An $O(c^k n)$ 5-Approximation Algorithm for Treewidth

Hans L. Bodlaender*, Pål Grønås Drange†, Markus S. Dregi†,
Fedor V. Fomin†, Daniel Lokshtanov† and Michał Pilipczuk†
* *Department of Information and Computing Sciences, Utrecht University, Utrecht, the Netherlands.*
Email: h.l.bodlaender@uu.nl
† *Department of Informatics, Univerity of Bergen, Bergen, Norway.*
Emails: {Pal.Drange, Markus.Dregi, fomin, Daniel.Lokshtanov, Michal.Pilipczuk}@ii.uib.no

***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.**

***Keywords*-treewidth, fixed-parameter tractability, approximation**

I. 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 $O(c^k n)$, 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 [10], [19] as well as new problems motivated by various applications (just a few of many examples are [1], [23], [25], [31]) 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 $O(c^k n)$ 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 $O(c^k n)$ either reports correctly that the treewidth of G is more that k, or construct a tree decomposition of width at most αk for some constant α.

The second, fourth and sixth author were supported by the ERC grant "Rigorous Theory of Preprocessing", reference 267959.

Table I
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.

Approximation	$f(k)$	$g(n)$	Reference
exact	$O(1)$	$O(n^{k+2})$	[4]
$4k+3$	$O(3^{3k})$	n^2	[33]
$8k+7$	$2^{O(k \log k)}$	$n \log^2 n$	[26]
$8k+O(1)$	$2^{O(k \log k)}$	$n \log n$	[30]
exact	$k^{O(k^3)}$	n	[9]
$4.5k$	$O(2^{3k} k^{3/2})$	n^2	[3]
$(3+2/3)k$	$O(2^{3.6982k} k^3)$	n^2	[3]
$O(k \log k)$	$O(k \log k)$	n^4	[3]
$O(k\sqrt{\log k})$	$O(1)$	$n^{O(1)}$	[21]
$3k+4$	$2^{O(k)}$	$n \log n$	This paper
$5k+4$	$2^{O(k)}$	n	This paper

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 $O(c^k n^2)$, $O(2^{O(k \log k)} n \log n)$, or $O(k^{O(k^3)} n)$, see Table I. 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 [29] is only a factor polynomial in k faster, however if the treewidth is larger than k, it gives a subgraph of treewidth more than k with a tree decomposition of width at most $2k$, 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. [20], but its running time is not linear.

In this paper, we give the first constant factor approximation algorithm for the treewidth such that its running time is single exponential in the 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 $O(c^k n)$ for some $c \in \mathbb{N}$, 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 $O(c^k n \log n)$.

Related results and techniques: The basic shape of our algorithm is along the same lines as most of the treewidth approximation algorithms [3], [13], [21], [26], [30], [33], i.e., a specific scheme of repeatedly finding separators. If we ask for polynomial time approximation algorithms, the currently best result is that of Feige et al. [21], which gives in polynomial (but not linear) time a tree decomposition of width $O(k\sqrt{\log k})$ 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. 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 performing 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], [16], [17], [18], [24].

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], [29], 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 II-C). 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 [27], [28].

Using the data structure to "implement" the algorithm of Robertson and Seymour [33] already gives an $O(c^k n \log n)$ 3-approximation for treewidth (Section II-C). 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 [30], 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" (compared to k), and one case when n is "large". We consider n to be small if $n \leq 2^{2^{c_0 k^3}}$, for some constant c_0. For small values of n, we apply the $O(c^k n \log^{(2)} n)$ algorithm from Section II-G. This will yield a running time linear in n since $\log^{(2)} n = O(k^3)$. For larger values of n, we show that the linear time algorithms of [9] or [29] 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 transitions 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 [22], see also [2]. Our algorithm assumes the RAM model of computation [34], and the only aspect of the RAM model which we exploit is the ability to look up an entry in a table in constant time, independently of the size of the table. This capability is essential in almost every linear time graph algorithm including breadth first search and depth first search.

Notation: We give some basic definitions and notation, used throughout the paper. For $\alpha \in \mathbb{N}$, by $\log^{(\alpha)} n$ we denote α-times folded function $\log n$. For the presentation of our results, it is more convenient to regard tree decompositions as rooted. Henceforth, 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, called *bags*, with the usual properties. 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.

II. Proof outline

This is an extended abstract. The full version can be found on arXiv.org [11]. In this section we give only an outline of the main ideas behind the results and refer to the full version where necessary. Our algorithm builds on the constant factor approximation algorithm for treewidth described in Graph Minors XIII [33] with running time $O(c^k n^2)$. We start with a brief explanation of a variant of this algorithm.

A. 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 use the variant of the lemma where vertices have weights 0 or 1.

Lemma II.1 (Graph Minors II [32]). *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.*

The set X with properties ensured by Lemma II.1 will be called a *balanced S-separator*. 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 II.1, which can be found in the full version, is constructive if one has access to a tree decomposition of G of width less than k. More precisely, for any such decomposition one of its bags satisfies the condition on S. Since we are trying to compute a decomposition of G of small width, the algorithm does not have such a decomposition at hand. We therefore have to settle for the following algorithmic variant of Robertson and Seymour [32].

Lemma II.2 ([33]). *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 $O(3^{|S|}k^{O(1)}(n+m))$.*

The proof of Lemma II.2 uses Lemma II.1 to certify existence of X such that $|X| \leq k+1$ and every component of $G \setminus X$ has at most $\frac{1}{2}|S|$ vertices of 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. Hence, it suffices to guess the intersection of S with the left side, the right side and with X ($3^{|S|}$ choices), and check existence of an appropriate separator X extending the guessed intersection with S using one run of max-flow.

The algorithm takes as input G, S, k, with S a set with at most $3k+4$ vertices, and either concludes that the treewidth of G is larger than k or finds a tree decomposition of width at most $4k+4$ such that the top bag of the decomposition contains S. On input G, S, k the algorithm starts by ensuring that $|S| = 3k+4$. If $|S| < 3k+4$ the algorithm just adds arbitrary vertices to S until equality is obtained. Then the algorithm applies Lemma II.2 and finds a set X of size at most $k+1$ such that each component C_i of $G \setminus X$ satisfies $|C_i \cap S| \leq \frac{2|S|}{3}$, so since $|S| = 3k+4$ then $|C_i \cap S| \leq 2k+2$. Note that this means that there are at least two components C_i. Also, for each C_i we have $|(S \cap C_i) \cup X| \leq (2k+2)+(k+1) < 3k+4$. 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 $4k+4$ 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 its size is at most $|S| + |X| \leq 4k+5$, and so the width of the decomposition is at most $4k+4$ as claimed.

The running time of the algorithm is governed by the recurrence $T(n,k) \leq O(3^{|S|}k^{O(1)}(n+m)) + \sum_{C_i} T(|C_i \cup X|, k)$, which solves to $T(n,k) \leq (3^{3k}k^{O(1)}n(n+m))$ since $|S| = 3k+4$ and there are at least two non-empty components of $G \setminus X$. Finally, since any graph of size n of treewidth k has at most nk edges [12], the algorithm can safely output that $\mathrm{tw}(G) > k$ if $|E| > nk$. Thus the algorithm runs in $O(3^{3k}k^{O(1)}n(n+m)) = O(3^{3k}k^{O(1)}n^2)$ time.

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

Reed [30] observed that the running time of the algorithm of Robertson and Seymour [33] can be sped up from $O(n^2)$ 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 [30] states neither the dependence on k of his algorithm nor the approximation ratio, but a careful analysis shows that this dependence and the approximation ratio are $k^{O(k)}$ and $8k+O(1)$, respectively. The main idea of this algorithm is that the recurrence above 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 ensured that each component C_i of $G \setminus X$ had at most $\lambda \cdot n$ vertices for some $\lambda < 1$, the recurrence above solves to $O(n \log n)$ for fixed k. To see this consider simply 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 a constant factor smaller than the size of the components considered in the previous level, the number of levels is $O(\log n)$ and we have $O(n \log n)$ work in total.

By using Lemma II.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 II.2 to *find* an X which splits G in a balanced way using $S = V(G)$, the algorithm of Lemma II.2 takes time $O(3^{|S|}k^{O(1)}(n+m)) = O(3^n n^{O(1)})$, which is exponential in n. Reed [30] gave an algorithmic variant of Lemma II.1 especially tailored for the case where $S = V(G)$.

Lemma II.3 ([30]). *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 II.3 as stated here is never explicitly proved in [30], but it follows easily from the arguments given there.

Using Lemmata II.2 and II.3, we can 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.

The algorithm is similar to the one in Section II-A, except that we let $|S| = 6k+6$ and that we find two separators, X_1 and X_2 instead of just one, using first Lemma II.2 and then Lemma II.3. The algorithm runs itself recursively on $(G[C_i \cup X], (S \cap C_i) \cup X, k)$, where $X = X_1 \cup X_2$ and C_i are the connected components of $G \setminus X$. The running time of the algorithm is governed by the recurrence $T(n,k) \leq O\left(k^{O(k)}(n+m)\right) + \sum_{C_i} T(|C_i \cup X|, k)$, which solves to $T(n,k) \leq O(k^{O(k)}(n+m)\log n)$ since each C_i has size at most $\frac{3}{4}n$. Again, since $m \leq nk$, the running time of the algorithm is upper bounded by $O(k^{O(k)}n \log n)$.

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

In this section we sketch a proof of the following theorem, whose full proof can be found in the full version.

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 $O(c^k n \log n)$ 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 [29]. 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 II.4. 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 II.5. *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 $m = O(kn)$, which is the case in graphs of treewidth at most k, the improved graph can be computed in $O(k^{O(1)}n)$ 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 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 ingredients that we use.

Lemma II.6. *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 $O(k^{O(1)}n)$ 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 II.6 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 $O(k)$; the goal is to either conclude that the treewidth of G is at least k or find a tree decomposition 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 II.6 as black boxes. We now state the properties of our algorithm for the compression step in the following lemma.

Lemma II.7. *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 $O(c^k n \log n)$ 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 an algorithm proving Theorem II assuming the correctness of Lemma II.7, which we argue in the subsequent sections. The 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 II.6 on $(G, 3k+4)$. If the algorithm of Lemma II.6 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 $\mathcal{T}_{\text{apx}}$ of G of width at most $6k+9$ [9]. Now we can run the algorithm of Lemma II.7 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 $\mathrm{tw}(G) > k$.

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 II.5. Otherwise we obtain a tree decomposition of $G_I \setminus X$ of width at most $3k+4$. We may now apply Lemma II.6 and obtain a tree decomposition $\mathcal{T}_{\text{apx}}$ 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 II.7 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 $\mathrm{tw}(G) > k$.

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 II.6 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 $\mathrm{tw}(G) > k$ 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 II.7, taking time $O(c^k n \log n)$. This gives the following recurrence: $T(n,k) \leq O\left(c^k n \log n\right) + T\left(n\left(1 - \frac{1}{O(k^6)}\right), k\right)$. The recurrence leads to a geometric series and solves to $T(n,k) \leq O(c^k k^{O(1)} n \log n)$, completing the proof. For a thorough analysis, see the full version.

D. A compression algorithm

We now proceed to give a sketch of a proof for a slightly weakened form of Lemma II.7. 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 $\mathrm{tw}(G) > k$ 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 II.7—we will discuss the proof of Lemma II.7 in Section II-E. The aim of this section is to demonstrate that the recursive scheme of Section II-C 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 this trick is used 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 II-B, but instead of using Lemma II.3 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, however, 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 we also need to store the cardinalities of the sides which gives us another dimension of size n. We now explain how to overcome this issue. We first mimic the proof of Lemma II.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 $|X_0| \leq \mathrm{w}(\mathcal{T}_{\text{apx}}) + 1$. For every way of writing $k_L + k_X + k_R = k+1$ and every partition of X_0 into $X_L \cup X_X \cup X_R$ with $|X_X| = k_X$, we do as follows:

First we find in time $O(c^k n)$ using dynamic programming over the tree decomposition $\mathcal{T}_{\text{apx}}$ 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. Then we find in time $O(c^k n)$ using dynamic programming over the tree decomposition $\mathcal{T}_{\text{apx}}$ 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 the full version) that if $\mathrm{tw}(G) \leq k$ then there exists a choice of k_L, k_X, k_R such that $k_L + k_X + k_R = k+1$ 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 II.7. It does however yield a version of Lemma II.7 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 II.7 to give an 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 II.7 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 [33] and the algorithm of Reed [30] find a separator X and proceed recursively on the components of $G \setminus X$. If we use $O(c^k n)$ time to find the separator X, then the total running time must be at least $O(c^k nd)$ 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 $\Omega(\log n)$. We now give a proof of Lemma II.7 that *almost* overcomes these issues.

E. A better compression algorithm

We give a sketch of the proof of Lemma II.7. 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 $\mathcal{T}_{\text{apx}}$ 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 II-A, 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 II.7, but runs in time $O(c^k n^2)$ rather than $O(c^k n \log n)$ we run the algorithm described in Section II-A setting $S = S_0$ in the beginning. Instead of using Lemma II.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 II.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 $\mathcal{T}_{\text{apx}}$ in time $O(c^k n)$ to find such an X. Going through the analysis of Section II-A 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 $\mathcal{T}_{\text{apx}}$ in time $O(c^k n)$ to find an X such that every component of $G \setminus X$ has at most $\frac{1}{2}|S|$ vertices which are in S. The problem is that $G \setminus X$ could potentially have many components and we cannot store information about each of these components individually. The following lemma, whose proof appears in the full version, shows how to deal with this problem.

Lemma II.8. *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 $i = 1, 2, 3$.*

Lemma II.8 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 $|M_i \cap S| \leq |S|/2$ for $i = 1, 2, 3$. This can easily be done in time $O(c^k n)$ by dynamic programming over the tree decomposition $\mathcal{T}_{\text{apx}}$. This yields the promised algorithm that satisfies all of the requirements of Lemma II.7, 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 II.7, 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 $\mathcal{T}_{\text{apx}}$. 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.

The following proposition lets us assume that the tree decomposition $\mathcal{T}_{\text{apx}}$ has depth $O(\log n)$:

Proposition II.9 (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.*

In Section II-F 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 $\mathcal{T}_{\text{apx}}$ 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 (is defined in the next section), 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 II.7 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 $\mathcal{T}_{\text{apx}}$. 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! The reason for this is 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 role 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": 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 queries are explained in the next section.

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 the full version. 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 II.7. A full proof of Lemma II.7 assuming the data structure as a black box may be found in the full version.

F. 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 $\mathcal{T}_{\text{apx}}$ 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 $O(c^k \log n)$. 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 is disjoint with F.

findNeighborhood
: Return $N(U) \cap S$ if $|N(U) \cap S| < 2k+3$ and a marker '$\boxtimes$' otherwise,

where U is the component of $G \setminus S$ containing π.

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 $\mathcal{T}_{\text{apx}}$ 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 II.8 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 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, k). 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 $O(c^k \log n)$? If the dynamic program is done carefully, one can ensure that adding/removing a vertex to/from S affects only the dynamic programming tables for a single node t in the decomposition, together with all of its $O(\log n)$ ancestors. Performing the changes thus takes time $O(c^k k^{O(1)} \log n)$, since for each update we recompute $O(\log n)$ dynamic programming tables. 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 storing "all possible ways the vertices in a bag could be connected below the bag". A complete exposition of the data structure can be found in the full version.

G. 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 $O(c^k n \log^{(\alpha)} n)$ 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 $O(c^k n \log\log n)$ 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 II-E. It uses the data structure of Section II-F 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 II-A would spend $O(c^k (\log\log n)^2)$ time in each recursive call, while our 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 $O(c^k n \log n)$ algorithm described in Section II-E, 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 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 II-A, 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 II-B 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 II-B has a worse approximation ratio than the algorithm of Robertson and Seymour described in Section II-A. 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 used 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 $O(c^k n \log^{(\alpha)} n)$ to $O(c^k n \log^{(\alpha+1)} n)$. 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 $O(c^k n \log^{(\alpha)} n)$ time algorithm on each of these components yields an algorithm with running time $O(c^k n \log^{(\alpha+1)} n)$.

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 II-D 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 II-D 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, together with a full proof can be found in the full version of this article. This completes the proof sketch of Theorem III.

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

The algorithms of Section II-G are in fact already $O(c^k n)$ algorithms unless n is astronomically large compared to k. If, for example, we have $n \leq 2^{2^{2^k}}$ then $\log^{(3)} n \leq k$ and so $O(c^k n \log^{(3)} n) \leq O(c^k kn) = O(c^k n)$ since we can assume that $k \leq n$. 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 II-C 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], [22] 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 III 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 the full version of this article.

References

[1] Ittai Abraham, Moshe Babaioff, Shaddin Dughmi, and Tim Roughgarden. Combinatorial auctions with restricted complements. In *EC'12*, pp. 3–16, 2012.

[2] Karl R. Abrahamson and Michael R. Fellows. Finite automata, bounded treewidth and well-quasiordering. In *Proc. of the AMS Summer Workshop on Graph Minors, Graph Structure Theory, Contemporary Mathematics vol. 147*, pp. 539–564, 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 *APPROX'12 and RANDOM'12*, pp. 13–24, 2012.

[6] Hans L. Bodlaender. Dynamic programming algorithms on graphs with bounded tree-width. In *ICALP'88*, pp. 105–119, 1988.

[7] Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *STOC'93*, pp. 226–234, 1993.

[8] Hans L. Bodlaender. Dynamic algorithms for graphs with treewidth 2. In *WG'93*, pp. 112–124, 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, Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. In *ICALP'13 (1)*, pp. 196–207, 2013.

[11] Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michał Pilipczuk. An $O(c^k n)$ 5-approximation algorithm for treewidth. *Report on arXiv* 1304.6321, 2013.

[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] 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.

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

[18] Robert F. Cohen, S. Sairam, Roberto Tamassia, and J. S. Vitter. Dynamic algorithms for optimization problems in bounded tree-width graphs. In *IPCO'93*, pp. 99–112, 1993.

[19] Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Fast Hamiltonicity checking via bases of perfect matchings. In *STOC'13*, pp. 301–310, 2013.

[20] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Logspace versions of the theorems of Bodlaender and Courcelle. In *FOCS'10*, pp. 143–152, 2010.

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

[22] Michael R. Fellows and Michael A. Langston. An analogue of the Myhill-Nerode theorem and its use in computing finite-basis characterizations. In *FOCS'89*, pp. 520–525, 1989.

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

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

[25] 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.

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

[27] Gary L. Miller and John Reif. Parallel tree contraction. Part 1: Fundamentals. In *Advances in Computing Research 5: Randomness and Computation*, pp. 47–72, 1989.

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

[29] 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.

[30] Bruce Reed. Finding approximate separators and computing tree-width quickly. In *STOC'92*, pp. 221–228, 1992.

[31] 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 *WABI'12*, pp. 149–164, 2012.

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

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

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