus (\mathrm{old}_S \cup \texttt{sep})$ (note that we refer to U before resetting the pin). This component is contained in some component C' of $G[\mathrm{old}_S \cup U] \setminus \texttt{sep}$, and all the vertices of old_S adjacent to C must be contained in C'. Since `sep` is a $\frac{1}{2}$-balanced old_S-separator, we know that C' contains at most $\frac{1}{2}|\mathrm{old}_S|$ vertices of old_S. Hence, when recursing we have that $|S| \leq \frac{1}{2}|\mathrm{old}_S| + |\texttt{sep}| = \frac{1}{2}(4k+3) + k + 1 = 3k + \frac{5}{2}$ and, since $|S|$ is an integer, it follows that $|S| \leq 3k+2$. Hence, the invariant also hold when $\texttt{whatsep} = s$.

Note that in both the checks we did not assume anything about the size of the component under consideration. Therefore, it also holds for components on which we do not recurse, i.e., those of size at most $\log n$, that the cardinalities of their neighborhoods will be bounded by $4k+3$.

The fact that the constructed partial tree decomposition is a valid tree decomposition of the subgraph induced by vertices contained in it, follows immediately from the construction, similarly as in Section 4. A simple inductive argument also shows that the width of this tree decomposition is at most $5k+4$: at each step of the construction, we add two bags of sizes at most $(4k+3)+(k+1) \leq 5k+5$ to the obtained decompositions of the components, which by inductive hypothesis are of width at most $5k+4$.

Data: Data structure $\mathcal{DS}$
Output: Partial tree decomposition of width at most k of $G[S \cup U]$ with S as root bag or conclusion that $\mathrm{tw}(G) > k$.

```
oldS ← DS.get_S()
old_π ← DS.get_π()
old_w ← DS.whatsep
if DS.whatsep = s then
    sep ← DS.findSSeparator()
    DS.whatsep ← u
else
    sep ← DS.findUSeparator()
    DS.whatsep ← s
end
if sep = ⊥ then
    DS.whatsep ← old_w
    return ⊥        /* safe to return: the state not changed */
end
DS.insert_X(sep)
pins ← ∅
while (u, l) ← DS.findNextPin() ≠ ⊥ and l ≥ log n do
    pins.append(u)
    DS.insert_F(u)
end
DS.clear_X()
DS.clear_F()
DS.insert_S(sep)
bags ← ∅
for u ∈ pins do
    DS.set_π(u)
    bags.append(DS.findNeighborhood())
end
children ← ∅
for u, b ∈ pins, bags do
    DS.set_π(u)
    DS.clear_S()
    DS.insert_S(b)
    children.append(FindPartialTD())
end
DS.whatsep ← old_w
DS.clear_S()
DS.insert_S(oldS)
DS.set_π(old_π)
if ⊥ ∈ children then
    return ⊥        /* postponed because of rollback of S and π */
end
return build(oldS, sep, children)
```

Algorithm 5: `FindPartialTD`

Finally, we show that every connected component of $G[S \cup U] \setminus V(G')$ has size at most $\log n$ and that the neighborhood of each of these connected component is contained in some bag on the partial tree decomposition $\mathcal{T}'$. First, by simply breaking out of the loop shown in Algorithm 5 at the point we get a pair (π, l) such that $l < \log n$, we are guaranteed that the connected component of $G[S \cup U] \setminus \texttt{sep}$ containing π has size less than $\log n$, and so does every other connected component of $G[S \cup U]$ not containing a vertex from F and which has not been visited by $\mathcal{DS}$.findNextPin(). Furthermore, since immediately before we break out of the loop due to small size we add $S \cup \texttt{sep}$ to a bag, we have ensured that the neighborhood of any such small component is contained in this bag. The bound on the size of this neighborhood has been already argued.

5.3.2 Complexity

Finally, we show that the running time of the algorithm is $O(c^k \cdot n)$. The data structure operations all take time $O(c^k \log n)$ and we get the data structure $\mathcal{DS}$ as input.

The following combinatorial lemma will be helpful to bound the number of bags in the tree decomposition produced by `FindPartialTD`. We aim to show that the tree decomposition $\mathcal{T}'$ contains at most $O(n/\log n)$ bags, so we will use the lemma with $\mu(i) = w_i/\log n$, where i is a node in a tree decomposition $\mathcal{T}'$ and w_i is the number of vertices in $G[U]$ when i is added to $\mathcal{T}'$. Having proven the lemma, we can show that the number of bags is bounded by $O(\mu(r)) = O(n/\log n)$, where r is the root node of $\mathcal{T}'$.

Lemma 5.3. *Let T be a rooted tree with root r. Assume that we are given a measure $\mu : V(T) \to \mathbb{R}$ with the following properties:*

(i) $\mu(v) \geq 1$ for every $v \in V(T)$,

(ii) for every vertex v, let $v_1, v_2, \ldots, v_p$ be its children, we have that $\sum_{i=1}^{p} \mu(v_i) \leq \mu(v)$, and

(iii) there exists a constant $0 < C < 1$ such that for for every two vertices v, v' such that v is a parent of v', it holds that $\mu(v') \leq C \cdot \mu(v)$.

Then $|V(T)| \leq \left(1 + \frac{1}{1-C}\right) \mu(r) - 1$.

Proof. We prove the claim by induction with respect to the size of $V(T)$. If $|V(T)| = 1$, the claim trivially follows from property (i). We proceed to the induction step.

Let $v_1, v_2, \ldots, v_p$ be the children of r and let $T_1, T_2, \ldots, T_p$ be subtrees rooted in $v_1, v_2, \ldots, v_p$, respectively. If we apply the induction hypothesis to trees $T_1, \ldots, T_p$, we infer that for each $i = 1, 2, \ldots, p$ we have that $|V(T_i)| \leq \left(1 + \frac{1}{1-C}\right) \mu(v_i) - 1$. By summing the inequalities we infer that:

$$|V(T)| \quad \leq \quad 1 - p + \left(1 + \frac{1}{1-C}\right) \sum_{i=1}^{p} \mu(v_i).$$

We now consider two cases. Assume first that $p \geq 2$; then:

$$|V(T)| \quad \leq \quad 1 - 2 + \left(1 + \frac{1}{1-C}\right) \sum_{i=1}^{p} \mu(v_i) \leq \left(1 + \frac{1}{1-C}\right) \mu(r) - 1,$$

and we are done. Assume now that $p = 1$; then

$$\begin{aligned} |V(T)| \quad &\leq \quad \left(1 + \frac{1}{1-C}\right) \mu(v_1) \leq C \left(1 + \frac{1}{1-C}\right) \mu(r) \\ &= \quad \left(1 + \frac{1}{1-C}\right) \mu(r) - (2 - C)\mu(r) \leq \left(1 + \frac{1}{1-C}\right) \mu(r) - 1, \end{aligned}$$

and we are done as well. □

We now prove the following claim.

Claim 5.4. *The partial tree decomposition $\mathcal{T}'$ contains at most $42n/\log n$ nodes.*

Proof. Let us partition the set of nodes $V(\mathcal{T}')$ into two subsets. At each recursive call of `FindPartialTD`, we create two nodes: one associated with the bag old_S, and one associated with the bag $\text{old}_S \cup \texttt{sep}$. Let I_{small} be the set of nodes associated with bags old_S, and let I_{large} the the set of remaining bags, associated with bags $\text{old}_S \cup \texttt{sep}$. As bags are always constructed in pairs, it follows that $|I_{\text{small}}| = |I_{\text{large}}| = \frac{1}{2}|V(\mathcal{T}')|$. Therefore, it remains to establish a bound on $|I_{\text{small}}|$.

For a node $i \in V(\mathcal{T}')$, let w_i be the number of vertices strictly below i in the tree decomposition $\mathcal{T}'$, also counting the vertices outside the tree decomposition. Note that by the construction it immediately follows that $w_i \geq \log n$ for each $i \in I_{\text{small}}$.

We now partition I_{small} into three parts: I^s_{small}, $I^{u,\text{int}}_{\text{small}}$, and $I^{u,\text{leaf}}_{\text{small}}$. I^s_{small} consists of all the nodes created in recursive calls where $\texttt{whatsep} = s$. $I^{u,\text{leaf}}_{\text{small}}$ consists of all the nodes created in recursive calls where $\texttt{whatsep} = u$, and moreover the algorithm did not make any more recursive calls to `FindTD` (in other words, all the components turned out to be of size smaller than $\log n$). $I^{u,\text{leaf}}_{\text{small}}$ consists of all the remaining nodes created in recursive calls where $\texttt{whatsep} = u$, that is, such that the algorithm made at least one more call to `FindTD`. We aim at bounding the size of each of the sets I^s_{small}, $I^{u,\text{int}}_{\text{small}}$, and $I^{u,\text{leaf}}_{\text{small}}$ separately.

We first claim that $|I^{u,\text{leaf}}_{\text{small}}| \leq n/\log n$. Indeed, we have that the sets of vertices strictly below nodes of $I^{u,\text{leaf}}_{\text{small}}$ are pairwise disjoint. And since any bag in $I^{i,\text{leaf}}_{\text{small}}$ is a subset of its parent and the recursive call to create the bag was made we know that there is at least $\log n$ vertices below. As their total union is of size at most n, the claim follows.

We now claim that $|I^{u,\text{int}}_{\text{small}}| \leq |I^s_{\text{small}}|$. Indeed, if with every node $i \in I^{u,\text{int}}_{\text{small}}$ we associate any of its grandchild belonging to I^s_{small}, whose existence is guaranteed by the definition of $I^{u,\text{int}}_{\text{small}}$, we obtain an injective map from $I^{u,\text{int}}_{\text{small}}$ into I^s_{small}.

We are left with bounding $|I^s_{\text{small}}|$. For this, we make use of Lemma 5.3. Recall that vertices of I^s_{small} are exactly those that are in levels whose indices are congruent to 1 modulo 4, where the root has level 1; in particular, $r \in I^s_{\text{small}}$. We define a rooted tree T as follows. The vertex set of T is I^s_{small}, and for every two nodes $i, i' \in I^s_{\text{small}}$ such that i' is an ancestor of i exactly 4 levels above (grand-grand-grand-parent), we create an edge between i and i'. It is easy to observe that T created in this manner is a rooted tree, with r as the root.

We can now construct a measure $\mu : V(T) \to \mathbb{R}$ by taking $\mu(i) = w_i/\log n$. Let us check that μ satisfies the assumptions of Lemma 5.3 for $C = \frac{8}{9}$. Property (i) follows from the fact that $w_i \geq \log n$ for every $i \in I_{\text{small}}$. Property (ii) follows from the fact that the parts of the components on which the algorithm recurses below the bags are always pairwise disjoint. Property (iii) follows from the fact that between every pair of parent, child in the tree T we have used a $\frac{8}{9}$-balanced U-separator. Application of Lemma 5.3 immediately gives that $|I^s_{\text{small}}| \leq 10n/\log n$, and hence $|V(\mathcal{T}')| \leq 42n/\log n$. □

To conclude the running time analysis of `FindPartialTD`, we provide a similar charging scheme as in Section 4. More precisely, we charge every node of $\mathcal{T}'$ with $O(c^k \cdot \log n)$ running time; Claim 5.4 ensures us that then the total running time of the algorithm is then $O(c^k \cdot n)$.

Let $B = \text{old}_S$ and $B' = \text{old}_S \cup \texttt{sep}$ be the two bags constructed at some call of `FindPartialTD`. All the operations in this call, apart from the two loops over the components, take $O(c^k \cdot \log n)$ time and are charged to B'. Moreover, the last call of findNextPin, when a component of size

smaller than $\log n$ is discovered, is also charged to B'. As this call takes $O(1)$ time, B' is charged with $O(c^k \cdot \log n)$ time in total.

We now move to examining the time spent while iterating through the loops. Let B_j be the root bag of the decomposition created for graph G_j. We charge B_j with all the operations that were done when processing G_j within the loops. Note that thus every such B_j is charged at most once, and with running time $O(c^k \cdot \log n)$. Summarizing, every bag of $\mathcal{T}'$ is charged with $O(c^k \cdot \log n)$ running time, and we have at most $42n/\log n$ bags, so the total running time of `FindPartialTD` is $O(c^k \cdot n)$.

6 An $O(c^k n)$ 5-approximation algorithm for treewidth

In this section we give the main result of the paper. The algorithm either calls $\texttt{Alg}_\alpha$ for $\alpha = 2$ or a version of Bodlaender [9] applying a table lookup implementation of the dynamic programming algorithm by Bodlaender and Kloks [15], depending on how n and k relate. These techniques combined will give us a 5-approximation algorithm for treewidth in time single exponential in k and linear in n.

Theorem VI (Theorem I, restated)**.** *There exists an algorithm, that given an n-vertex graph G and an integer k, in time $2^{O(k)}n$ either outputs that the treewidth of G is larger than k, or constructs a tree decomposition of G of width at most $5k + 4$.*

As mentioned above, our algorithm distinguishes between two cases. The first case is when n is "sufficiently small" compared to k. By this, we mean that $n \leq 2^{2^{c_0 k^3}}$. The other case is when this is *not* the case. For the first case, we can apply $\texttt{Alg}_2$ and since n is sufficiently small compared to k we can observe that $\log^{(2)} n = k^{O(1)}$, resulting in a $2^{O(k)}n$ time algorithm. For the case when n is large compared to k, we construct a tree automata in time double exponential in k. In this case, double exponential in k is in fact also linear in n. This automaton is then applied on a nice expression tree constructed from our tree decomposition and this results in an algorithm running in time $2^{O(k)}n$.

Lemma 6.1 (Bodlaender and Kloks [15])**.** *There is an algorithm, that given a graph G, an integer k, and a nice tree decomposition of G of width at most ℓ with $O(n)$ bags, either decides that the treewidth of G is more than k, or finds a tree decomposition of G of width at most k in time $O(2^{O(k\ell^2)}n)$.*

Our implementation with table lookup of this result gives the following:

Lemma 6.2. *There is an algorithm, that given a graph G, an integer k, and a nice tree decomposition of G of width at most $\ell = O(k)$ with $O(n)$ bags, either decides that the treewidth of G is more than k, or finds a tree decomposition of G of width at most k, in time $O(2^{2^{c_0 k\ell^2}} + k^{c_1}n)$, for constants c_0 and c_1.*

The proof of Lemma 6.2 will be given later. We first discuss how Lemma 6.2 combined with the results in other sections imply the main result of the paper. First we handle the case when $n \leq 2^{2^{c_0 k^3}}$. We then call $\texttt{Alg}_2$ on each connected component of G separately, with $S = \emptyset$. This algorithm runs in time $O(c'^k \cdot n \log\log n) = O(c'^k n k^3) = O(c^k n)$ time for some constants c and c'.

For the remaining of this section we will assume $n > 2^{2^{c_0 k^3}}$. An inspection of Bodlaender's algorithm [9] shows that it contains the following parts:

- A recursive call to the algorithm is made on a graph with $c_3 n$ vertices, for $c_3 = 1 - \frac{1}{8k^6 + O(k^4)}$.

- The algorithm of Lemma 6.1 is called with $\ell = 2k + 1$.
- Some additional work that uses time linear in n and polynomial in k.

The main difference of our algorithm in the case of "large n" is that we replace the call to the algorithm of Lemma 6.1 with a call to the algorithm of Lemma 6.2, again with $\ell = 2k + 1$. At some point in the recursion, instances will have size less than $2^{2^{c_0 k^3}}$. At that point, we call $\texttt{Alg}_2$ on this instance; with the main difference that at one level higher in the recursion, the algorithm of Lemma 6.2 is called with $\ell = 10k + 9$.

The analysis of the running time is now simple. A call to the algorithm of Lemma 6.2 uses time which is bounded by $O(2^{2^{c_0 k \ell^2}} + k^{c_1} n) = O(n + k^{c_1} n)$, i.e., time linear in n and polynomial in k. The total work of the algorithm is thus bounded by a function $T(n)$ that fulfills $T(n) \leq T(c_3 n) + O(k^{c_4} n) = O(k^{6+c_4} n)$, for constant c_4, i.e., time polynomial in k and linear in n. This proves our main result. What remains for this section is a proof of Lemma 6.2.

6.1 Nice expression trees

The dynamic programming algorithm in [15] is described with help of so called *nice tree decompositions*. As we need to represented a nice tree decomposition as a labeled tree with the label alphabet of size a function of k, we use a slightly changed notion of *labeled nice tree decomposition*. The formalism is quite similar to existing formalisms, e.g., the operations on k-terminal graphs by Borie [16].

A labeled terminal graph is a 4-tuple $G = (V, E, X, f)$, with (V, E) a graph, $X \to V$ a set of *terminals*, and $f : X \to \mathcal{N}$ an injective mapping of the terminals to non-negative integers, which we call *labels*. A k-labeled terminal graph is a labeled terminal graph with the maximum label at most k, i.e., $\max_{x \in X} f(x) \leq k$. Let O_k be the set of the following operations on k-terminal graphs.

Leaf$_\ell$(): gives a k-terminal graph with one vertex v, no edges, with v a terminal with label ℓ

Introduce$_{\ell,S}(G)$: $G = (V, E, X, f)$ is a k-terminal graph, ℓ a non-negative integer, and $S \subseteq \{1, \ldots, k\}$ a set of labels. If there is a terminal vertex in G with label ℓ, then the operation returns G, otherwise it returns the graph, obtained by adding a new vertex v, making v a terminal with label ℓ, and adding edges $\{v, w\}$ for each terminal $w \in X$ with $f(w) \in S$. I.e., we make the new vertex adjacent to each existing terminal whose label is in S.

Forget$_\ell(G)$: Again $G = (V, E, X, f)$ is a k-terminal graph. If there is no vertex $v \in X$ with $f(v) = \ell$, then the operation returns G, otherwise, we 'turn v into a non-terminal, i.e., we return the k-terminal graph $(V, E, X - \{v\}, f')$ for the vertex v with $f(v) = \ell$, and f' is the restriction of f to $X - \{v\}$.

Join(G,H): $G = (V, E, X, f)$ and $H = (W, F, Y, g)$ are k-terminal graphs. If the range of f and g are not equal, then the operation returns G. Otherwise, the result is obtained by taking the disjoint union of the two graphs, and then identifying terminals with the same label.

Note that for given k, O_k is a collection of $k + k \cdot 2^k + k + k^2$ operations. When the treewidth is k, we work with $k + 1$-terminal graphs. The set of operations mimics closely the well known notion of nice tree decompositions (see e.g., [30, 10]).

Proposition 6.3. *Suppose a tree decomposition of G is given of width at most k with m bags. Then, in time, linear in n and polynomial in k, we can construct an expression giving a graph isomorphic to G in terms of operations from O_{k+1} with the length of the expression $O(mk)$.*

Proof. First build with standard methods a nice tree decomposition of G of width k; this has $O(km)$ bags, and $O(m)$ join nodes. Now, construct the graph $H = (V, F)$, with for all $v, w \in V$, $\{v, w\} \in F$, if and only if there is a bag i with $v, w \in X_i$. It is well known that H is a chordal super graph of G with maximum clique size $k + 1$ (see e.g., [10]). Use a greedy linear time algorithm to find an optimal vertex coloring c of H (see [28, Section 4.7].)

Now, we can transform the nice tree decomposition to the expression as follows: each leaf bag that contains a vertex v is replaced by the operation $\text{Leaf}_{c(v)}$, i.e., we label the vertex by its color in H. We can now replace bottom up each bag in the nice tree decomposition by the corresponding operation; as we labeled vertices with the color in H, we have that all vertices in a bag have different colors, which ensures that a Join indeed performs identifications of vertices correctly. Bag sizes are bounded by $k + 1$, so all operations belong to O_{k+1}. □

View the expression as a labeled rooted tree, i.e., each node is labeled with an operation from O_{k+1}; leaves are labeled with a Leaf operation, and binary nodes have a Join-label. To each node of the tree i, we can associate a graph G_i, and the graph G_r associated to the root node r is isomorphic to G. Call such a labeled rooted tree a *nice expression tree* of width k.

6.2 Dynamic programming and finite state tree automata

The discussion in this paragraph holds for all problems invariant under isomorphism,. Note that the treewidth of a graph is also invariant under isomorphisms. We use ideas from the early days of treewidth, see e.g., [26, 2].

A dynamic programming algorithm on nice tree decompositions can be viewed also as a dynamic programming algorithm on a nice expression tree of width k. Suppose that we have a dynamic programming algorithm that computes in bottom-up order for each node of the expression tree a table with at most $r = O(1)$ bits per table, and to compute a table, only the label of the node (type of operation) and the tables of the children of the node are used. We remark that the DP algorithm for treewidth from Bodlaender and Kloks [15] is indeed of this form, if we see k as a fixed constant. Such an algorithm can be seen as a finite state tree automaton: the states of the automaton correspond to the at most $2^r = O(1)$ different tables; the alphabet are the $O(1)$ different labels of tree nodes.

To decide if the treewidth of G is at most k, we first explicitly build this finite state tree automaton, and then execute it on the expression tree. For actually building the corresponding tree decomposition of G of width at most k, if existing, some more work has to be done, which is described later.

6.3 Table lookup implementation of dynamic programming

The algorithm of Bodlaender and Kloks [15] builds for each node in the nice tree decomposition of *table of characteristics*: each characteristic represents the 'essential information' of a tree decomposition of width at most k of the graph associated with the bag.

Inspection of the algorithm [15] easily shows that the number of different characteristics is bounded by $2^{O(k \cdot \ell^2)}$ when we are given an expression tree of width ℓ and want to test if the treewidth is at most k. (See [15, Definition 5.9].)

We now use that we represent the vertices in the bag, i.e., the terminals, by a label from $\{1, \ldots, \ell + 1\}$ if ℓ is the width of the nice expression tree. Thus, we have a set $C_{k,\ell}$ (only depending of k and ℓ) that contains all possible characteristics that belong to a table; each table is just a subset of $C_{k,\ell}$, i.e., an element of $\mathcal{P}(C_{k,\ell})$. I.e., we can view the decision variant of the dynamic programming algorithm of Bodlaender and Kloks as a finite state tree automaton with alphabet $O_{\ell+1}$ and state set $\mathcal{P}(C_{k,\ell})$.

The first step of the algorithm is now to explicitly construct this finite state tree automaton. We can do this as follows. Enumerate all characteristics in $\mathcal{P}(C_{k,\ell})$, and number them $c_1, \ldots, c_s$, $s = 2^{O(k \cdot \ell^2)}$. Enumerate all elements of $\mathcal{P}(C_{k,\ell})$, and number them $t_1, \ldots, t_{s'}$, $s' = 2^{2^{O(k \cdot \ell^2)}}$; store with t_i the elements of its set.

Then, we compute a transition function $F : O_{\ell+1} \times \{1, \ldots, s'\} \times \{1, \ldots, s'\} \to \{1, \ldots, s'\}$. In terms of the finite state automaton view, F gives the state of a node given its symbol and the states of its children. (If a node has less than two children, the third, and possibly the second argument are ignored.) In terms of the DP algorithm, if we have a tree node i with operation $o \in O_{\ell+1}$, and the children of i have tables corresponding to t_α and t_β, then $F(o, \alpha, \beta)$ gives the number of the table obtained for i by the algorithm. To compute one value of F, we just execute one part of the algorithm of Bodlaender and Kloks [15]. Suppose we want to compute $F(o, \alpha, \beta)$. If o is a shift operation, then a simple renaming suffices. Otherwise, build the tables T_α and T_β corresponding to t_α and t_β, and execute the step of the algorithms from [15] for a node with operation o whose children have tables T_α and T_β. (If the node is not binary, we ignore the second and possibly both tables.) Then, lookup what is the index of the resulting table; this is the value of $F(o, \alpha, \beta)$.

We now estimate the time to compute F. We need to compute $O(2^\ell \cdot \ell \cdot s'^2) = O(2^{2^{O(k \cdot \ell^2)}})$ values; each executes one step of the DP algorithm and does a lookup in the table, which is easily seen to be bounded again by $O(2^{2^{O(k \cdot \ell^2)}})$, so the total time to compute F is still bounded by $O(2^{2^{O(k \cdot \ell^2)}})$. To decide if the treewidth of G is at most k, given a nice tree decomposition of width at most ℓ, we thus carry out the following steps:

- Compute F.
- Transform the nice tree decomposition to a nice expression tree of width ℓ.
- Compute bottom-up (e.g., in post-order) for each node i in the expression tree a value v_i, with a node i labeled by operation $o \in O_{\ell+1}$ and children with values v_{j_1}, v_{j_2}, we have $v_i = F(o, v_{j_1}, v_{j_2})$. If v has less than two children, we take some arbitrary argument for the values of missing children. In this way, v_i corresponds to the table that is computed by the DP algorithm of [15].
- If the value v_r for the root of the expression tree corresponds to the empty set, then the treewidth of G is more than k, otherwise the treewidth of G is at most k. (See [15, Section 4.6 and 5.6].)

If our decision algorithm decides that the treewidth of G is more than k, we reject, and we are done. Otherwise, we need to do additional work to construct a tree decomposition of G of width at most k, which is described next.

6.4 Constructing tree decompositions

After the decision algorithm has determined that the treewidth of G is at most k, we need to find a tree decomposition of G of width at most k. Again, the discussion is necessarily not self contained and we refer to details given in [15, Chapter 6].

Basically, each table entry (characteristic) in the table of a join node is the result of a combination of a table entry in the table of the left child and a table entry of the table of the right child. Similarly, for nodes with one child, each table entry is the result of an operation to a table entry in the table of the child node. Leaf nodes represent a graph with one vertex, and we have just one tree decomposition of this graph, and thus one table entry in the table of a leaf node.

If we select a characteristic of the root bag, this recursively defines one characteristic from each table. In the first phase of the construction, we make such a selection. In order to do this, we first pre-compute another function g, that helps to make this selection. g has four arguments: an operation from $O_{\ell+1}$, the index of a characteristic (a number between 1 and s), and the indexes of two states (numbers between 1 and $s' = 2^s$). As value, g yields $\perp$ or a pair of two indexes of characteristics. The intuition is as follows: suppose we have a node i in the nice expression tree labeled with o, an index c_i of a characteristic of a (not yet known) tree decomposition of G_i, and indexes of the tables of the children of i, say t_{j_1} and t_{j_2}. Now, $g(o, c_i, t_{j_1}, t_{j_2})$ should give a pair (c_{j_1}, c_{j_2}) such that c_i is the result of the combination of c_{j_1} and c_{j_2} (in case that o is the join operation) or of the operation as indicated above to c_{j_1} (in case o is an operation with one argument; c_{j_2} can have any value and is ignored). If no such pair exists, the output is $\perp$.

To compute g, we can perform the following steps for each 4-tuple o, c_i, t_{j_1}, t_{j_2}. Let $S_1 \subseteq C_{k,\ell}$ be the set corresponding to t_{j_1}, and $S_2 \subseteq C_{k,\ell}$ be the set corresponding to t_{j_2}. For each $c \in S_1$ and $c' \in S_2$, see if a characteristic c and a characteristic c' can be combined (or, in case of a unary operation, if the relevant operation can be applied to c) to obtain c_1. If we found a pair, we return it; if no combination gives c_1, we return $\perp$. Again, in case of unary operations o, we ignore c'. We do not need g in case o is a leaf operation, and can give any return values in such cases. One can easily see that the computation of g uses again $2^{2^{O(k \cdot \ell^2)}}$ time.

The first step of our construction phase is to build g, as described. After this, we select a characteristic from $C_{k,\ell}$ for each node in the nice expression tree, as follows. As we arrived in this phase, the state of the root bag corresponds to a nonempty set of characteristics, and we take an arbitrary characteristic from this set (e.g., the first one from the list). Now, we select top-down in the expression tree (e.g., in pre-order) a characteristic. Leaf nodes always receive the characteristic of the trivial tree decomposition of a graph with one vertex. In all other cases, if node i has operation o and has selected characteristic c, the left child of i has state t_{j_1} and the right child of i has state t_{j_2} (or, take any number, e.g., 1, if i has only one child, i.e., o is a unary operation), look up the precomputed value of $g(o, c, t_{j+1}, t_{j_2})$. As c is a characteristic in the table that is the result of $F(o, t_{j+1}, t_{j_2})$, $g \neq \perp$, so suppose g is the pair (c', c''). We associate c' as characteristic with the left child of i, and (if i has two children) c'' as characteristic with the right child of i.

At this point, we have associated a characteristic with each node in the nice expression tree. These characteristics are precisely the same as the characteristics that are computed in the constructive phase of the algorithm from [15, Section 6], with the sole difference that we work with labeled terminals instead of the 'names' of the vertices (i.e., in [15], terminals / bag elements are identified as elements from V).

From this point on, we can follow without significant changes the algorithm from [15, Section 6]: bottom-up in the expression tree, we build for each node i, a tree decomposition of G_i whose characteristic is the characteristic we just selected for i, together with a number of pointers from the characteristic to the tree decomposition. Again, the technical details can be found in [15], our only change is that we work with terminals labeled with integers in $\{1, \ldots, \ell + 1\}$ instead of bag vertices.

At the end of this process, we obtain a tree decomposition of the graph associated with the root bag $G_r = G$ whose characteristic belongs to the set corresponding to the state of r. As we only work with characteristics of tree decompositions of width at most k, we obtained a tree decomposition of G of width at most k.

All work we do, except for the pre-computation of the tables of F and g is linear in n and polynomial in k; the time for the pre-computation does not depend on n, and is bounded by $2^{2^{O(k\ell^2)}}$. This ends the description of the proof of Lemma 6.2.

7 A data structure for queries in $O(c^k \log n)$ time

7.1 Overview of the data structure

Assume we are given a tree decomposition $(\{B_i \mid i \in I\}, T = (I, F))$ of G of width $O(k)$. First we turn our tree decomposition into a tree decomposition of depth $O(\log n)$, keeping the width to $t = O(k)$, by the work of Bodlaender and Hagerup [14]. Furthermore, by standard arguments we turn this decomposition into a *nice* tree decomposition in $O(t^{O(1)} \cdot n)$ time, that is, a decomposition of the same width and satisfying following properties:

- All the leaf bags, as well as the root bag, are empty.
- Every node of the tree decomposition is of one of four different types:
  - **Leaf node**: a node i with $B_i = \emptyset$ and no children.
  - **Introduce node**: a node i with exactly one child j such that $B_i = B_j \cup \{v\}$ for some vertex $v \notin B_j$; we say that v is *introduced* in i.
  - **Forget node**: a node i with exactly one child j such that $B_i = B_j \setminus \{v\}$ for some vertex $v \in B_i$; we say that v is *forgotten* in i.
  - **Join node**: a node i with two children j_1, j_2 such that $B_i = B_{j_1} = B_{j_2}$.

The standard technique of turning a tree decomposition into a nice one includes (i) adding paths to the leaves of the decomposition on which we consecutively introduce the vertices of corresponding bags; (ii) adding a path to the root on which we consecutively forget the vertices up to the new root, which is empty; (iii) introducing paths between every non-root node and its parent, on which we first forget all the vertices that need to be forgotten, and then introduce all the vertices that need to be introduced; (iv) substituting every node with $d > 2$ children with a balanced binary tree of $O(\log d)$ depth. It is easy to check that after performing these operations, the tree decomposition has depth at most $O(t \log n)$ and contains at most $O(t \cdot n)$ bags. Moreover, using folklore preprocessing routines, in $O(t^{O(1)} \cdot n)$ time we may prepare the decomposition for algorithmic uses, e.g., for each bag compute and store the list of edges contained in this bag. We omit here the details of this transformation and refer to Kloks [30].

In the data structure, we store a number of tables: three special tables that encode general information on the current state of the graph, and one table per each query. The information stored in the tables reflect some choice of subsets of V, which we will call the *current state of the graph*. More precisely, at each moment the following subsets will be distinguished: S, X, F and a single vertex π, called the pin. The meaning of these sets is described in Section 4. On the data structure we can perform the following updates: adding/removing vertices to S, X, F and marking/unmarking a vertex as a pin. In the following table we gather the tables used by the algorithm, together with an overview of the running times of updates. The meaning of the table entries uses terminology that is described in the following sections.

The following lemma follows from each of the entries in the table below, and will be proved in this section:

Lemma 7.1. *The data structure can be initialized in $O(c^k n)$ time.*

Table	Meaning	Update	Initialization
$P[i]$	Boolean value $\pi \in W_i$	$O(t \cdot \log n)$	$O(t \cdot n)$
$C[i][(S_i, U_i)]$	Connectivity information on U_i^{ext}	$O(3^t \cdot t^{O(1)} \cdot \log n)$	$O(3^t \cdot t^{O(1)} \cdot n)$
$CardU[i][(S_i, U_i)]$	Integer value $\lvert U_i^{\text{ext}} \cap W_i \rvert$	$O(3^t \cdot t^{O(1)} \cdot \log n)$	$O(3^t \cdot t^{O(1)} \cdot n)$
$T_1[i][(S_i, U_i)]$	Table for query findNeighborhood	$O(3^t \cdot k^{O(1)} \cdot \log n)$	$O(3^t \cdot k^{O(1)} \cdot n)$
$T_2[i][(S_i, U_i)][\psi]$	Table for query findSSeparator	$O(9^t \cdot k^{O(1)} \cdot \log n)$	$O(9^t \cdot k^{O(1)} \cdot n)$
$T_3[i][(S_i, U_i, X_i, F_i)]$	Table for query findNextPin	$O(6^t \cdot t^{O(1)} \cdot \log n)$	$O(6^t \cdot t^{O(1)} \cdot n)$
$T_4[i][(S_i, U_i)][\psi]$	Table for query findUSeparator	$O(5^t \cdot k^{O(1)} \cdot \log n)$	$O(5^t \cdot k^{O(1)} \cdot n)$

We now proceed to description of the description of the table P, and then to the two tables C and $CardU$ that handle the important component U. The tables T_1, T_2, T_3 are described together with the description of realization of the corresponding queries. Whenever describing the table, we argue how the table is updated during updates of the data structure, and initialized in the beginning.

7.2 The table P

In the table P, for every node i of the tree decomposition we store a boolean value $P[i]$ equal to $(\pi \in W_i)$. We now show how to maintain the table P when the data structure is updated. The table P needs to be updated whenever the pin π is marked or unmarked. Observe, that the only nodes i for which the information whether $\pi \in W_i$ changed, are the ones on the path from r_π to the root of the tree decomposition. Hence, we can simply follow this path and update the values. As the tree decomposition has depth $O(t \log n)$, this update can be performed in $O(t \cdot \log n)$ time. As when the data structure is initialized, no pin is assigned, P is initially filled with $\perp$.

7.3 Maintaining the important component U

Before we proceed to the description of the queries, let us describe what is the reason of introducing the pin π. During the computation, the algorithm recursively considers smaller parts of the graph, separated from the rest via a small separator: at each step we have distinguished set S and we consider only one connected component U of $G \setminus S$. Unfortunately, we cannot afford recomputing the tree decomposition of U at each recurrence call, or even listing the vertices of U. Therefore we employ a different strategy for identification of U. We will distinguish one vertex of U as a representative pin π, and U can then be defined as the set of vertices reachable from π in $G \setminus S$. Instead of recomputing U at each recursive call we will simply change the pin.

In the tables, for each node i of the tree decomposition we store entries for each possible intersection of U with B_i, and in this manner we are prepared for every possible interaction of U with G_i. In this manner, changing the pin can be done more efficiently. Information needs to be recomputed on two paths to the root in the tree decomposition, corresponding the previous and the next pin, while for subtrees unaffected by the change we do not need to recompute anything as the tables stored there already contain information about the new U as well — as they contain information for *every possible* new U. As the tree decomposition is of logarithmic depth, the update time is logarithmic instead of linear.

We proceed to the formal description. We store the information about U in two special tables: C and $CardU$. As we intuitively explained, tables C and $CardU$ store information on the connectivity behavior in the subtree, for every possible interaction of U with the bag. Formally, for every node of the tree decomposition i we store an entry for every member of the family of *signatures* of the bag B_i. A signature of the bag B_i is a pair (S_i, U_i), such that S_i, U_i are disjoint subsets of B_i. Clearly, the number of signatures is at most $3^{|B_i|}$.

Let i be a node of the tree decomposition. For a signature $\phi = (S_i, U_i)$ of B_i, let $S_i^{\text{ext}} = S_i \cup (S \cap W_i)$ and U_i^{ext} consists of all the vertices reachable in $G_i \setminus S_i^{\text{ext}}$ from U_i or π, providing that it belongs to W_i. Sets S_i^{ext} and U_i^{ext} are called *extensions* of the signature ϕ; note that given S_i and U_i, the extensions are defined uniquely. We remark here that the definition of extensions depend not only on ϕ but also on the node i; hence, we will talk about extensions of signatures only when the associated node is clear from the context.

We say that signature ϕ of B_i with extensions S_i^{ext} and U_i^{ext} is *valid* if it holds that

(i) $U_i^{\text{ext}} \cap B_i = U_i$,

(ii) if $U_i \neq \emptyset$ and $\pi \in W_i$ (equivalently, $P[i]$ is true), then the component of $G[U_i^{\text{ext}}]$ that contains π contains also at least one vertex of U_i.

Intuitively, invalidity means that ϕ cannot contain consistent information about intersection of U and G_i. The second condition says that we cannot fully forget the component of π, unless the whole U_i^{ext} is already forgotten.

Formally, the following invariant explains what is stored in tables C and $CardU$:

- if ϕ is invalid then $C[i][\phi] = CardU[i][\phi] = \bot$;
- otherwise, $C[i][\phi]$ contains an equivalence relation R consisting of all pairs of vertices $(a, b) \in U_i$ that are connected in $G_i[U_i^{\text{ext}}]$, while $CardU[i][\phi]$ contains $|U_i^{\text{ext}} \cap W_i|$.

Note that in this definition we actually ignore the information about alignment of vertices of B_i to set S, F, X in the current state of the graph: the stored information depends only on the alignment of forgotten vertices and the signature of the bag that overrides the actual information in the current state. In this manner we are prepared for possible changes in the data structure, as after an update some other signature will reflect the current state of the graph. Moreover, it is clear from this definition that during the computation, the alignment of every vertex v in the current state of the graph is being checked only in the single node r_v when this vertex is being forgotten; we use this property heavily to implement the updates efficiently enough.

We now explain how for every node i, values in tables $C[i]$ and $CardU[i]$ can be computed using entries for the children of i. These formulas will be crucial both for implementing updates and initialization. We consider different cases, depending on the type of node i.

Case 1: Leaf node. If i is a leaf node then $C[i][(\emptyset, \emptyset)] = \emptyset$ and $CardU[i][(\emptyset, \emptyset)] = 0$.

Case 2: Introduce node. Let i be a node that introduces vertex v, and j be its only child. Consider some signature $\phi = (S_i, U_i)$ of B_i; we would like to compute $R_i = C[i][\phi]$. Let ϕ' be a natural projection of ϕ onto B_j, that is, $\phi' = (S_i \cap B_j, U_i \cap B_j)$. Let $R_j = C[j][\phi']$. We consider some sub-cases, depending on the alignment of v in ϕ.

Case 2.1: $v \in S_i$**.** If we introduce a vertex from S_i, then it follows that extensions of $U_i = U_j$ are equal. Therefore, we can put $C[i][\phi] = C[j][\phi']$ and $CardU[i][\phi] = CardU[j][\phi']$.

Case 2.2: $v \in U_i$**.** In the beginning we check whether conditions of validity are not violated. First, if v is the only vertex of U_i and $P[i] = \top$, then we simply put $C[i][\phi] = \bot$: condition (ii) of validity is violated. Second, we check whether v is adjacent only to vertices of S_j and U_j; if this is not the case, we put $C[i][\phi] = \bot$ as condition (i) of validity is violated.

If the validity checks are satisfied, we can infer that the extension U_i^{ext} of U_i is extension U_j^{ext} of U_j with v added; this follows from the fact that B_j separates v from W_j, so the only vertices of U_i^{ext} adjacent to v are already belonging to U_j. Now we would like to compute the equivalence relation R_i out of R_j. Observe that R_i should be basically R_j augmented by connections introduced by the new vertex v between its neighbors in B_j. Formally, R_i may be

obtained from R_j by merging equivalence classes of all the neighbors of v from U_j, and adding v to the obtained equivalence class; if v does not have any neighbors in U_j, we put it as a new singleton equivalence class. Clearly, $CardU[i][\phi] = CardU[j][\phi']$.

Case 2.3: $v \in B_i \setminus (S_i \cup U_i)$**.** We first check whether the validity constraints are not violated. As v is separated from W_j by B_j, the only possible violation introduced by v is that v is adjacent to a vertex from U_j. In this situation we put $C[i][\phi] = CardU[i][\phi] = \bot$, and otherwise we can put $C[i][\phi] = C[j][\phi']$ and $CardU[i][\phi] = CardU[j][\phi']$, because extensions of ϕ and ϕ' are equal.

Case 3: Forget node. Let i be a node that forgets vertex w, and j be its only child. Consider some signature $\phi = (S_i, U_i)$ of B_i and define extensions S_i^{ext}, U_i^{ext} for this signature. Observe that there is at most one valid signature $\phi' = (S_j, U_j)$ of B_j for which $S_j^{\text{ext}} = S_i^{\text{ext}}$ and $U_j^{\text{ext}} = U_i^{\text{ext}}$, and this signature is simply ϕ with w added possibly to S_i or U_i, depending whether it belongs to S_i^{ext} or U_i^{ext}: the three candidates are $\phi_S = (S_i \cup \{w\}, U_i)$, $\phi_U = (S_i, U_i \cup \{w\})$ and $\phi_0 = (S_i, U_i)$. Moreover, if ϕ is valid then so is ϕ'. Formally, in the following manner we can define signature ϕ', or conclude that ϕ is invalid:

- if $w \in S$, then $\phi' = \phi_S$;
- otherwise, if $w = \pi$ then $\phi' = \phi_U$;
- otherwise, we look into entries $C[j][\phi_U]$ and $C[j][\phi_0]$. If
  - (i) $C[j][\phi_U] = C[j][\phi_0] = \bot$ then ϕ is invalid, and we put $C[i][\phi] = CardU[i][\phi] = \bot$;
  - (ii) if $C[j][\phi_U] = \bot$ or $C[j][\phi_0] = \bot$, we take $\phi' = \phi_0$ or $\phi' = \phi_U$, respectively;
  - (iii) if $C[j][\phi_U] \neq \bot$ and $C[j][\phi_0] \neq \bot$, it follows that w must be a member of a component of $G_i \setminus S_i^{\text{ext}}$ that is fully contained in W_i and does not contain π. Hence we take $\phi' = \phi_0$.

The last point is in fact a check whether $w \in U_i^{\text{ext}}$: whether w is connected to a vertex from U_i in G_i, can be looked up in table $C[j]$ by adding or not adding w to U_i, and checking the stored connectivity information. If $w \in S_i^{\text{ext}}$ or $w \in U_i^{\text{ext}}$, we should be using the information for the signature with S_i or U_i updated with w, and otherwise we do not need to add w anywhere.

As we argued before, if ϕ is valid then so does ϕ', hence if $C[j][\phi'] = \bot$ then we can take $C[i][\phi] = CardU[i][\phi] = \bot$. On the other hand, if ϕ' is valid, then the only possibility for ϕ to be invalid is when condition (ii) cease to be satisfied. This could happen only if $\phi' = \phi_U$ and w is in a singleton equivalence class of $C[j][\phi']$ (note that then the connected component corresponding to this class needs to necessarily contain π, as otherwise we would have $\phi' = \phi_0$). Therefore, if this is the case, we put $C[i][\phi] = CardU[i][\phi] = \bot$, and otherwise we conclude that ϕ is valid and move to defining $C[i][\phi]$ and $CardU[i][\phi]$.

Let now $R_j = C[j][\phi']$. As extensions of ϕ' and ϕ are equal, it follows directly from the maintained invariant that R_i is equal to R_j with w removed from its equivalence class. Moreover, $CardU[i][\phi]$ is equal to $CardU[j][\phi']$, possibly incremented by 1 if we concluded that $\phi' = \phi_U$.

Case 4: Join node. Let i be a join node and j_1, j_2 be its two children. Consider some signature $\phi = (S_i, U_i)$ of B_i. Let $\phi_1 = (S_i, U_i)$ be a signature of B_{j_1} and $\phi_2 = (S_i, U_i)$ be a signature of B_{j_2}. From the maintained invariant it follows that $C[i][\phi]$ is a minimum transitive closure of $C[j_1][\phi_1] \cup C[j_2][\phi_2]$, or $\bot$ if any of these entries contains $\bot$. Similarly, $CardU[i][\phi] = CardU[j_1][\phi_1] + CardU[j_2][\phi_2]$.

We now explain how to update tables C and $CardU$ in $O(3^t \cdot t^{O(1)} \cdot \log n)$ time. We perform a similar strategy as with table P: whenever some vertex v is included or removed from S, or

marked or unmarked as a pin, we follow the path from r_v to the root and fully recompute the whole tables $C, CardU$ in the traversed nodes using the formulas presented above. At each step we recompute the table for some node using the tables of its children; these tables are up to date since they did not need an update at all, or were updated in the previous step. Observe that since the alignment of v in the current state of the graph is accessed only in computation for r_v, the path from r_v to the root of the decomposition consists of all the nodes for which the tables should be recomputed. Note also that when marking or unmarking the pin π, we must first update P and then C and $CardU$. The update takes $O(3^t \cdot t^{O(1)} \cdot \log n)$ time: re-computation of each table takes $O(3^t \cdot t^{O(1)})$ time, and we perform $O(t \log n)$ re-computations as the tree decomposition has depth $O(t \log n)$.

Similarly, tables C and $CardU$ can be initialized in $O(3^t \cdot t^{O(1)} \cdot n)$ time by processing the tree in a bottom-up manner: for each node of the tree decomposition, in $O(3^t \cdot t^{O(1)})$ time we compute its table based on the tables of the children, which were computed before.

7.4 Queries

In our data structure we store one table per each query. When describing every query, we first introduce the formal invariant on storage of table's entry, and how this stored information can be computed based on the entries for children. We then shortly discuss performing updates and initialization of the tables, as they are in all the cases based on the same principle as with tables C and $CardU$. Queries themselves can be performed by reading a single entry of the data structure, with the exception of query findUSeparator, whose implementation is more complex.

7.4.1 Query findNeighborhood

We begin the description of the queries with the simplest one, namely findNeighborhood. This query lists all the vertices of S that are adjacent to U. In the algorithm we have an implicit bound on the size of this neighborhood, which we can use to cut the computation when the accumulated list grows too long. We use ℓ to denote this bound; in our case we have that $\ell = O(k)$.

findNeighborhood
Output: A list of vertices of $N(U) \cap S$, or marker '$\boxtimes$' if their number is larger than ℓ.
Time: $O(\ell)$

Let i be a node of the tree decomposition, let $\phi = (S_i, U_i)$ be a signature of B_i, and let $U_i^{\text{ext}}, S_i^{\text{ext}}$ be extensions of this signature. In entry $T_1[i][\phi]$ we store the following:

- if ϕ is invalid then $T_1[i][\phi] = \perp$;
- otherwise $T_1[i][\phi]$ stores the list of elements of $N(U_i^{\text{ext}}) \cap S_i^{\text{ext}}$ if there is at most ℓ of them, and $\boxtimes$ if there is more of them.

Note that the information whether ϕ is invalid can be looked up in table C. The return value of the query is stored in $T[r][(\emptyset, \emptyset)]$.

We now present how to compute entries of table T_1 for every node i depending on the entries of children of i. We consider different cases, depending of the type of node i. For every case, we consider only signatures that are valid, as for the invalid ones we just put value $\perp$.

Case 1: Leaf node. If i is a leaf node then $T_1[i][(\emptyset, \emptyset)] = \emptyset$.

Case 2: Introduce node. Let i be a node that introduces vertex v, and j be its only child. Consider some signature $\phi = (S_i, U_i)$ of B_i; we would like to compute $T_1[i][\phi] = L_i$. Let ϕ' be

a natural intersection of ϕ with B_j, that is, $\phi' = (S_i \cap B_j, U_i \cap B_j)$. Let $T_1[j][\phi'] = L_j$. We consider some sub-cases, depending on the alignment of v in ϕ.

Case 2.1: $v \in S_i$**.** If we introduce a vertex from S_i, we have that U-extensions of ϕ and ϕ' are equal. It follows that L_i should be simply list L_j with v appended if it is adjacent to any vertex of $U_j = U_i$. Note here that v cannot be adjacent to any vertex of $U_i^{\text{ext}} \setminus U_i$, as B_j separates v from W_j. Hence, we copy the list L_j and append v if it is adjacent to any vertex of U_j and $L_j \neq \boxtimes$. However, if the length of the new list exceeds the ℓ bound, we replace it by $\boxtimes$. Note that copying the list takes $O(\ell)$ time, as its length is bounded by ℓ.

Case 2.2: $v \in U_i$**.** If we introduce a vertex from U_i, then possibly some vertices of S_i gain a neighbor in U_i^{ext}. Note here that vertices of $S_i^{\text{ext}} \setminus S_i$ are not adjacent to the introduced vertex v, as B_j separates v from W_j. Hence, we copy list L_j and append to it all the vertices of S_i that are adjacent to v, but were not yet on L_j. If we exceed the ℓ bound on the length of the list, we put $\boxtimes$ instead. Note that both copying the list and checking whether a vertex of S_i is on it can be done in $O(\ell)$ time, as its length is bounded by ℓ.

Case 2.3: $v \in B_i \setminus (S_i \cup U_i)$**.** In this case extensions of ϕ and ϕ' are equal, so it follows from the invariant that we may simply put $T[i][\phi] = T[j][\phi']$.

Case 3: Forget node. Let i be a node that forgets vertex w, and j be its only child. Consider some signature $\phi = (S_i, U_i)$ of B_i. Define ϕ' in the same manner as in the Forget step in the computation of C. As extensions of ϕ and ϕ' are equal, it follows that $T_1[i][\phi] = T_1[j][\phi']$.

Case 4: Join node. Let i be a join node and j_1, j_2 be its two children. Consider some signature $\phi = (S_i, U_i)$ of B_i. Let $\phi_1 = (S_i, U_i)$ be a signature of B_{j_1} and $\phi_2 = (S_i, U_i)$ be a signature of B_{j_2}. It follows that $T_1[i][\phi]$ should be the merge of lists $T_1[j_1][\phi_1]$ and $T_1[j_2][\phi_2]$, where we remove the duplicates. Of course, if any of these entries contains $\boxtimes$, we simply put $\boxtimes$. Otherwise, the merge can be done in $O(\ell)$ time due to the bound on lengths of $T_1[j_1][\phi_1]$ and $T_1[j_2][\phi_2]$, and if the length of the result exceeds the bound ℓ, we replace it by $\boxtimes$.

Similarly as before, for every addition/removal of vertex v to/from S, or marking/unmarking v as a pin, we can update table T_1 in $O(3^t \cdot k^{O(1)} \cdot \log n)$ time by following the path from r_v to the root and recomputing the tables in the traversed nodes. Also, T_1 can be initialized in $O(3^t \cdot k^{O(1)} \cdot n)$ time by processing the tree decomposition in a bottom-up manner and applying the formula for every node. Note that updating/initializing table T_1 must be performed after updating/initializing tables P and C.

7.4.2 Query findSSeparator

We now move to the next query, namely finding a balanced S-separator. By Lemma 2.1, as $G[U \cup S]$ has treewidth at most k, such a $\frac{1}{2}$-balanced S-separator of size at most $k+1$ always exists. We therefore implement the following query.

findSSeparator
Output: A list of elements of a $\frac{1}{2}$-balanced S-separator of $G[U \cup S]$ of size at most $k+1$, or $\perp$ if no such exists.
Time: $O(t^{O(1)})$

Before we proceed to the implementation of the query, we show how to translate the problem of finding a S-balanced separator into a partitioning problem.

Lemma 7.2 (Lemma 2.9, restated)**.** *Let G be a graph and $S \subseteq V(G)$. Then a set X is a balanced S-separator if and only if there exists a partition (M_1, M_2, M_3) of $V(G) \setminus X$, such that there is no edge between M_i and M_j for $i \neq j$, and $|M_i \cap S| \leq |S|/2$ for $i = 1, 2, 3$.*

The following combinatorial observation is crucial in the proof of Lemma 2.9.

Lemma 7.3. *Let $a_1, a_2, \ldots, a_p$ be non-negative integers such that $\sum_{i=1}^{p} a_i = q$ and $a_i \leq q/2$ for $i = 1, 2, \ldots, p$. Then there exists a partition of these integers into three sets, such that sum of integers in each set is at most $q/2$.*

Proof. Without loss of generality assume that $p > 3$, as otherwise the claim is trivial. We perform a greedy procedure as follows. At each time step of the procedure we have a number of sets, maintaining an invariant that each set is of size at most $q/2$. During the procedure we gradually merge the sets, i.e., we take two sets and replace them with their union. We begin with each integer in its own set. If we arrive at three sets, we end the procedure, thus achieving a feasible partition of the given integers. We therefore need to present how the merging step is performed.

At each step we choose the two sets with smallest sums of elements and merge them (i.e., replace them by their union). As the number of sets is at least 4, the sum of elements of the two chosen ones constitute at most half of the total sum, so after merging them we obtain a set with sum at most $q/2$. Hence, unless the number of sets is at most 3, we can always apply this merging step. □

Proof of Lemma 2.9. One of the implications is trivial: if there is a partition (M_1, M_2, M_3) of $G \setminus X$ with the given properties, then every connected component of $G \setminus X$ must be fully contained either in M_1, M_2, or M_3, hence it contains at most $|S|/2$ vertices of S. We proceed to the second implication.

Assume that X is a balanced S-separator of G and let $C_1, C_2, \ldots, C_p$ be connected components of $G \setminus X$. For $i = 1, 2, \ldots, p$, let $a_p = |S \cap C_i|$. By Lemma 7.3, there exists a partition of integers a_i into three sets, such that the sum of elements of each set is at most $|S|/2$. If we partition vertex sets of components $C_1, C_2, \ldots, C_p$ in the same manner, we obtain a partition (M_1, M_2, M_3) of $V(G) \setminus X$ with postulated properties. □

Lemma 2.9 shows that, when looking for a balanced S-separator, instead of trying to bound the number of elements of S in each connected component of $G[U \cup S] \setminus X$ separately, which could be problematic because of connectivity condition, we can just look for a partition of $G[U \cup S]$ into four sets with prescribed properties that can be checked locally. This suggest the following definition of table T_2.

In table T_2 we store entries for every node i of the tree decomposition, for every signature $\phi = (S_i, U_i)$ of B_i, and for every 8-tuple $\psi = (M_1, M_2, M_3, X, m_1, m_2, m_3, x)$ where

- (M_1, M_2, M_3, X) is a partition of $S_i \cup U_i$,
- m_1, m_2, m_3 are integers between 0 and $|S|/2$,
- and x is an integer between 0 and $k + 1$.

This 8-tuple ψ will be called the *interface*, and intuitively it encodes the interaction of a potential solution with the bag. Observe that the set U is not given in our graph directly but rather via connectivity information stored in table C, so we need to be prepared also for all the possible signatures of the bag; this is the reason why we introduce the interface on top of the signature. Note however, that the number of possible pairs (ϕ, ψ) is at most $9^{|B_i|} \cdot k^{O(1)}$, so for every bag B_i we store $9^{|B_i|} \cdot k^{O(1)}$ entries.

We proceed to the formal definition of what is stored in table T_2. For a fixed signature $\phi = (S_i, U_i)$ of B_i, let $(S_i^{\text{ext}}, U_i^{\text{ext}})$ be its extension, we say that partitioning $(M_1^{\text{ext}}, M_2^{\text{ext}}, M_3^{\text{ext}}, X^{\text{ext}})$ of $S_i^{\text{ext}} \cup U_i^{\text{ext}}$ is an *extension consistent* with interface $\psi = (M_1, M_2, M_3, X, m_1, m_2, m_3, x)$, if:

- $X^{\text{ext}} \cap B_i = X$ and $M_j^{\text{ext}} \cap B_i = M_j$ for $j = 1, 2, 3$;
- there is no edge between vertices of M_j^{ext} and $M_{j'}^{\text{ext}}$ for $j \neq j'$;
- $|X^{\text{ext}} \cap W_i| = x$ and $|M_j^{\text{ext}} \cap W_i| = m_j$ for $j = 1, 2, 3$.

In entry $T_2[i][\phi][\psi]$ we store:

- $\perp$ if ϕ is invalid or no consistent extension of ψ exists;
- otherwise, a list of length x of vertices of $X^{\text{ext}} \cap W_i$ in some consistent extension of ψ.

The query findSSeparatorcan be realized in $O(t^{O(1)})$ time by checking entries in the table T, namely $T[r][(\emptyset, \emptyset)][(\emptyset, \emptyset, \emptyset, \emptyset, m_1, m_2, m_3, x)]$ for all possible values $0 \leq m_j \leq |S|/2$ and $0 \leq x \leq k+1$, and outputting the list contained in any of them that is not equal to $\perp$, or $\perp$ if all of them are equal to $\perp$.

We now present how to compute entries of table T_2 for every node i depending on the entries of children of i. We consider different cases, depending of the type of node i. For every case, we consider only signatures that are valid, as for the invalid ones we just put value $\perp$.

Case 1: Leaf node. If i is a leaf node then $T_2[i][(\emptyset, \emptyset)][(\emptyset, \emptyset, \emptyset, \emptyset, 0, 0, 0, 0)] = \emptyset$, and all the other interfaces are assigned $\perp$.

Case 2: Introduce node. Let i be a node that introduces vertex v, and j be its only child. Consider some signature $\phi = (S_i, U_i)$ of B_i and an interface $\psi = (M_1, M_2, M_3, X, m_1, m_2, m_3, x)$; we would like to compute $T_2[i][\phi][\psi] = L_i$. Let ϕ', ψ' be natural intersections of ϕ, ψ with B_j, respectively, that is, $\phi' = (S_i \cap B_j, U_i \cap B_j)$ and $\psi' = (M_1 \cap B_j, M_2 \cap B_j, M_3 \cap B_j, X \cap B_j, m_1, m_2, m_3, x)$. Let $T_2[j][\phi'][\psi'] = L_j$. We consider some sub-cases, depending on the alignment of v in ϕ and ψ. The cases with v belonging to M_1, M_2 and M_3 are symmetric, so we consider only the case for M_1.

Case 2.1: $v \in X$**.** Note that every extension consistent with interface ψ is an extension consistent with ψ' after trimming to G_j. On the other hand, every extension consistent with ψ' can be extended to an extension consistent with ψ by adding v to the extension of X. Hence, it follows that we can simply take $L_i = L_j$.

Case 2.2: $v \in M_1$**.** Similarly as in the previous case, every extension consistent with interface ψ is an extension consistent with ψ' after trimming to G_j. On the other hand, if we are given an extension consistent with ψ', we can add v to M_1 and make an extension consistent with ψ if and only if v is not adjacent to any vertex of M_2 or M_3; this follows from the fact that B_j separates v from W_j, so the only vertices from M_2^{ext}, M_3^{ext} that v could be possibly adjacent to, lie in B_j. However, if v is adjacent to a vertex of M_2 or M_3, we can obviously put $L_i = \perp$, as there is no extension consistent with ψ: property that there is no edge between M_1^{ext} and $M_3^{\text{ext}} \cup M_3^{\text{ext}}$ is broken already in the bag. Otherwise, by the reasoning above we can put $L_i = L_j$.

Case 2.3: $v \in B_i \setminus (S_i \cup U_i)$**.** Again, in this case we have one-to-one correspondence of extensions consistent with ψ with ψ' after trimming to B_j, so we may simply put $L_i = L_j$.

Case 3: Forget node. Let i be a node that forgets vertex w, and j be its only child. Consider some signature $\phi = (S_i, U_i)$ of B_i, and some interface $\psi = (M_1, M_2, M_3, X, m_1, m_2, m_3, x)$; we would like to compute $T_2[i][\phi][\psi] = L_i$. Let $\phi' = (S_j, U_j)$ be the only extension of signature ϕ to B_j that has the same extension as ϕ; ϕ' can be deduced by looking up which signatures are found valid in table C in the same manner as in the forget step for computation of table C. We consider three cases depending on alignment of w in ϕ':

Case 3.1: $w \notin S_j \cup U_j$**.** If w is not in $S_j \cup U_j$, then it follows that we may put $L_i = T_2[j][\phi'][\psi']$: extensions of ψ consistent with ψ correspond one-to-one to extensions consistent with ψ'.

Case 3.2: $w \in S_j$**.** Assume that there exist some extension $(M_1^{\text{ext}}, M_2^{\text{ext}}, M_3^{\text{ext}}, X^{\text{ext}})$ consistent with ψ. In this extension, vertex w is either in M_1^{ext}, M_2^{ext}, M_3^{ext}, or in X^{ext}. Let us define the corresponding interfaces:

- $\psi_1 = (M_1 \cup \{w\}, M_2, M_3, X, m_1 - 1, m_2, m_3, x)$;
- $\psi_2 = (M_1, M_2 \cup \{w\}, M_3, X, m_1, m_2 - 1, m_3, x)$;
- $\psi_3 = (M_1, M_2, M_3 \cup \{w\}, X, m_1, m_2, m_3 - 1, x)$;
- $\psi_X = (M_1, M_2, M_3, X \cup \{w\}, m_1, m_2, m_3, x - 1)$.

If any of integers $m_1 - 1, m_2 - 1, m_3 - 1, x - 1$ turns out to be negative, we do not consider this interface. It follows that for at least one $\psi' \in \{\psi_1, \psi_2, \psi_3, \psi_X\}$ there must be an extension consistent with ψ': it is just the extension $(M_1^{\text{ext}}, M_2^{\text{ext}}, M_3^{\text{ext}}, X^{\text{ext}})$. On the other hand, any extension consistent with any of interfaces $\psi_1, \psi_2, \psi_3, \psi_X$ is also consistent with ψ. Hence, we may simply put $L_i = T_2[i][\phi'][\psi']$, and append w on the list in case $\psi' = \psi_X$.

Case 3.3: $w \in U_j$**.** We proceed in the same manner as in Case 3.2, with the exception that we do not decrement m_j by 1 in interfaces ψ_j for $j = 1, 2, 3$.

Case 4: Join node. Let i be a join node and j_1, j_2 be its two children. Consider some signature $\phi = (S_i, U_i)$ of B_i, and an interface $\psi = (M_1, M_2, M_3, X, m_1, m_2, m_3, x)$; we would like to compute $T_2[i][\phi][\psi] = L_i$. Let $\phi_1 = (S_i, U_i)$ be a signature of B_{j_1} and $\phi_2 = (S_i, U_i)$ be a signature of B_{j_2}. Assume that there is some extension $(M_1^{\text{ext}}, M_2^{\text{ext}}, M_3^{\text{ext}}, X^{\text{ext}})$ consistent with ψ. Define $m_q^p = |W_{j_p} \cap M_q|$ and $x^p = |W_{j_p} \cap X|$ for $p = 1, 2$ and $q = 1, 2, 3$; note that $m_q^1 + m_q^2 = m_q$ for $q = 1, 2, 3$ and $x^1 + x^2 = x$. It follows that in G_{j_1}, G_{j_2} there are some extensions consistent with $(M_1, M_2, M_3, X, m_1^1, m_2^1, m_3^1, x^1)$ and $(M_1, M_2, M_3, X, m_1^2, m_2^2, m_3^2, x^2)$, respectively — these are simply extension $(M_1^{\text{ext}}, M_2^{\text{ext}}, M_3^{\text{ext}}, X^{\text{ext}})$ intersected with V_i, V_j, respectively. On the other hand, if we have some extensions in G_{j_1}, G_{j_2} consistent with $(M_1, M_2, M_3, X, m_1^1, m_2^1, m_3^1, x^1)$ and $(M_1, M_2, M_3, X, m_1^2, m_2^2, m_3^2, x^2)$ for numbers m_p^q, x^p such that $m_q^1 + m_q^2 = m_q$ for $q = 1, 2, 3$ and $x^1 + x^2 = x$, then the point-wise union of these extensions is an extension consistent with $(M_1, M_2, M_3, X, m_1, m_2, m_3, x)$. It follows that in order to compute L_i, we need to check if for any such choice of m_p^q, x^p we have non-$\perp$ entries in $T_2[j_1][\phi_1][(M_1, M_2, M_3, X, m_1^1, m_2^1, m_3^1, x^1)]$ and $T_2[j_2][\phi_2][(M_1, M_2, M_3, X, m_1^2, m_2^2, m_3^2, x^2)]$. This is the case, we put the union of the lists contained in these entries as L_i, and otherwise we put $\perp$. Note that computing the union of these lists takes $O(k)$ time as their lengths are bounded by k, and there is $O(k^4)$ possible choices of m_p^q, x^p to check.

Similarly as before, for every addition/removal of vertex v to/from S or marking/unmarking v as a pin, we can update table T_2 in $O(9^t \cdot k^{O(1)} \cdot \log n)$ time by following the path from r_v to the root and recomputing the tables in the traversed nodes. Also, T_2 can be initialized in $O(9^t \cdot k^{O(1)} \cdot n)$ time by processing the tree decomposition in a bottom-up manner and applying the formula for every node. Note that updating/initializing table T_2 must be performed after updating/initializing tables P and C.

7.4.3 Query findNextPin

We now proceed to the next query. Recall that at each point, the algorithm maintains the set F of vertices marking components of $G[U \cup S] \setminus (X \cup S)$ that have been already processed. A component is marked as processed when one of its vertices is added to F. Hence, we need

a query that finds the next component to process by returning any of its vertices. As in the linear-time approximation algorithm we need to process the components in decreasing order of sizes, the query in fact provides a vertex of the largest component.

findNextPin
Output: A pair (u, ℓ), where (i) u is a vertex of a component of $G[U \cup S] \setminus (X \cup S)$ that does not contain a vertex from F and is of maximum size among such components, and (ii) ℓ is the size of this component; or, $\perp$ if no such component exists.
Time: $O(1)$

To implement the query we create a table similar to table C, but with entry indexing enriched by subsets of the bag corresponding to possible intersections with X and F. Formally, we store entries for every node i, and for every signature $\phi = (S_i, U_i, X_i, F_i)$, which is a quadruple of subsets of B_i such that (i) $S_i \cap U_i = \emptyset$, (ii) $X_i \subseteq S_i \cup U_i$, (iii) $F_i \subseteq U_i \setminus X_i$. The number of such signatures is equal to $6^{|B_i|}$.

For a signature $\phi = (S_i, U_i, X_i, F_i)$, we say that $(S_i^{\text{ext}}, U_i^{\text{ext}}, X_i^{\text{ext}}, F_i^{\text{ext}})$ is the extension of ϕ if (i) $(S_i^{\text{ext}}, U_i^{\text{ext}})$ is the extension of (S_i, U_i) as in the table C, (ii) $X_i^{\text{ext}} = X_i \cup (W_i \cap X)$ and $F_i^{\text{ext}} = F_i \cup (W_i \cap F)$. We may now state what is stored in entry $T_3[i][(S_i, U_i, X_i, F_i)]$:

- if (S_i, U_i) is invalid then we store $\perp$;
- otherwise we store:
  - an equivalence relation R between vertices of $U_i \setminus X_i$, such that $(v_1, v_2) \in R$ if and only if v_1, v_2 are connected in $G[U_i^{\text{ext}} \setminus X_i^{\text{ext}}]$;
  - for every equivalence class K of R, an integer m_K equal to the number of vertices of the connected component of $G[U_i^{\text{ext}} \setminus X_i^{\text{ext}}]$ containing K, which are contained in W_i, or to $\perp$ if this connected component contains a vertex of F_i^{ext};
  - a pair (u, m), where m is equal to the size of the largest component of $G[U_i^{\text{ext}} \setminus X_i^{\text{ext}}]$ not containing any vertex of F_i^{ext} or U_i, while u is any vertex of this component; if no such component exists, then $(u, m) = (\perp, \perp)$.

Clearly, query findNextPinmay be implemented by outputting the pair (u, m) stored in the entry $T_3[r][(\emptyset, \emptyset, \emptyset, \emptyset)]$, or $\perp$ if this pair is equal to $(\perp, \perp)$.

We now present how to compute entries of table T_3 for every node i depending on the entries of children of i. We consider different cases, depending of the type of node i. For every case, we consider only signatures (S_i, U_i, X_i, F_i) for which (S_i, U_i) is valid, as for the invalid ones we just put value $\perp$.

Case 1: Leaf node. If i is a leaf node then $T_3[i][(\emptyset, \emptyset, \emptyset, \emptyset)] = (\emptyset, \emptyset, (\perp, \perp))$.

Case 2: Introduce node. Let i be a node that introduces vertex v, and j be its only child. Consider some signature $\phi = (S_i, U_i, X_i, F_i)$ of B_i; we would like to compute $T_3[i][\phi] = (R_i, (m_K^i)_{K \in R_i}, (u_i, m_i))$. Let ϕ' be a natural projection of ϕ onto B_j, that is, $\phi' = (S_i \cap B_j, U_i \cap B_j, X_i \cap B_j, F_i \cap B_j)$. Let $T_3[j][\phi'] = (R_j, (m_K^j)_{K \in R_j}, (u_j, m_j))$; note that this entry we know, but entry $T_3[i][\phi]$ we would like to compute. We consider some sub-cases, depending on the alignment of v in ϕ.

Case 2.1: $v \in U_i \setminus (X_i \cup F_i)$**.** If we introduce a vertex from $U_i \setminus (X_i \cup F_i)$, then the extension of ϕ is just the extension of ϕ' plus vertex v added to U_i^{ext}. If we consider the equivalence classes of R_i, then these are equivalence classes of R_j but possibly some of them have been merged because of connections introduced by vertex v. As B_j separates v from W_j, v could only create connections between two vertices from $B_j \cap (U_j \setminus X_j)$. Hence, we can obtain R_i from R_j by

merging all the equivalence classes of vertices of $U_j \setminus X_j$ adjacent to v; the corresponding entry in sequence $(m_K)_{K \in R_i}$ is equal to the sum of entries from the sequence $(m^j_K)_{K \in R_j}$ corresponding to the merged classes. If any of these entries is equal to $\perp$, we put simply $\perp$. If v was not adjacent to any vertex of $U_j \setminus X_j$, we put v in a new equivalence class K with $m_K = 0$. Clearly, we can also put $(u_i, m_i) = (u_j, m_j)$.

Case 2.2: $v \in (U_i \setminus X_i) \cap F_i$**.** We perform in the same manner as in Case 2.2, with the exception that the new entry in sequence $(m_K)_{K \in R_i}$ will be always equal to $\perp$, as the corresponding component contains a vertex from F^{ext}_i.

Case 2.3: $v \in S_i \cup X_i$**.** In this case we can simply put $T_3[i][\phi] = T_3[j][\phi']$ as the extensions of ϕ and ϕ' are the same with the exception of v being included into X^{ext}_i and/or into S^{ext}_i, which does not influence information to be stored in the entry.

Case 2.4: $v \in B_i \setminus (S_i \cup U_i)$**.** In this case we can simply put $T_3[i][\phi] = T_3[j][\phi']$ as the extensions of ϕ and ϕ' are equal.

Case 3: Forget node. Let i be a node that forgets vertex w, and j be its only child. Consider some signature $\phi = (S_i, U_i, X_i, F_i)$ of B_i; we would like to compute $T_3[i][\phi] = (R_i, (m^i_K)_{K \in R_i}, (u_i, m_i))$. Let $(S^{\text{ext}}_i, U^{\text{ext}}_i, X^{\text{ext}}_i, F^{\text{ext}}_i)$ be extension of ϕ. Observe that there is exactly one signature $\phi' = (S_j, U_j, X_j, F_j)$ of B_j with the same extension as ϕ, and this signature is simply ϕ with w added possibly to S_i, U_i, X_i or F_i, depending whether it belongs to S^{ext}_i, U^{ext}_i, X^{ext}_i, or F^{ext}_i. Coloring ϕ' may be defined similarly as in case of forget node for table C; we just need in addition to include w in X^{ext}_i or F^{ext}_i if it belongs to X or F, respectively.

Let $T_3[j][\phi] = (R_j, (m^j_K)_{K \in R_j}, (u_j, m_j))$. As the extensions of ϕ and ϕ' are equal, it follows that we may take R_i equal to R_j with w possibly excluded from its equivalence class. Similarly, for every equivalence class $K \in R_i$ we put m^i_K equal to $m^j_{K'}$, where K' is the corresponding equivalence class of R_j, except the class that contained w which should get the previous number incremented by 1, providing it was not equal to $\perp$. We also put $(u_i, m_i) = (u_j, m_j)$ except the situation, when we forget the last vertex of a component of $G[U^{\text{ext}}_j \setminus X^{\text{ext}}_j]$: this is the case when w is in $U_j \setminus X_j$ and constitutes a singleton equivalence class of R_j. Let then $m^j_{\{w\}}$ be the corresponding entry in sequence $(m^j_K)_{K \in R_j}$. If $m^j_{\{w\}} = \perp$, we simply put $(u_i, m_i) = (u_j, m_j)$. Else, if $(u_j, m_j) = (\perp, \perp)$ or $m^j_{\{w\}} > m_j$, we put $(u_i, m_i) = (w, m^j_{\{w\}})$, and otherwise we put $(u_i, m_i) = (u_j, m_j)$.

Case 4: Join node. Let i be a join node and j_1, j_2 be its two children. Consider some signature $\phi = (S_i, U_i, X_i, F_i)$ of B_i; we would like to compute $T_3[i][\phi] = (R_i, (m^i_K)_{K \in R_i}, (u_i, m_i))$. Let $\phi_1 = (S_i, U_i, X_i, F_i)$ be a signature of B_{j_1} and $\phi_2 = (S_i, U_i, X_i, F_i)$ be a signature of B_{j_2}. Let $T_3[j_1][\phi_1] = (R_{j_1}, (m^{j_1}_K)_{K \in R_{j_1}}, (u_{j_1}, m_{j_1}))$ and $T_3[j_2][\phi_2] = (R_{j_2}, (m^{j_2}_K)_{K \in R_{j_2}}, (u_{j_2}, m_{j_2}))$. Note that equivalence relations R_{j_1} and R_{j_2} are defined on the same set $U_i \setminus X_i$. It follows from the definition of T_3 that we can put:

- R_i to be the minimum transitive closure of $R_{j_1} \cup R_{j_2}$;
- for every equivalence class K of R_i, m^i_K equal to the sum of (i) numbers $m^{j_1}_{K_1}$ for $K_1 \subseteq K$, K_1 being an equivalence class of R_{j_1}, and (ii) numbers $m^{j_2}_{K_2}$ for $K_2 \subseteq K$, K_2 being an equivalence class of R_{j_2}; if any of these numbers is equal to $\perp$, we put $m^i_K = \perp$;
- (u_i, m_i) to be equal to (u_{j_1}, m_{j_1}) or (u_{j_2}, m_{j_2}), depending whether m_{j_1} or m_{j_2} is larger; if any of these numbers is equal to $\perp$, we take the second one, and if both are equal to $\perp$, we put $(u_i, m_i) = (\perp, \perp)$.

Similarly as before, for every addition/removal of vertex v to/from S, to/from X, to/from F, or marking/unmarking v as a pin, we can update table T_3 in $O(6^t \cdot t^{O(1)} \cdot \log n)$ time by following the path from r_v to the root and recomputing the tables in the traversed nodes. Also, T_3 can be initialized in $O(6^t \cdot t^{O(1)} \cdot n)$ time by processing the tree decomposition in a bottom-up manner and applying the formula for every node. Note that updating/initializing table T_3 must be performed after updating/initializing tables P and C.

7.4.4 Query findUSeparator

In this section we implement the last query, needed for the linear-time algorithm; the query is significantly more involved than the previous one. The query specification is as follows:

findUSeparator
Output: A list of elements of a $\frac{8}{9}$-balanced separator of $G[U]$ of size at most $k+1$, or $\perp$ if no such exists.
Time: $O(c^t \cdot k^{O(1)} \cdot \log n)$

Note that Lemma 2.1 guarantees that in fact $G[U]$ contains a $\frac{1}{2}$-balanced separator of size at most $k+1$. Unfortunately, we are not able to find a separator with such a good guarantee on the sizes of the sides; the difficulties are explained in Section 2. Instead, we again make use of the precomputed approximate tree decomposition to find a balanced separator with slightly worse guarantees on the sizes of the sides.

In the following we will also use the notion of a *balanced separation*. For a graph G, we say that a partition (L, X, R) of $V(G)$ is an *α-balanced separation of G*, if there is no edge between L and R, and $|L|, |R| \leq \alpha |V(G)|$. The *order* of a separation is the size of X. Clearly, if (L, X, R) is an α-balanced separation of G, then X is an α-balanced separator of G. By folklore [see the proof of Lemma 2.2] we know that every graph of treewidth at most k has a $\frac{2}{3}$-balanced separation of order at most $k+1$.

Expressing the search for a balanced separator as a maximization problem. Before we start explaining the query implementation, we begin with a few definitions that enable us to express finding a balanced separator as a simple maximization problem.

Definition 7.4. Let G be a graph, and T_L, T_R be disjoint sets of terminals in G. We say that a partition (L, X, R) of $V(G)$ is a *terminal separation of G of order ℓ*, if the following conditions are satisfied:

(i) $T_L \subseteq L$ and $T_R \subseteq R$;

(ii) there is no edge between L and R;

(iii) $|X| \leq \ell$.

We moreover say that (L, X, R) is *left-pushed* (*right-pushed*) if $|L|$ ($|R|$) is maximum among possible terminal separations of order ℓ.

Pushed terminal separations are similar to important separators of Marx [33], and their number for fixed T_L, T_R can be exponential in ℓ. Pushed terminal separations are useful for us because of the following lemma, that enables us to express finding a small balanced separator as a maximization problem, providing that some separator of a reasonable size is given.

Lemma 7.5. *Let G be a graph of treewidth at most k and let (A_1, B, A_2) be some separation of G, such that $|A_1|, |A_2| \leq \frac{3}{4}|V(G)|$. Then there exists a partition (T_L, X_B, T_R) of B and integers*

k_1, k_2 with $k_1+k_2+|X_B| \leq k+1$, such that if G_1, G_2 are $G[A_1 \cup (B \setminus X_B)]$ and $G[A_2 \cup (B \setminus X_B)]$ with terminals T_L, T_R, then

(i) there exist a terminal separations of G_1, G_2 of orders k_1, k_2, respectively;

(ii) for any left-pushed terminal separation (L_1, X_1, R_1) of order k_1 in G_1 and any right-pushed separation (L_2, X_2, R_2) of order k_2 in G_2, the triple $(L_1 \cup T_L \cup L_2, X_1 \cup X_B \cup X_2, R_1 \cup T_R \cup R_2)$ is a terminal separation of G of order at most $k+1$ with $|L_1 \cup T_L \cup L_2|, |R_1 \cup T_R \cup R_2| \leq \frac{7}{8}|V(G)| + \frac{|X|+(k+1)}{2}$.

Proof. As the treewidth of G is at most k, there is a separation (L, X, R) of G such that $|L|, |R| \leq \frac{2}{3}|V(G)|$ and $|X| \leq k+1$ by folklore [see the proof of lemma 2.2]. Let us set $(T_L, X_B, T_R) = (L \cap B, X \cap B, R \cap B)$, $k_1 = |X \cap A_1|$ and $k_2 = |X \cap A_2|$. Observe that $X \cap A_1$ and $X \cap A_2$ are terminal separations in G_1 and G_2 of orders k_1 and k_2, respectively, hence we are done with (i). We proceed to the proof of (ii).

Let us consider sets $L \cap A_1$, $L \cap A_2$, $R \cap A_1$ and $R \cap A_2$. Since (A_1, B, A_2) and (L, X, R) are $\frac{1}{4}$- and $\frac{1}{3}$- balanced separations, respectively, we know that:

- $|L \cap A_1| + |L \cap A_2| + |B| \geq \frac{1}{3}|V(G)| - (k+1)$;
- $|R \cap A_1| + |R \cap A_2| + |B| \geq \frac{1}{3}|V(G)| - (k+1)$;
- $|L \cap A_1| + |R \cap A_1| + (k+1) \geq \frac{1}{4}|V(G)| - |B|$;
- $|L \cap A_2| + |R \cap A_2| + (k+1) \geq \frac{1}{4}|V(G)| - |B|$.

We claim that either $|L \cap A_1|, |R \cap A_2| \geq \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$, or $|L \cap A_2|, |R \cap A_1| \geq \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$. Assume first that $|L \cap A_1| < \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$. Observe that then $|L \cap A_2| \geq \frac{1}{3}|V(G)| - |B| - (k+1) - (\frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}) \geq \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$. Similarly, $|R \cap A_1| \geq \frac{1}{4}|V(G)| - |B| - (k+1) - (\frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}) \geq \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$. The case when $|R \cap A_2| < \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$ is symmetric. Without loss of generality, by possibly flipping separation (L, X, R), assume that $|L \cap A_1|, |R \cap A_2| \geq \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$.

Let (L_1, X_1, R_1) be any left-pushed terminal separation of order k_1 in G_1 and (L_2, X_2, R_2) be any right-pushed terminal separation of order k_2 in G_2. By the definition of being left- and right-pushed, we have that $|L_1 \cap A_1| \geq |L \cap A_1| \geq \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$ and $|R_2 \cap A_2| \geq |R \cap A_2| \geq \frac{1}{8}|V(G)| - \frac{|B|+(k+1)}{2}$. Therefore, we have that $|L_1 \cup T_L \cup L_2| \leq \frac{7}{8}|V(G)| + \frac{|B|+(k+1)}{2}$ and $|L_1 \cup T_L \cup L_2| \leq \frac{7}{8}|V(G)| + \frac{|B|+(k+1)}{2}$. □

The idea of the rest of the implementation is as follows. First, given an approximate tree decomposition of with $O(k)$ in the data structure, in logarithmic time we will find a bag B_{i_0} that splits the component U in a balanced way. This bag will be used as the separator B in the invocation of Lemma 7.5; the right part of the separation will consist of vertices contained in the subtree below B_{i_0}, while the whole rest of the tree will constitute the left part. Lemma 7.5 ensures us that we may find some balanced separator of U by running two maximization dynamic programs: one in the subtree below B_{i_0} to identify a right-pushed separation, and one on the whole rest of the tree to find a left-pushed separation. As in all the other queries, we will store tables of these dynamic programs in the data structure, maintaining them with $O(c^t \log n)$ update times.

Case of a small U At the very beginning of the implementation of the query we read $|U|$, which is stored in the entry $CardU[r][(\emptyset,\emptyset)]$. If it turns out that $|U| < 36(k+t+2) = O(k)$, we perform the following explicit construction. We apply a DFS search from π to identify the whole U; note that this search takes $O(k^2)$ time, as U and S are bounded linearly in k. Then we build subgraph $G[U]$, which again takes $O(k^2)$ time. As this subgraph has $O(k)$ vertices and treewidth at most k, we may find its $\frac{1}{2}$-balanced separator of order at most $k+1$ in c^k time using a brute-force search through all the possible subsets of size at most $k+1$. This separator may be returned as the result of the query. Hence, from now on we assume that $|U| \geq 36(k+t+2)$.

Tracing U We first aim to identify bag B_{i_0} in logarithmic time. The following lemma encapsulates the goal of this subsection. Note that we are not only interested in the bag itself, but also in the intersection of the bag with of S and U (defined as the connected component of $G \setminus S$ containing π). While intersection with S can be trivially computed given the bag, we will need to trace the intersection with U inside the computation.

Lemma 7.6. *There exists an algorithm that, given access to the data structure, in $O(t^{O(1)} \cdot \log n)$ time finds a node i_0 of the tree decomposition such that $|U|/4 \leq |W_{i_0} \cap U| \leq |U|/2$ together with two subsets U_i, S_i of B_{i_0} such that $U_0 = U \cap B_{i_0}$ and $S_0 = S \cap B_{i_0}$.*

Proof. The algorithm keeps track of a node i of the tree decomposition together with a pair of subsets $(U_i, S_i) = (B_i \cap U, B_i \cap S)$ being the intersections of the bag associated to the current node with U and S, respectively. The algorithm starts with the root node r and two empty subsets, and iteratively traverses down the tree keeping an invariant that $CardU[i][(U_i, S_i)] \geq |U|/2$. Whenever we consider a join node i with two sons j_1, j_2, we choose to go down to the node where $CardU[i_t][(U_{j_t}, U_{j_t})]$ is larger among $t = 1, 2$. In this manner, at each step $CardU[i][(U_i, S_i)]$ can be decreased by at most 1 in case of a forget node, or can be at most halved in case of a join node. As $|U| \geq 36(k+t+2)$, it follows that the first node i_0 when the invariant $CardU[i][(U_i, S_i)] \geq |U|/2$ ceases to hold, satisfies $|U|/4 \leq CardU[i_0][(U_{i_0}, S_{i_0})] \leq |U|/2$, and therefore can be safely returned by the algorithm.

It remains to argue how sets (U_i, S_i) can be updated at each step of the traverse down the tree. Updating S_i is trivial as we store an explicit table remembering for each vertex whether it belongs to S. Therefore, now we focus on updating U.

The cases of introduce and join nodes are trivial. If i is an introduce node with son j, then clearly $U_j = U_i \cap B_j$. Similarly, if i is a join node with sons j_1, j_2, then $U_{j_1} = U_{j_2} = U_i$. We are left with the forget node.

Let i be a forget node with son j, and let $B_j = B_i \cup \{w\}$. We have that $U_j = U_i \cup \{w\}$ or $U_j = U_i$, depending whether $w \in U$ or not. This information can be read from the table $C[j]$ as follows:

- if $C[j][(U_i \cup \{w\}, S_j)] = \bot$, then $w \notin U$ and $U_j = U_i$;
- if $C[j][(U_i, S_j)] = \bot$, then $w \in U$ and $U_j = U_i \cup \{w\}$;
- otherwise, both $C[j][(U_i, S_j)]$ and $C[j][(U_i \cup \{w\}, S_j)]$ are not equal to $\bot$; this follows from the fact that at least one of them, corresponding to the correct choice whether $w \in U$ or $w \notin U$, must be not equal to $\bot$. Observe that in this case w is in a singleton equivalence class of $C[j][(U_i \cup \{w\}, S_j)]$, and the connected component of w in the extension of $U_i \cup \{w\}$ cannot contain the pin π. It follows that $w \notin U$ and we take $U_j = U_i$.

Computation at each step of the tree traversal takes $O(t^{O(1)})$ time. As the tree has logarithmic depth, the whole algorithm runs in $O(t^{O(1)} \cdot \log n)$ time. □

Dynamic programming for pushed separators In this subsection we show how to construct dynamic programming tables for finding pushed separators. The implementation resembles that of table T_2, used for balanced S-separators.

In table T_4 we store entries for every node i of the tree decomposition, for every signature $\phi = (S_i, U_i)$ of B_i, and for every 4-tuple $\psi = (L, X, R, x)$, called again the *interface*, where

- (L, X, R) is a partition of U_i,
- x is an integer between 0 and $k+1$.

Again, the intuition is that the interface encodes the interaction of a potential solution with the bag. Note that for every bag B_i we store at most $5^{|B_i|} \cdot (k+2)$ entries.

We proceed to the formal definition of what is stored in table T_4. Let us fix a signature $\phi = (S_i, U_i)$ of B_i, and let $(S_i^{\text{ext}}, U_i^{\text{ext}})$ be its extension. For an interface $\psi = (L, X, R, x)$, we say that a terminal separation $(L^{\text{ext}}, X^{\text{ext}}, R^{\text{ext}})$ in $G[U_i^{\text{ext}}]$ with terminals L, R is an *extension consistent* with interface $\psi = (L, X, R, x)$ if

- $L^{\text{ext}} \cap B_i = L$, $X^{\text{ext}} \cap B_i = X$ and $R^{\text{ext}} \cap B_i = R$;
- $|X^{\text{ext}} \cap W_i| = x$.

Then entry $T_4[i][\phi][\psi]$ contains the pair (r, X_0) where r is the maximum possible $|R^{\text{ext}} \cap W_i|$ among extensions consistent with ψ, and X_0 is the corresponding set $X^{\text{ext}} \cap W_i$ for which this maximum was attained, or $\perp$ if the signature ϕ is invalid or no consistent extension exists.

We now present how to compute entries of table T_4 for every node i depending on the entries of children of i. We consider different cases, depending of the type of node i. For every case, we consider only signatures that are valid, as for the invalid ones we just put value $\perp$.

Case 1: Leaf node. If i is a leaf node then $T_4[i][(\emptyset, \emptyset)][(\emptyset, \emptyset, \emptyset, 0)] = (0, \emptyset)$, and all the other entries are assigned $\perp$.

Case 2: Introduce node. Let i be a node that introduces vertex v, and j be its only child. Consider some signature $\phi = (S_i, U_i)$ of B_i and an interface $\psi = (L, X, R, x)$; we would like to compute $T_4[i][\phi][\psi] = (r_i, X_0^i)$. Let ϕ', ψ' be natural intersections of ϕ, ψ with B_j, respectively, that is, $\phi' = (S_i \cap B_j, U_i \cap B_j)$ and $\psi' = (L \cap B_j, X \cap B_j, R \cap B_j, x)$. Let $T_4[j][\phi'][\psi'] = (r_j, X_0^j)$. We consider some sub-cases, depending on the alignment of v in ϕ and ψ. The cases with v belonging to L and R are symmetric, so we consider only the case for L.

Case 2.1: $v \in X$**.** Note that every extension consistent with interface ψ is an extension consistent with ψ' after trimming to G_j. On the other hand, every extension consistent with ψ' can be extended to an extension consistent with ψ by adding v to the extension of X. Hence, it follows that we can simply take $(r_i, X_0^i) = (r_j, X_0^j)$.

Case 2.2: $v \in L$**.** Similarly as in the previous case, every extension consistent with interface ψ is an extension consistent with ψ' after trimming to G_j. On the other hand, if we are given an extension consistent with ψ', then we can add v to L and make an extension consistent with ψ if and only if v is not adjacent to any vertex of R; this follows from the fact that B_j separates v from W_j, so the only vertices from R^{ext} that v could be possibly adjacent to, lie in B_j. However, if v is adjacent to a vertex of R, then we can obviously put $(r_i, X_0^i) = \perp$ as there is no extension consistent with ψ: property that there is no edge between L and R is broken already in the bag. Otherwise, by the reasoning above we can put $(r_i, X_0^i) = (r_j, X_0^j)$.

Case 2.3: $v \in B_i \setminus U_i$**.** Again, in this case we have one-to-one correspondence of extensions consistent with ψ and extensions consistent with ψ', so we may simply put $(r_i, X_0^i) = (r_j, X_0^j)$.

Case 3: Forget node. Let i be a node that forgets vertex w, and j be its only child. Consider some signature $\phi = (S_i, U_i)$ of B_i, and some interface $\psi = (L, X, R, x)$; we would like to compute $T_4[i][\phi][\psi] = (r_i, X_0^i)$. Let $\phi' = (S_j, U_j)$ be the only the extension of signature ϕ to B_j that has the same extension as ϕ; ϕ' can be deduced by looking up which signatures are found valid in table C in the same manner as in the forget step for computation of table C. We consider two cases depending on alignment of w in ϕ':

Case 3.1: $w \notin U_j$**.** If w is not in U_j, then it follows that we may put $(r_i, X_0^i) = T_4[j][\phi'][\psi']$: extensions consistent with ψ correspond one-to-one to extensions consistent with ψ'.

Case 3.2: $w \in U_j$**.** Assume that there exists some extension $(L^{\text{ext}}, X^{\text{ext}}, R^{\text{ext}})$ consistent with ψ, and assume further that this extension is the one that maximizes $|R^{\text{ext}} \cap W_i|$. In this extension, vertex w is either in L^{ext}, X^{ext}, or in R^{ext}. Let us define the corresponding interfaces:

- $\psi_L = (L \cup \{w\}, X, R, x)$;
- $\psi_X = (L, X \cup \{w\}, R, x - 1)$;
- $\psi_R = (L, X, R \cup \{w\}, x)$.

If $x - 1$ turns out to be negative, we do not consider ψ_X. For $t \in \{L, X, R\}$, let $(r_j, X_0^{j,t}) = T_4[j][\phi'][\psi_t]$. It follows that for at least one $\psi' \in \{\psi_L, \psi_X, \psi_R\}$ there must be an extension consistent with ψ': it is just the extension $(L^{\text{ext}}, X^{\text{ext}}, R^{\text{ext}})$. On the other hand, any extension consistent with any of interfaces ψ_L, ψ_X, ψ_R is also consistent with ψ. Hence, we may simply put $r_i = \max(r_L, r_X, r_R + 1)$, and define X_0^i as the corresponding $X_0^{j,t}$, with possibly w appended if $t = X$. Of course, in this maximum we do not consider the interfaces ψ_t for which $T_4[j][\phi'][\psi_t] = \perp$, and if $T_4[j][\phi'][\psi_t] = \perp$ for all $t \in \{L, X, R\}$, we put $(r_i, X_0^i) = \perp$.

Case 4: Join node. Let i be a join node and j_1, j_2 be its two children. Consider some signature $\phi = (S_i, U_i)$ of B_i, and an interface $\psi = (L, X, R, x)$; we would like to compute $T_4[i][\phi][\psi] = (r_i, X_0^i)$. Let $\phi_1 = (S_i, U_i)$ be a signature of B_{j_1} and $\phi_2 = (S_i, U_i)$ be a signature of B_{j_2}. Assume that there is some extension $(L^{\text{ext}}, X^{\text{ext}}, R^{\text{ext}})$ consistent with ψ, and assume further that this extension is the one that maximizes $|R^{\text{ext}} \cap W_i|$. Define $r^p = |W_{j_p} \cap R|$ and $x^p = |W_{j_p} \cap X|$ for $p = 1, 2$; note that $r^1 + r^2 = r_i$ and $x^1 + x^2 = x$. It follows that in G_{j_1}, G_{j_2} there are some extensions consistent with (L, X, R, x^1) and (L, X, R, x^2), respectively — these are simply extension $(L^{\text{ext}}, X^{\text{ext}}, R^{\text{ext}})$ intersected with V_i, V_j, respectively. On the other hand, if we have some extensions in G_{j_1}, G_{j_2} consistent with (L, X, R, x^1) and (L, X, R, x^2) for numbers x^p such that $x^1 + x^2 = x$, then the point-wise union of these extensions is an extension consistent with (L, X, R, x). It follows that in order to compute (r_i, X_0^i), we need to iterate through choices of x^p such that we have non-$\perp$ entries in $T_2[j_1][\phi_1][(L, X, R, x^1)] = (r_{j_1}^{x^1}, X_0^{j_1, x^1})$ and $T_2[j_2][\phi_2][(L, X, R, x^2)] = (r_{j_1}^{x^1}, X_0^{j_1, x^1})$, choose x^1, x^2 for which $r_{j_1}^{x^1} + r_{j_2}^{x^2}$ is maximum, and define $(r_i, X_0^i) = (r_{j_1}^{x^1} + r_{j_2}^{x^2}, X_0^{j_1, x^1} \cup X_0^{j_2, x^2})$. Of course, if for no choice of x^1, x^2 it is possible, we put $(r_i, X_0^i) = \perp$. Note that computing the union of the sets $X_0^{j_p, x^p}$ for $p = 1, 2$ takes $O(k)$ time as their sizes are bounded by k, and there is $O(t)$ possible choices of x^p to check.

Similarly as before, for every addition/removal of vertex v to/from S or marking/unmarking v as a pin, we can update table T_4 in $O(5^t \cdot k^{O(1)} \cdot \log n)$ time by following the path from r_v to the root and recomputing the tables in the traversed nodes. Also, T_4 can be initialized in $O(5^t \cdot k^{O(1)} \cdot n)$ time by processing the tree decomposition in a bottom-up manner and applying the formula for every node. Note that updating/initializing table T_4 must be performed after updating/initializing tables P and C.

Implementing query findUSeparator We now show how to combine Lemmata 7.5 and 7.6 with the construction of table T_4 to implement query findUSeparator.

The algorithm performs as follows. First, using Lemma 7.6 we identify a node i_0 of the tree decomposition, together with disjoint subsets $(U_{i_0}, S_{i_0}) = (U \cap B_{i_0}, S \cap B_{i_0})$ of B_{i_0}, such that $|U|/4 \leq |W_{i_0} \cap U| \leq |U|/2$. Let $A_2 = W_{i_0}$ and $A_1 = V(G) \setminus V_{i_0}$. Consider separation $(A_1 \cap U, B_{i_0} \cap U, A_2 \cap U)$ of $G[U]$ and apply Lemma 7.5 to it. Let (T_L^0, X_B^0, T_R^0) be the partition of B_{i_0} and k_1^0, k_2^0 be the integers with $k_1^0 + k_2^0 + |X_B^0| \leq k+1$, whose existence is guaranteed by Lemma 7.5.

The algorithm now iterates through all possible partitions (T_L, X_B, T_R) of B_{i_0} and integers k_1, k_2 with $k_1 + k_2 + |X_B| \leq k+1$. We can clearly discard the partitions where there is an edge between T_L and T_R. For a partition (T_L, X_B, T_R), let G_1, G_2 be defined as in Lemma 7.5 for the graph $G[U]$. For a considered tuple $(T_L, X_B, T_R, k_1, k_2)$, we try to find:

(i) a separator of a right-pushed separation of order k_2 in G_2, and the corresponding cardinality of the right side;

(ii) a separator of a left-pushed separation of order k_1 in G_1, and the corresponding cardinality of the left side.

Goal (i) can be achieved simply by reading entries $T_4[i_0][(U_{i_0}, S_{i_0})][(T_L, X_B, T_R, k')]$ for $k' \leq k_2$, and taking the right-pushed separation with the largest right side. We are going to present how goal (ii) is achieved in the following paragraphs, but firstly let us show that achieving both of the goals is sufficient to answer the query.

Observe that if for some (T_L, X_B, T_R) and (k_1, k_2) we obtained both of the separators, denote them X_1, X_2, together with cardinalities of the corresponding sides, then using these cardinalities and precomputed $|U|$ we may check whether $X_1 \cup X_2 \cup X_B$ gives us a $\frac{8}{9}$-separation of $G[U]$. On the other hand, Lemma 7.5 asserts that when (T_L^0, X_B^0, T_R^0) and (k_1^0, k_2^0) are considered, we will find some pushed separations, and moreover any such two separations will yield a $\frac{8}{9}$-separation of $G[U]$. Note that this is indeed the case as the sides of the obtained separation have cardinalities at most $\frac{7}{8}|U| + \frac{(k+1)+(t+1)}{2} = \frac{8}{9}|U| + \frac{k+t+2}{2} - \frac{|U|}{72} \leq \frac{8}{9}|U|$, since $|U| \geq 36(k+t+2)$.

We are left with implementing goal (ii). Let G_1' be G_1 with terminal sets swapped; clearly, left-pushed separations in G_1 correspond to right-pushed separations in G_1'. We implement finding a right-pushed separations in G_1' as follows.

Let $P = (i_0, i_1, \ldots, i_h = r)$ be the path from i_0 to the root r of the tree decomposition. The algorithm traverses the path P, computing tables $D[i_t]$ for consecutive indexes $t = 1, 2, \ldots, t$. The table $D[i_t]$ is indexed by signatures ϕ and interfaces ψ in the same manner as T_4. Formally, for a fixed signature $\phi = (S_{i_t}, U_{i_t})$ of B_{i_t} with extension $(S_{i_t}^{\text{ext}}, U_{i_t}^{\text{ext}})$, we say that this signature is *valid with respect to* (S_{i_0}, U_{i_0}) if it is valid and moreover $(S_{i_0}, U_{i_0}) = (S_{i_t}^{\text{ext}} \cap B_{i_0}, U_{i_t}^{\text{ext}} \cap B_{i_0})$. For an interface ψ we say that separation $(L^{\text{ext}}, X^{\text{ext}}, R^{\text{ext}})$ in $G[U_i^{\text{ext}} \setminus W_{i_0}]$ with terminals L, R is *consistent with* ψ *with respect to* (T_L, X_B, T_R), if it is consistent in the same sense as in table T_4, and moreover $(T_L, X_B, T_R) = (L^{\text{ext}} \cap B_{i_0}, X^{\text{ext}} \cap B_{i_0}, R^{\text{ext}} \cap B_{i_0})$. Then entry $T[i_t][\phi][\psi]$ contains the pair (r, X_0) where r is the maximum possible $|R^{\text{ext}} \cap W_i|$ among extensions consistent with ψ with respect to (T_L, X_B, T_R), and X_0 is the corresponding set $X^{\text{ext}} \cap W_i$ for which this maximum was attained, or $\perp$ if the signature ϕ is invalid with respect to (S_{i_0}, U_{i_0}) or no such consistent extension exists.

The tables $D[i_t]$ can be computed by traversing the path P using the same recurrential formulas as for table T_4. When computing the next $D[i_t]$, we use table $D[i_{t-1}]$ computed in the previous step and possible table T_4 from the second child of i_t. Moreover, as $D[i_0]$ we insert the dummy table $Dummy[\phi][\psi]$ defined as follows:

- $Dummy[(U_{i_0}, S_{i_0})][(T_R, X_B, T_L, 0)] = 0$;

- all the other entries are evaluated to $\perp$.

It is easy to observe that table *Dummy* exactly satisfies the definition of $D[i_0]$. It is also straightforward to check that the recurrential formulas used for computing T_4 can be used in the same manner to compute tables $D[i_t]$ for $t = 1, 2, \ldots, h$. The definition of D and the method of constructing it show, that the values $D[r][(\emptyset, \emptyset)][(\emptyset, \emptyset, \emptyset, x)]$ for $x = 0, 1, \ldots, k$, correspond to exactly right-pushed separations with separators of size exactly x in the graph G'_1: insertion of the dummy table removes A_2 from the graph and forces the separation to respect the terminals in B_{i_0}.

Let us conclude with a summary of the running time of the query. Algorithm of Lemma 7.6 uses $O(t^{O(1)} \cdot \log n)$ time. Then we iterate through at most $O(3^t \cdot k^2)$ tuples (T_L, X_B, T_R) and (k_1, k_2), and for each of them we spend $O(k)$ time on achieving goal (i) and $O(5^t \cdot k^{O(1)} \cdot \log n)$ time on achieving goal (ii). Hence, in total the running time is $O(15^t \cdot k^{O(1)} \cdot \log n)$.

References

[1] Ittai Abraham, Moshe Babaioff, Shaddin Dughmi, and Tim Roughgarden. Combinatorial auctions with restricted complements. In *Proceedings of the 13th ACM Conference on Electronic Commerce, EC 2012*, pages 3–16, 2012.

[2] Karl R. Abrahamson and Michael R. Fellows. Finite automata, bounded treewidth and well-quasiordering. In N. Robertson and P. Seymour, editors, *Proceedings of the AMS Summer Workshop on Graph Minors, Graph Structure Theory, Contemporary Mathematics vol. 147*, pages 539–564. American Mathematical Society, 1993.

[3] Eyal Amir. Approximation algorithms for treewidth. *Algorithmica*, 56:448–479, 2010.

[4] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8:277–284, 1987.

[5] Per Austrin, Toniann Pitassi, and Yu Wu. Inapproximability of treewidth, one-shot pebbling, and related layout problems. In Anupam Gupta, Klaus Jansen, José D. P. Rolim, and Rocco A. Servedio, editors, *Proceedings APPROX 2012 and RANDOM 2012*, volume 7408 of *Lecture Notes in Computer Science*, pages 13–24. Springer Verlag, 2012.

[6] Hans L. Bodlaender. Dynamic programming algorithms on graphs with bounded tree-width. In Timo Lepistö and Arto Salomaa, editors, *Proceedings of the 15th International Colloquium on Automata, Languages and Programming, ICALP'88*, volume 317 of *Lecture Notes in Computer Science*, pages 105–119. Springer Verlag, 1988.

[7] Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proceedings of the 25th Annual Symposium on Theory of Computing, STOC'93*, pages 226–234, 1993.

[8] Hans L. Bodlaender. Dynamic algorithms for graphs with treewidth 2. In Jan van Leeuwen, editor, *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science, WG'93*, volume 790 of *Lecture Notes in Computer Science*, pages 112–124. Springer Verlag, 1994.

[9] Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25:1305–1317, 1996.

[10] Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209:1–45, 1998.

[11] Hans L. Bodlaender, Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Solving weighted and counting variants of connectivity problems parameterized by treewidth deterministically in single exponential time. Report on arXiv 1211.1505, 2012.

[12] Hans L. Bodlaender and Fedor V. Fomin. Equitable colorings of bounded treewidth graphs. *Theoretical Computer Science*, 349:22–30, 2005.

[13] Hans L. Bodlaender, John R. Gilbert, H. Hafsteinsson, and Ton Kloks. Approximating treewidth, pathwidth, frontsize, and minimum elimination tree height. *Journal of Algorithms*, 18:238–255, 1995.

[14] Hans L. Bodlaender and Torben Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. *SIAM Journal on Computing*, 27:1725–1746, 1998.

[15] Hans L. Bodlaender and Ton Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms*, 21:358–402, 1996.

[16] Richard B. Borie. *Recursively Constructed Graph Families*. PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1988.

[17] Shiva Chaudhuri and Christos D. Zaroliagis. Shortest paths in digraphs of small treewidth. Part II: Optimal parallel algorithms. *Theoretical Computer Science*, 203:205–223, 1998.

[18] Shiva Chaudhuri and Christos D. Zaroliagis. Shortest paths in digraphs of small treewidth. Part I: Sequential algorithms. *Algorithmica*, 27:212–226, 2000.

[19] Robert F. Cohen, S. Sairam, Roberto Tamassia, and J. S. Vitter. Dynamic algorithms for optimization problems in bounded tree-width graphs. In Giovanni Rinaldi and Laurence A. Wolsey, editors, *Proceedings of the 3rd Conference on Integer Programming and Combinatorial Optimization, IPCO'93*, pages 99–112, 1993.

[20] Bruno Courcelle. The monadic second-order logic of graphs I: Recognizable sets of finite graphs. *Information and Computation*, 85:12–75, 1990.

[21] Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Fast Hamiltonicity checking via bases of perfect matchings. Report on arXiv 1211.1506, 2012. To appear in Proceedings STOC 2013.

[22] Erik D. Demaine, Fedor V. Fomin, Mohammadtaghi Hajiaghayi, and Dimitrios M. Thilikos. Subexponential parameterized algorithms on graphs of bounded genus and H-minor-free graphs. *Journal of the ACM*, 52:866–893, 2005.

[23] Erik D. Demaine and MohammadTaghi Hajiaghayi. The bidimensionality theory and its algorithmic applications. *The Computer Journal*, 51:292–302, 2008.

[24] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Logspace versions of the theorems of Bodlaender and Courcelle. In *Proceedings of the 51st Annual Symposium on Foundations of Computer Science, FOCS 2010*, pages 143–152, 2010.

[25] Uriel Feige, MohammadTaghi Hajiaghayi, and James R. Lee. Improved approximation algorithms for minimum weight vertex separators. *SIAM Journal on Computing*, 38:629–657, 2008.

[26] Michael R. Fellows and Michael A. Langston. An analogue of the Myhill-Nerode theorem and its use in computing finite-basis characterizations. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science, FOCS'89*, pages 520–525, 1989.

[27] Daniel Gildea. Grammar factorization by tree decomposition. *Computational Linguistics*, 37:231–248, 2011.

[28] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

[29] Torben Hagerup. Dynamic algorithms for graphs of bounded treewidth. *Algorithmica*, 27:292–315, 2000.

[30] Ton Kloks. *Treewidth. Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1994.

[31] Arie M. C. A. Koster, Stan P. M. van Hoesel, and Antoon W. J. Kolen. Solving partial constraint satisfaction problems with tree decomposition. *Networks*, 40(3):170–180, 2002.

[32] Jens Lagergren. Efficient parallel algorithms for graphs of bounded tree-width. *Journal of Algorithms*, 20:20–44, 1996.

[33] Dániel Marx. Parameterized graph separation problems. *Theoretical Computer Science*, 351:394–406, 2006.

[34] Gary L. Miller and John Reif. Parallel tree contraction. Part 1: Fundamentals. In S. Micali, editor, *Advances in Computing Research 5: Randomness and Computation*, pages 47–72. JAI Press, Greenwich, CT, 1989.

[35] Gary L. Miller and John Reif. Parallel tree contraction. Part 2: Further applications. *SIAM Journal on Computing*, 20:1128 – 1147, 1991.

[36] Ljubomir Perković and Bruce Reed. An improved algorithm for finding tree decompositions of small width. *International Journal of Foundations of Computer Science*, 11:365–371, 2000.

[37] Bruce Reed. Finding approximate separators and computing tree-width quickly. In *Proceedings of the 24th Annual Symposium on Theory of Computing, STOC'92*, pages 221–228, New York, 1992. ACM Press.

[38] Rhilipped Rinaudo, Yann Ponty, Dominique Barth, and Alain Denise. Tree decomposition and parameterized algorithms for RNA structure-sequence alignment including tertiary interactions and pseudoknots (extended abstract). In Benjamin J. Raphael and Jijun Tang, editors, *Proceedings of the 12th International Workshop on Algorithms in Bioinformatics, WABI 2012*, volume 7534 of *Lecture Notes in Computer Science*, pages 149–164. Springer Verlag, 2012.

[39] Neil Robertson and Paul D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7:309–322, 1986.

[40] Neil Robertson and Paul D. Seymour. Graph minors. XIII. The disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63:65–110, 1995.

[41] John E. Savage. *Models of computation - exploring the power of computing*. Addison-Wesley, 1998.