Computing Treedepth in Polynomial Space and Linear FPT Time

Wojciech Nadara

Institute of Informatics, University of Warsaw, Poland

Michał Pilipczuk

Institute of Informatics, University of Warsaw, Poland

Marcin Smulewicz

Institute of Informatics, University of Warsaw, Poland

Abstract

The *treedepth* of a graph G is the least possible depth of an *elimination forest* of G: a rooted forest on the same vertex set where every pair of vertices adjacent in G is bound by the ancestor/descendant relation. We propose an algorithm that given a graph G and an integer d, either finds an elimination forest of G of depth at most d or concludes that no such forest exists; thus the algorithm decides whether the treedepth of G is at most d. The running time is $2^{\mathcal{O}(d^2)} \cdot n^{\mathcal{O}(1)}$ and the space usage is polynomial in n. Further, by allowing randomization, the time and space complexities can be improved to $2^{\mathcal{O}(d^2)} \cdot n$ and $d^{\mathcal{O}(1)} \cdot n$, respectively. This improves upon the algorithm of Reidl et al. [ICALP 2014], which also has time complexity $2^{\mathcal{O}(d^2)} \cdot n$, but uses exponential space.

2012 ACM Subject Classification Theory of computation → Fixed parameter tractability; Theory of computation → Graph algorithms analysis

Keywords and phrases treedepth, FPT, polynomial space

Digital Object Identifier 10.4230/LIPIcs.ESA.2022.79

Related Version *Full Version*: `https://arxiv.org/abs/2205.02656` [13]

Funding *Wojciech Nadara*: This paper is a part of project CUTACOMBS that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 714704).
Michał Pilipczuk: This paper is a part of project BOBR that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 948057).

Acknowledgements We would like to thank Marcin Mucha and Marcin Pilipczuk for discussions on the topic of this work.

1 Introduction

An *elimination forest* of a graph G is a rooted forest F whose vertex set is the same as that of G, where for every edge uv of G, either u is an ancestor of v in F or vice versa. The *treedepth* of G is the least possible depth of an elimination forest of G. Compared to the better-known parameter *treewidth*, treedepth measures the depth of a tree-like decomposition of a graph, instead of width. The two parameters are related: if by $\mathrm{td}(G)$ and $\mathrm{tw}(G)$ we denote the treedepth and the treewidth of an n-vertex graph G, then it always holds that $\mathrm{tw}(G) \leqslant \mathrm{td}(G) \leqslant \mathrm{tw}(G) \cdot \log_2 n$ (Section 6.4 of [16]). However, the two notions are qualitatively different: for instance, a path on t vertices has treewidth 1 and treedepth $\Theta(\log t)$.

Treedepth appears prominently in structural graph theory, especially in the theory of sparse graphs of Nešetřil and Ossona de Mendez. There, it serves as a basic building block for fundamental decompositions of sparse graphs – *low treedepth colorings* – which can be used for multiple algorithmic purposes, including designing algorithms for Subgraph Isomorphism and model-checking First-Order logic. See [16, Chapters 6 and 7] for an introduction and [8, 9, 15, 17, 18, 19, 21, 20] for examples of applications.

In this work we are interested in using treedepth as a parameter for the design of fixed-parameter (FPT) algorithms. Clearly, every dynamic programming algorithm working on a tree decomposition of a graph can be adjusted to work also on an elimination forest, just because an elimination forest of depth d can be easily transformed into a tree decomposition of width $d-1$. However, it has been observed in [7, 22, 10, 14, 20] that for multiple basic problems, one can design FPT algorithms working on elimination forests of bounded depth that have polynomial space complexity without sacrificing on the time complexity. These include the following: (In all results below, n is the vertex count and d is the depth of the given elimination forest.)

- A $3^d \cdot n^{\mathcal{O}(1)}$-time $\mathcal{O}(d + \log n)$-space algorithm for 3-Coloring [22].
- A $2^d \cdot n^{\mathcal{O}(1)}$-time $n^{\mathcal{O}(1)}$-space algorithm for counting perfect matchings [7].
- A $3^d \cdot n^{\mathcal{O}(1)}$-time $n^{\mathcal{O}(1)}$-space algorithm for Dominating Set [7, 22].
- A $d^{|V(H)|} \cdot n^{\mathcal{O}(1)}$-time $n^{\mathcal{O}(1)}$-space algorithm for Subgraph Isomorphism [20]. (Here, H is the sought pattern graph.)
- A $3^d \cdot n^{\mathcal{O}(1)}$-time $n^{\mathcal{O}(1)}$-space algorithm for Connected Vertex Cover [10].
- A $5^d \cdot n^{\mathcal{O}(1)}$-time $n^{\mathcal{O}(1)}$-space algorithm for Hamiltonian Cycle [14].

We note that the approach used in [10, 14] to obtain the last two results applies also to several other problems with connectivity constraints. However, as these algorithms are based on the Cut&Count technique [4], they are randomized and no derandomization preserving the polynomial space complexity is known. An in-depth complexity-theoretical analysis of the time-space tradeoffs for algorithms working on different graph decompositions can be found in [22].

In the algorithms mentioned above one assumes that the input graph is supplied with an elimination depth of depth at most d. Therefore, it is imperative to design algorithms that given the graph alone, computes, possibly approximately, such an elimination forest. Compared to the setting of treewidth and tree decompositions, where multiple approaches have been proposed over the years (see e.g. [2, 11] for an overview), so far there is only a handful of algorithms to compute the treedepth exactly or approximately.

- It is well-known (see e.g. [16, Section 6.2]) that just running depth-first search and outputing the forest of recursive calls gives an elimination forest of depth at most $2^{\mathrm{td}(G)}$. So this gives a very simple linear-time approximation algorithm, but with the approximation factor exponential in the optimum.
- Czerwiński et al. [5] gave a polynomial-time algorithm that outputs an elimination forest of depth at most $\mathcal{O}(\mathrm{td}(G)\mathrm{tw}(G)\log^{3/2}\mathrm{tw}(G))$, which is thus an $\mathcal{O}(\mathrm{tw}(G)\log^{3/2}\mathrm{tw}(G))$-approximation algorithm.
- Reidl et al. [23] gave an exact FPT algorithm that in time $2^{\mathcal{O}(d\cdot\mathrm{tw}(G))}\cdot n$ either constructs an elimination forest of depth at most d, or concludes that the treedepth is larger than d.
- There is a naive algorithm computing treedepth directly from its definition that works in $\mathcal{O}(n^{\mathrm{td}(G)})$ time and uses polynomial space.
- Using a tradeoff that runs either of the last two approaches, depending on whether d is greater than $\sqrt{\epsilon \cdot \log n}$ or not, for any fixed $\epsilon > 0$, one can obtain an algorithm running in time $2^{\mathcal{O}(\frac{d^3}{\epsilon})} + \mathcal{O}(n^{1+\epsilon})$ and using polynomial space.

Recall here that $\mathrm{tw}(G) \leqslant \mathrm{td}(G)$, hence when parameterized by treedepth only, the mentioned results can be seen as an $\mathcal{O}(\mathrm{td}(G)\log^{3/2}\mathrm{td}(G))$-approximation in polynomial time, as well as an exact FPT algorithm with running time $2^{\mathcal{O}(d^2)}\cdot n$. In particular, obtaining a constant-factor approximation for treedepth running in time $2^{\mathcal{O}(\mathrm{td}(G))} \cdot n^{\mathcal{O}(1)}$ is a well-known open problem, see e.g. [5]. We note that implementation of practical FPT algorithms for computing treedepth was the topic of the 2020 Parameterized Algorithms and Computational Experiments (PACE) Challenge [12].

Our contribution. The exact algorithm of Reidl et al. [23] uses not only exponential time (in the treedepth), but also exponential space. This would make it a space bottleneck when applied in combination with any of the polynomial-space algorithms developed in [7, 22, 10, 14, 20]. The mentioned tradeoff trick brings back the polynomial space, but significantly deteriorates the running time both in terms of the factor exponential in d and the one polynomial in n. In this work we bridge these issues by proving the following result.

▶ **Theorem 1.** *There is an algorithm that given an n-vertex graph G and an integer d, either constructs an elimination forest of G of depth at most d, or concludes that the treedepth of G is larger than d. The algorithm runs in $2^{\mathcal{O}(d^2)} \cdot n^{\mathcal{O}(1)}$ time and uses $n^{\mathcal{O}(1)}$ space.*

The space and time complexities can be improved to $d^{\mathcal{O}(1)} \cdot n$ and expected $2^{\mathcal{O}(d^2)} \cdot n$, respectively, at the cost of allowing randomization: the algorithm may return a false negative with probability at most $\frac{1}{c \cdot n^c}$, where c is any constant fixed a priori; there are no false positives.

Thus, the randomized variant of the algorithm of Theorem 1 has the same time complexity as the algorithm of Reidl et al. [23], but uses polynomial space. However, the algorithm of Reidl et al. [23] is deterministic, contrary to ours. Note that apart from possible false negatives, the bound on the running time is only in expectation and not worst-case (in other words, our algorithm is both Monte Carlo and Las Vegas). However, one can turn this into a worst-case bound at the cost of increasing the probability of false negatives to 1/2 by forcefully terminating the execution if the algorithm runs for twice as long as expected.

Simultaneously achieving time complexity linear in n and polynomial space complexity is a property that is desired from an algorithm for computing the treedepth of a graph. While many of the polynomial-space FPT algorithms working on elimination forests do not have time complexity linear in n due to the usage of various algebraic techniques, the simplest ones that exploit only recursion – like the ones for 3-Coloring or Independent Set considered in [22] – can be easily implemented to run in time $2^{\mathcal{O}(d)} \cdot n$ and space $d^{\mathcal{O}(1)} \cdot n$. Thus, the randomized variant of the algorithm of Theorem 1 would neither be a bottleneck from the point of view of space complexity nor from the point of view of the dependency of the running time on n. Admittedly, the parametric factor in the runtime of our algorithm is $2^{\mathcal{O}(d^2)}$, as compared to $2^{\mathcal{O}(d)}$ in most of the aforementioned polynomial-space FPT algorithms working on elimination forests; this brings us back to the open problem about constant-factor approximation for treedepth running in time $2^{\mathcal{O}(\mathrm{td}(G))} \cdot n^{\mathcal{O}(1)}$ raised in [5].

Let us briefly discuss the techniques behind the proof of Theorem 1. The algorithm of Reidl et al. [23] starts by approximating the treewidth of the graph (which is upper bounded by the treedepth) and tries to constructs an elimination forest of depth at most d by bottom-up dynamic programming on the obtained tree decomposition. By applying the iterative compression technique, we may instead assume that we are supplied with an elimination forest of depth at most $d+1$, and the task is to construct one of depth at most d.

Applying now the approach of Reidl et al. [23] directly (that is, after a suitable adjustment from the setting of tree decompositions to the setting of elimination forests) would not give an algorithm with polynomial space complexity. The reason is that their dynamic programming procedure is quite involved and in particular keeps track of certain disjointness conditions; this is a feature that is notoriously difficult to achieve using only polynomial space. Therefore, we resort to the technique of *inclusion-exclusion branching*, used in previous polynomial-space algorithms working on elimination forests; see [7, 22] for basic applications of this approach. In a nutshell, the idea is to count more general objects where the disjointness contraints are relaxed, and to use inclusion-exclusion at each step of the computation to make sure that objects not satisfying the constraints eventually cancel out. We note that while the application of inclusion-exclusion branching was rather simple in [7, 22], in our case it poses a considerable technical challenge. In particular, along the way we do not count single values, but rather polynomials with one formal variable that keeps track of how much the disjointness constraints are violated. In the exposition layer, our application of inclusion-exclusion branching mostly follows the algorithm for Dominating Set of Pilipczuk and Wrochna [22].

In this way, we can count the number of elimination forests[1] of depth at most d in time $2^{\mathcal{O}(d^2)} \cdot n^{\mathcal{O}(1)}$ and using polynomial space. So in particular, we can decide whether there exists at least one such elimination forest. Such a decision algorithm can be quite easily turned into a construction algorithm using self-reducibility of the problem. This establishes the first part of Theorem 1.

As for the second part – the randomized linear-time FPT algorithm using polynomial space – there are several obstacles that need to be overcome. First, there is a multiplicative factor n in the running time coming from the iterative compression scheme. We mitigate this issue by replacing iterative compression with the recursive contraction scheme used by Bodlaender in his linear-time FPT algorithm to compute the treewidth of a graph [1]. Second, when using self-reducibility, we may apply the decision procedure n times, each taking at least linear time. This is replaced by an approach based on color coding, whose correctness relies on the fact that in a connected graph of treedepth at most d there are at most $d^{\mathcal{O}(d)}$ different feasible candidates for the root of an optimum-depth elimination tree [3]. Finally, in the counting procedure we may operate on numbers of bitsize as large as polynomial in n. This is resolved by hashing them modulo a random prime of magnitude $\Theta(\log n)$, so that we may assume that arithmetic operations take unit time.

We remark that it is relatively rare that a polynomial-space algorithm based on algebraic techniques can be also implemented so that it runs in time linear in the input size. Therefore, we find it interesting and somewhat surprising that this can be achieved for the problem of computing the treedepth of a graph, which combinatorially is rather involved.

Organization. After brief preliminaries in Section 2, in Section 3 we prove the first part of Theorem 1: we give a deterministic algorithm that runs in time $2^{\mathcal{O}(d^2)} \cdot n^{\mathcal{O}(1)}$ time and uses polynomial space. Then, in Section 4 we improve the time and space complexities to $2^{\mathcal{O}(d^2)}n$ and $d^{\mathcal{O}(1)}n$ respectively, at the cost of introducing randomization. The proofs of the results marked ★ are deferred to the full version of the work due to the space constraints.

2 Preliminaries

Standard notation. All graphs in this paper are finite, undirected, and simple (i.e. with no loops on vertices or multiple edges with the same endpoints). For a graph G and a vertex subset $A \subseteq V(G)$, by $N_G[A]$ we denote the *closed neighborhood* of A: the set consisting of all vertices that are in A or have a neighbor in A.

[1] Formally, we count only elimination forests satisfying some basic connectivity property, which we call *sensibility*.

For a function $f\colon A \to B$ and a subset of the domain $X \subseteq A$, by $f(X)$ we denote the image of f on X. The image of f is denoted $\mathrm{im}(f) = f(A)$. For an element e outside of the domain and a value α, by $f[e \to \alpha]$ we denote the extension of f obtained by additionally mapping e to α.

We denote the set $\{1, 2, \ldots, k\}$ as $[k]$. We assume the standard word RAM model of computation with words of length $\log n$, where n is the vertex count of the input graph.

(Elimination) forests and treedepth. Consider a rooted forest F. By Anc_F we denote the ancestor/descendant relation in F: for $u, v \in V(F)$, $\mathsf{Anc}_F(u, v)$ holds if and only if u is an ancestor of v or v is an ancestor of u in F. We assume that a vertex is an ancestor of itself, so in particular $\mathsf{Anc}_F(u, u)$ is always true. We also use the following notation. For $u \in V(F)$, by $\mathsf{tail}_F[u]$ we denote the set of all ancestors of u (including u) and by $\mathsf{tree}_F[u]$ we denote the set of all descendants of u, including u. Further, let $\mathsf{tail}_F(u) = \mathsf{tail}_F[u] - \{u\}$, $\mathsf{tree}_F(u) = \mathsf{tree}_F[u] - \{u\}$, and $\mathsf{comp}_F[u] = \mathsf{tail}_F[u] \cup \mathsf{tree}_F[u]$. Note that $v \in \mathsf{comp}_F[u]$ if and only if $\mathsf{Anc}_F(u, v)$ holds. By $\mathsf{chld}_F(u)$ we denote the set of children of u in F, and by $\mathsf{depth}_F(u)$ we denote the depth of u in F, that is, $\mathsf{depth}_F(u) = |\mathsf{tail}_F[u]|$ (in particular, roots have depth one). The *depth* of a rooted forest F is the maximum depth_F among its vertices. For a set of vertices $A \subseteq V(F)$, by $\mathsf{cl}_F(A) = \bigcup_{u \in A} \mathsf{tail}_F[u]$ we denote the ancestor closure of A. A *prefix* of a rooted forest F is a rooted forest induced by some ancestor-closed set $A \subseteq V(F)$; that is, it is the forest on A with the parent-child relation inherited from F.

In this paper we are mostly interested in the notion of an elimination forest and of the treedepth of a graph.

▶ **Definition 2.** *An* elimination forest *of a graph G is a rooted forest F on the same set of vertices as G such that for every edge $uv \in E(G)$, we have that $\mathsf{Anc}_F(u, v)$ holds. The* treedepth *of a graph G is the least possible depth of an elimination forest of G.*

Note that an elimination forest of a connected graph must be connected as well, so in this case we may speak about an *elimination tree* (however, elimination forest of a disconnected graph could be a tree as well). Sometimes, instead of identifying $V(G)$ and $V(F)$, we treat them as disjoint sets and additionally provide a bijective mapping $\phi\colon V(G) \to V(F)$ such that $uv \in E(G)$ entails $\mathsf{Anc}_F(\phi(u), \phi(v))$. In such case we consider the pair (F, ϕ) to be an elimination forest of G. This will be always clear from the context. More generally, for $B \subseteq V(G)$ and a rooted forest F, we shall say that a mapping $\phi\colon B \to V(F)$ *respects edges* if $uv \in E(G)$ entails $\mathsf{Anc}_F(u, v)$ for all $u, v \in B$. In this notation, (F, ϕ) is an elimination forest of G if and only if ϕ is a bijection from $V(G)$ to $V(F)$ that respects edges on $V(G)$.

3 Deterministic FPT algorithm

In this section we prove the first part of Theorem 1: we give a deterministic polynomial-space algorithm with running time $2^{\mathcal{O}(d^2)} \cdot n^{\mathcal{O}(1)}$ that for a given n-vertex graph G, either outputs an elimination forest of G of depth at most d or concludes that no such forest exists. The most complex part of the algorithm will be procedure `CountElimTrees`, which, roughly speaking, counts the number of different elimination trees of a connected graph G of depth at most d. We describe `CountElimTrees` first, and then we utilize it to achieve the main result.

3.1 Description of `CountElimTrees`

As mentioned above, procedure `CountElimTrees` counts the number of different elimination trees of G of depth at most d. However, we will not count all of them, but only such that are in some sense minimal; a precise formulation will follow later. We remark that this part is inspired by the $3^d \cdot n^{\mathcal{O}(1)}$-time polynomial space algorithm of Pilipczuk and Wrochna [22] for counting dominating sets in a graph of bounded treedepth. This algorithm exploits the same underlying trick – sometimes dubbed "inclusion-exclusion branching" – but the application here is technically more involved than in [22].

Before describing `CountElimTrees`, let us carefully define objects that we are going to count. We start by recalling the following standard fact about the existence of elimination forests with basic connectivity properties.

▶ **Lemma 3 (★).** *Let H be a graph and let R be an elimination forest of H. Then there exists an elimination forest R' of H such that*

- *for every vertex u of H, we have $\mathsf{depth}_{R'}(u) \leqslant \mathsf{depth}_R(u)$; and*
- *whenever vertices $u, v \in V(H)$ belong to the same connected component of R', they also belong to the same connected component of H.*

We remark that computing R' can be easily done in linear time by using depth-first search from the root of each elimination tree in R. This procedure will be used many times throughout the algorithm when justifying the usual assumption that our current graph is connected. (Disconnected graphs will often naturally appear when recursing after performing some deletions in the original graph.)

The following lemma can be proved using a very similar, though a bit more involved reasoning. We will work with a fixed connected graph G and its elimination tree T.

▶ **Lemma 4 (★).** *Let G be a connected graph of treedepth at most d and T be an elimination tree of G (possibly of depth larger than d). Then there exists an elimination tree R of G of depth at most d that satisfies the following property: for every $u \in V(G)$ and $v_1, v_2 \in \mathsf{chld}_T(u)$, $v_1 \neq v_2$, we have*

$$\mathsf{cl}_R(\mathsf{comp}_T[v_1]) \cap \mathsf{cl}_R(\mathsf{comp}_T[v_2]) = \mathsf{cl}_R(\mathsf{tail}_T[u]). \tag{1}$$

An elimination tree R of a graph G satisfying the conclusion of Lemma 4 (that is, the depth of R is at most d and for all $u \in V(G)$ and distinct $v_1, v_2 \in \mathsf{chld}_T(u)$ we have (1)) will be called *sensible* with respect to T. In our search for elimination trees of low depth, we will restrict attention only to trees that are sensible with respect to some fixed elimination tree T. Then Lemma 4 justifies that we may do this without losing all solutions.

With all ingredients introduced, we may finally precisely state the goal of this section.

▶ **Lemma 5.** *There exists an algorithm* `CountElimTrees`(G, T, d) *that, given a connected graph G on n vertices, an elimination tree T of depth k, and an integer d, runs in time $2^{\mathcal{O}(dk)} \cdot n^{\mathcal{O}(1)}$, uses $n^{\mathcal{O}(1)}$ space, and outputs the number of different elimination trees of G of depth at most d that are sensible with respect to T.*

Note here that the input to `CountElimTrees` consists not only of G and d, but also of an auxiliary elimination tree T of G. The depth k of T may be, and typically will be, larger than d. Also, we assume that an elimination tree is represented solely by its vertex set and the ancestor relation. In particular, permuting children of a vertex yields the *same* elimination tree, which should be counted as the same object by procedure `CountElimTrees`.

The remainder of this section is devoted to the proof of Lemma 5. We first need to introduce some definition.

Let us arbitrarily enumerate the vertices of G as $v_1, v_2, \ldots, v_n$ in a top-down manner in T. That is, whenever v_i is an ancestor of v_j, we have $i \leqslant j$. Consider another rooted tree R and a mapping $\phi\colon V(T) \to V(R)$. For a vertex u of T, we call a vertex $v_i \in \mathsf{tree}_T(u)$ a *proper surplus image* (for u and (R, ϕ)) if at least one of the following conditions holds:

(i) $\phi(v_i) \in \mathsf{cl}_R(\phi(\mathsf{tail}_T[u]))$, or
(ii) there exists j such that $j < i$, $v_j \in \mathsf{tree}_T(u)$, and $\phi(v_j) = \phi(v_i)$.

We define *non-proper surplus images* analogously, but using sets $\mathsf{tail}_T(u)$ and $\mathsf{tree}_T[u]$ instead of $\mathsf{tail}_T[u]$ and $\mathsf{tree}_T(u)$, respectively.

We will work in the ring of polynomials $\mathbb{Z}[x]$, where x is a formal variable. By an abuse of notation, we equip this ring with an operation of division by x defined through equations:

$$\frac{x^i}{x} = \begin{cases} x^{i-1} & \text{if } i \geqslant 1, \\ 0 & \text{if } i = 0 \end{cases}$$

$$\frac{\alpha A + \beta B}{x} = \alpha \cdot \frac{A}{x} + \beta \cdot \frac{B}{x} \qquad \text{for all} \qquad A, B \in \mathbb{Z}[x] \text{ and } \alpha, \beta \in \mathbb{Z}.$$

Formally speaking, division by x is just the unique function from $\mathbb{Z}[x]$ to $\mathbb{Z}[x]$ satisfying the two properties above.

Even though our final goal is to count the number of elimination trees, along the way we are going to count more general objects, called *generalized elimination trees*. A generalized elimination tree of a graph H is a rooted tree R along with a mapping $\phi\colon V(H) \to V(R)$ such that ϕ respects edges. Note that in particular, it may be the case that $\mathrm{im}(\phi) \subsetneq V(R)$ or that $\phi(u) = \phi(v)$ for some $u, v \in V(H)$. Clearly, a generalized elimination tree is an elimination tree in the usual sense if and only if ϕ is a bijection between $V(H)$ and $V(R)$. We shall call two generalized elimination trees (R, ϕ) and (R', ϕ') *isomorphic* if there is an isomorphism of rooted trees ψ mapping R to R' such that $\phi' = \psi \circ \phi$.

A generalized elimination tree (R, ϕ) of an induced subgraph H of G is *sensible* for T if for every $u \in V(H)$ and distinct $v_1, v_2 \in \mathsf{chld}_T(u) \cap V(H)$, we have $\mathsf{cl}_R(\phi(\mathsf{comp}_T[v_1])) \cap \mathsf{cl}_R(\phi(\mathsf{comp}_T[v_2])) = \mathsf{cl}_R(\phi(\mathsf{tail}_T[u]))$. Thus, this notion projects to sensibility of (standard) elimination trees when $H = G$ and (R, ϕ) is an elimination tree of G. Generalized elimination trees of induced subgraphs of G that are sensible for T shall be called *monsters*.

For a rooted tree K, a mapping ϕ with co-domain $V(K)$ is called a *cover* of K if $\mathsf{cl}_K(\mathrm{im}(\phi)) = V(K)$, or equivalently, every leaf of K is in the image of ϕ. For a vertex $u \in V(G)$, rooted tree K of depth at most d, a subset of vertices $A \subseteq V(K)$ that contains all leaves of K, and a mapping $\phi\colon \mathsf{tail}_T(u) \to A$ that is a cover of K, we define

$$f(u, K, \phi, A) = \sum_{i=0}^{n} a_i x^i \in \mathbb{Z}[x],$$

where a_i is the number of non-isomorphic monsters $(R, \overline{\phi})$ such that:
(i) $(R, \overline{\phi})$ is a generalized elimination tree of $G[\mathsf{comp}_T[u]]$ of depth at most d;
(ii) K is a prefix of R;
(iii) $\overline{\phi}$ is an extension of ϕ satisfying

$$V(R) - V(K) \subseteq \mathrm{im}(\overline{\phi}) \subseteq (V(R) - V(K)) \cup A; \qquad \text{and}$$

(iv) in $\mathsf{tree}_T[u]$ there are exactly i non-proper surplus images for u and $(R, \overline{\phi})$.

Note that since ϕ is assumed to be a cover of K, and by the second and third condition, the last condition can be rephrased as follows:

$$i = |\mathsf{tree}_T[u]| - |V(R) - V(K)|.$$

We define polynomial $g(u, K, \phi, L)$ analogously, but using $\mathsf{tail}_T[u]$, $\mathsf{tree}_T(u)$, and proper surplus images, instead of $\mathsf{tail}_T(u)$, $\mathsf{tree}_T[u]$ and non-proper surplus images. That in $\mathsf{tree}_T(u)$ there are i proper surplus images is then equivalent to $i = |\mathsf{tree}_T(u)| - |V(R) - V(K)|$.

Informally, f and g count partial solutions on subgraphs induced on subtrees of T, where in g we exclude the root u of the subtree. Values of g are computed by combining results of f from separate subtrees rooted at the children u into the result for the forest representing their union. Values of f are computed from the values of g by including a new vertex u that connects that forest into one tree.

Our goal now is to compute the polynomials $f(\cdot,\cdot,\cdot,\cdot)$ and $g(\cdot,\cdot,\cdot,\cdot)$ recursively over the elimination tree T. It can be easily seen that if $\mathsf{chld}_T(u) = \emptyset$ then

$$g(u, K, \phi, A) = \begin{cases} 1 & \text{if } \phi \text{ respects edges,} \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

Indeed, $(R, \overline{\phi}) = (K, \phi)$ is the only possible pair that can satisfy the last three conditions, and it is a sensible generalized elimination tree of $G[\mathsf{comp}_T[u]]$ if and only if ϕ respects edges.

First, we show how to compute polynomials $g(u,\cdot,\cdot,\cdot)$ based on the knowledge of polynomials $f(v,\cdot,\cdot,\cdot)$ for children v of u.

▶ **Lemma 6 (★).** *If* $\mathsf{chld}_T(u) \neq \emptyset$*, then for all relevant* u, K, ϕ, A *we have*

$$g(u, K, \phi, A) = \prod_{v \in \mathsf{chld}_T(u)} f(v, K, \phi, A)$$

The detailed proof of this Lemma can be found in the full version, however a short justification is that monsters counted in the definition of $g(u, K, \Phi, A)$ can be expressed in a product structure of monsters counted in the definitions of $f(v_i, K, \Phi, A)$, where $\mathsf{chld}_T(u) = \{v_1, \ldots, v_c\}$.

Let us elaborate on the intuition on what happened in Lemma 4. Intuitively, we aggregated information about the children of u to the information about u itself. Since in the definitions of monsters we do not insist on the mappings being injective, this aggregation could have been performed by a simple product of polynomials (though, the assumption of sensibility was crucial for arguing the correctness). In a natural dynamic programming, such as the one in [23], one would need to ensure injectivity when aggregating information from the children of u, which would result in a dynamic programming procedure that would need to keep track of all subsets of K (and thus use exponential space). Thus, relaxing injectivity here allows us to use simple multiplication of polynomials, but obviously we will eventually need to enforce injectivity. The idea is that we enforce surjectivity instead, and make sure that the size of the co-domain matches the size of the domain. In turn, surjectivity is enforced using inclusion-exclusion in the computation of polynomials $f(u,\cdot,\cdot,\cdot)$ based on polynomials $g(u,\cdot,\cdot,\cdot)$, which is the subject of the next lemma. Ensuring that the size of the co-domain matches the size of the domain is done through maintaining the number of surplus images. The intuition behind their somewhat intricate definition is the following. We want to maintain the difference between (i) the number of vertices we have already processed and forgotten about, and (ii) the number of forgotten vertices in a monster that we introduced in order to accommodate the vertices (i). Vertices contributing to this difference are exactly the vertices that have been mapped to forgotten vertices of a monster that have already been "taken" or that have not been forgotten yet; this corresponds to the definition of surplus images. We know that if multiple vertices were mapped to the same vertex of a partial monster, then this partial monster will not become a valid treedepth decomposition. We do not have a way of discovering this immediately (as we cannot keep track of any disjointness conditions), but extensions of such partial monsters will not be counted in the final result. The reason for that is that either the sizes of the co-domains will not match the sizes of the domains, or they will cancel out in the exclusion-inclusion computation due to not being surjective.

▶ **Lemma 7** (★). *For all relevant u, K, ϕ, A, we have:*

$$f(u,K,\phi,A) = \sum_{v\in A} x\cdot g(u,K,\phi[u\to v],A) + \sum_{w\in K}\sum_{p=1}^{d-\mathsf{depth}(w)} \frac{1}{x^{p-1}} \sum_{B\subseteq\{w_1,\ldots,w_{p-1}\}} (-1)^{p-1-|B|} g(u, K[w,w_1,\ldots,w_p], \phi[u\to w_p], A\cup B\cup\{w_p\}),$$

where $K[w,w_1,\ldots,w_p]$ denotes the rooted tree obtained from K by adding a path $[w,w_1,\ldots,w_p]$ so that w is the parent of w_1 and each w_i is the parent of w_{i+1}, for $i\in\{1,\ldots,p-1\}$.

We need to take an additional care of how to deduce the overall number of elimination trees based on the polynomial $f(\cdot,\cdot,\cdot,\cdot)$ and $g(\cdot,\cdot,\cdot,\cdot)$. Define polynomial

$$h = \sum_{p=1}^{d} \frac{1}{x^{p-1}} \sum_{B\subseteq\{w_1,\ldots,w_{p-1}\}} (-1)^{p-1-|B|} g(r,[w_1,\ldots,w_p],[r\to w_p], B\cup\{w_p\}) \in \mathbb{Z}[x],$$

where r is the root of T, $[w_1,\ldots,w_p]$ is a path on p vertices rooted at w_1, and $[r\to w_p]$ denotes the function with domain $\{r\}$ that maps r to w_p.

▶ **Lemma 8.** *The number of elimination trees of G that are sensible with respect to T and have depth at most d is the term in h standing by x^0.*

Proof. By Lemma 7, the formula can be seen as the formula for $f(r,K,\phi,A)$ for empty K, ϕ, and A. Therefore, h can be written as $h=\sum_{i=0}^{n} a_i x^i$, where a_i is the number of non-isomorphic sensible generalized elimination trees $(R,\overline{\phi})$ such that R has depth at most d, $\overline{\phi}\colon V(G)\to V(R)$ is surjective, and in G there are i non-proper surplus images for r and $(R,\overline{\phi})$. However, since K is empty, the number of surplus images is exactly the number of vertices $v_j\in V(G)$ that are mapped by $\overline{\phi}$ to the same vertex of R as some other vertex of G with a smaller index. Then the assertion that $\overline{\phi}$ is injective is equivalent to the assertion that the number of such surplus images is 0. It follows that the number of non-isomorphic sensible elimination trees of G of depth at most d is equal to the term in h that stands by x^0. ◀

Having established Lemmas 6, 7 and 8, we can conclude the description of procedure `CountElimTrees`. By 8, the goal is to compute polynomial h and return the coefficient standing by x^0. We initiate the computation using the formula for h, and then we use two mutually-recursive procedures to compute polynomials $f(\cdot,\cdot,\cdot,\cdot)$ and $g(\cdot,\cdot,\cdot,\cdot)$ using formulas provided by Lemmas 6 and 7. The base case of recursion is for a leaf of T, where we use formula (2).

The correctness of the procedure is established by Lemmas 6, 7 and 8. So it remains to bound its time complexity and memory usage. It is clear that polynomials that we compute will always have degrees at most n. Trees K relevant in the computation will never have more than dk vertices, for at every recursive call the tree K can grow by at most d new vertices.

As the next step, we bound the numbers that can be present in the computations.

▶ **Lemma 9** (★). *Every coefficient of $f(u,K,\phi,A)$ is an integer from the range $[0,(dk\cdot 2^d)^{|\mathsf{tree}_T[u]|}]$ and every coefficient of $g(u,K,\phi,A)$ is an integer from the range $[0,(dk\cdot 2^d)^{|\mathsf{tree}_T(u)|}]$. Hence, all integers present in the computations are at most $(dk2^d)^n$.*

It follows that all integers present in the computation have bitsize bounded polynomially in n.

As for the memory usage, the run of the algorithm is a recursion of depth bounded by $2k$. The memory used is a stack of at most $2k$ frames for recursive calls of procedures computing polynomials $f(\cdot,\cdot,\cdot,\cdot)$ and $g(\cdot,\cdot,\cdot,\cdot)$ for relevant arguments. Each of these frames requires space polynomial in n, hence the total space complexity is polynomial in n.

As for the time complexity, each call to a procedure computing a polynomial of the form $f(u,\cdot,\cdot,\cdot)$ makes at most $dk \cdot 2^d$ recursive calls to procedures computing polynomials of the form $g(u,\cdot,\cdot,\cdot)$. In turn, each of these calls makes one call to a procedure computing a polynomial of the form $f(v,\cdot,\cdot,\cdot)$ for each child v of u. It follows that the total number of calls to procedures computing polynomials of the form $f(u,\cdot,\cdot,\cdot)$ and $g(u,\cdot,\cdot,\cdot)$ is bounded by $2\cdot(dk\cdot 2^d)^k = 2^{\mathcal{O}(dk)}$. The internal work needed in each recursive call is bounded by $2^{\mathcal{O}(d)}\cdot n^{\mathcal{O}(1)}$. As T has n vertices, the total time complexity is $2^{\mathcal{O}(dk)} \cdot 2^{\mathcal{O}(d)} \cdot n^{\mathcal{O}(1)} \cdot n = 2^{\mathcal{O}(dk)} \cdot n^{\mathcal{O}(1)}$, as claimed. This concludes the proof of Lemma 5.

We note that having designed `CountElimTrees`(G, T, d), it is easy to design a similar function `CountElimForest`(G, T, d) that does not need an assumption of G being connected and where T is some elimination forest instead of an elimination tree (by using the procedure described after Lemma 3).

3.2 Utilizing `CountElimTrees`

With the description of `CountElimTrees` completed, we can describe how we can utilize it in order to construct a bounded-depth elimination tree of a graph. That is, we prove the first part of Theorem 1.

First, we lift `CountElimTrees` to a constructive procedure that still requires to be provided an auxiliary elimination tree of the graph.

▶ **Lemma 10.** *There is an algorithm* `ConstructElimForest`(G, T, d) *that, given an n-vertex graph G, an elimination forest T of G of depth at most k, and an integer d, runs in time $2^{\mathcal{O}(dk)} \cdot n^{\mathcal{O}(1)}$, uses $n^{\mathcal{O}(1)}$ space, and either correctly concludes that $\mathrm{td}(G) > d$ or returns an elimination forest of G of depth at most d.*

Proof. By treating every connected component separately, we may assume that G is connected (see the remark after Lemma 3). Thus T is an elimination tree of G.

The first step of `ConstructElimForest`(G, T, d) is calling `CountElimTrees`(G, T, d). If this call returns 0, we terminate `ConstructElimForest` and report that $\mathrm{td}(G) > d$; this is correct by Lemma 4. Otherwise we are sure that $\mathrm{td}(G) \leqslant d$, and we need to construct any elimination tree of depth at most d. In order to do so, we check, for every vertex $v \in V(G)$, whether v is a feasible candidate for the root of desired elimination tree. Note that a vertex v can be the root of an elimination tree of G of depth at most d if and only if $\mathrm{td}(G - v) < d$, or equivalently, if an only if the procedure `CountElimForest`$(G - v, T - v, d - 1)$ returns a positive value. (Here, by $T - v$ we mean the forest T with v removed and all former children of v made into children of the parent of v, or to roots in case v was a root.) As $\mathrm{td}(G) \leqslant d$, we know that for at least one vertex v, this check will return a positive outcome. Then we recursively call `ConstructElimForest`$(G - v, T - v, d - 1)$, thus obtaining an elimination forest F' of $G - v$ of depth at most $d - 1$, and we turn it into an elimination tree F of G by adding v as the new root and making it the parent of all the roots of F'. As F has depth at most d, it can be returned as the result of the procedure.

That the procedure is correct is clear. As for the time and space complexity, it is easy to see that there will be at most dn calls to the procedure `CountElimTrees` in total, because at each level of the recursion there will be at most one invocation of `CountElimTrees` per vertex of the original graph. As each of these calls uses $2^{\mathcal{O}(dk)} \cdot n^{\mathcal{O}(1)}$ time and $n^{\mathcal{O}(1)}$ space, the same complexity bounds also follow for `ConstructElimForest`. ◀

It remains to show how to lift the assumption of being provided an auxiliary elimination forest of bounded depth. For this we use the iterative compression technique.

Proof of the first part of Theorem 1. Arbitrarily enumerate the vertices of G as $v_1, \ldots, v_n$. For $i \in \{1, \ldots, n\}$, let $G_i = G[\{v_1, \ldots, v_i\}]$ be the graph induced by the first i vertices. For each $i = 1, 2, \ldots, n$ we will compute F_i, an elimination forest of G_i of depth at most d. For $i = 1$ this is trivial. Assume now that we have already computed F_i and want to compute F_{i+1}. We first construct T_{i+1}, an elimination tree of G_{i+1}, by taking F_i, adding v_{i+1}, and making v_{i+1} the parent of all the roots of F_i. Note that T_{i+1} has depth at most $d+1$. We now call `ConstructElimForest`(G_{i+1}, T_{i+1}, d). If this procedure concludes that $\mathrm{td}(G_{i+1}) > d$, then this implies that $\mathrm{td}(G) > d$ as well, and we can terminate the algorithm and provide a negative answer. Otherwise, the procedure returns an elimination forest F_{i+1} of G_{i+1} of depth at most d, with which we can proceed. Eventually, the algorithm constructs an elimination forest $F = F_n$ of $G = G_n$ of depth at most d.

The algorithm is clearly correct. Since every call to `ConstructElimForest` is supplied with an elimination forest of depth at most $d+1$, and there are at most n calls, the total time complexity is $2^{\mathcal{O}(d^2)} \cdot n^{\mathcal{O}(1)}$ and the space complexity is $n^{\mathcal{O}(1)}$, as desired. ◀

4 Randomized linear FPT algorithm

In this section we sketch the second part of Theorem 1, where we reduce the time and space complexities to linear in n at the cost of relying on randomization. There are three main reasons why the algorithm presented in the previous section does not run in time linear in n.

- First, in procedure `ConstructElimForest`, we applied `CountElimTrees` $\mathcal{O}(dn)$ times. Even if `CountElimTrees` runs in time linear in n, this gives at least a quadratic time complexity for `ConstructElimForest`.
- Second, in the iterative compression scheme we add vertices one by one and apply procedure `ConstructElimForest` n times. Again, even if `ConstructElimForest` runs in linear time, this gives at least a quadratic time complexity.
- Third, in procedure `CountElimTrees` we handle polynomials of degree at most n and with coefficients of bitsize bounded only polynomially in n. Algebraic operations on those need time polynomial in n.

In short, the second and third obstacles are mitigated as follows:

- Iterative compression is replaced by a contraction scheme of Bodlaender [1] that allows us to replace iteration with recursion, where every recursive step reduces the total number of vertices by a constant fraction, rather than peels off just one vertex.
- We observe that in `CountElimTrees`, we may care only about monomials with degrees bounded by dk, so the degrees are not a problem. As for coefficients, we hash them modulo a sufficiently large prime. This is another source of randomization.

As the techniques discussed above are rather standard, their details can be found in the full version only. Here we provide an overview of how we optimize the procedure `ConstructElimForest`, as this part contains original ideas.

Faster root recovery

We assume we have already improved the running time of `CountElimTrees` to linear. Now we are going to improve the running time of `ConstructElimForest` to linear. Recall that `ConstructElimForest` in its current version works over a connected graph G, iterates over all vertices $v \in V(G)$ and checks whether $\mathrm{td}(G - v) \leqslant d - 1$ by calling `CountElimTrees` with appropriate parameters. Such vertices v could be placed as roots of an elimination tree of G of depth at most d. Finding any feasible root is the crucial part that needs to be optimized in order to achieve a linear running time for `ConstructElimForest`. The key fact we are going to use is that the number of possible roots of optimum-depth elimination forests of a connected graph is bounded in terms of the treedepth [3, 6].

▶ **Lemma 11** (★)**.** *Suppose G is a graph whose treedepth is equal to d. Then there are at most $d^{\mathcal{O}(d)}$ vertices $v \in V(G)$ such that $\mathrm{td}(G - v) < d$.*

Observe that supposing G is connected, vertices v satisfying $\mathrm{td}(G-v) < \mathrm{td}(G)$ are exactly those that can be placed as roots of an optimum-depth elimination tree.

We are going to modify the procedure `CountElimTrees`(G, T, d) by introducing weights. Let G be a connected graph. Enumerate vertices of G as $V(G) = \{v_1, \ldots, v_n\}$ and let t_i be the number of elimination trees of G that are sensible with respect to T and in which v_i is the root. Then the result of `CountElimTrees`(G, T, d) can be expressed as $t_1 + t_2 + \ldots + t_n$. However, with a slight modification, we are able to compute $t_1\mu_1 + t_2\mu_2 + \ldots + t_n\mu_n$ for any sequence $\mu_1, \mu_2, \ldots, \mu_n \in \mathbb{Z}$. That can be achieved by multiplying by μ_i the contribution of transitions when we map v_i to the root of a monster.

Assume wishfully that there is exactly one vertex $v_i \in V(G)$ that could serve as the root of an elimination tree of G of depth d; equivalently, v_i is the only vertex such that $\mathrm{td}(G - v_i) < d$. In other words, t_j is nonzero if and only if $i = j$. Note that in such case we have $i = \frac{\sum_{j=1}^{n} j \cdot t_j}{\sum_{j=1}^{n} t_j}$. The denominator of this expression is simply the number of all elimination trees of G of depth at most d that are sensible with respect to T, while the numerator is the result of the modified version of `CountElimTrees` where we set $\mu_j = j$ for all $j \in [n]$. Hence, we can find i (that is: pinpoint the unique root) by dividing the outcomes of two calls to weighted `CountElimTrees`, instead of calling `CountElimTrees` n times, as we did previously.

Next, we lift the assumption about the uniqueness of the candidate for the root of an elimination tree. There are two key ingredients here. The first one is Lemma 11, which bounds the number of possible candidate roots for elimination trees of optimum depth. The second one is the color coding technique. The idea is to randomly color vertices into $d^{\mathcal{O}(d)}$ colors. Because there are at most $d^{\mathcal{O}(d)}$ possible roots, with high probability there will exist a color such that there is exactly one possible root in it. By modifying the idea from the previous paragraph, we can generalize it to identifying root within a color class provided there is exactly one possible root of this color.

Similarly as in the slower version, after identifying any possible root, we can delete it and recurse to connected components of the remaining part of the graph.

References

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

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

3 Jiehua Chen, Wojciech Czerwiński, Yann Disser, Andreas Emil Feldmann, Danny Hermelin, Wojciech Nadara, Marcin Pilipczuk, Michał Pilipczuk, Manuel Sorge, Bartlomiej Wróblewski, and Anna Zych-Pawlewicz. Efficient fully dynamic elimination forests with applications to detecting long paths and cycles. In *2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*, pages 796–809. SIAM, 2021. `doi:10.1137/1.9781611976465.50`.

4 Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michał Pilipczuk, Johan M. M. van Rooij, and Jakub Onufry Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011*, pages 150–159. IEEE Computer Society, 2011. `doi:10.1109/FOCS.2011.23`.

5 Wojciech Czerwiński, Wojciech Nadara, and Marcin Pilipczuk. Improved bounds for the excluded-minor approximation of treedepth. *SIAM J. Discret. Math.*, 35(2):934–947, 2021. `doi:10.1137/19M128819X`.

6 Zdeněk Dvořák, Archontia C. Giannopoulou, and Dimitrios M. Thilikos. Forbidden graphs for tree-depth. *European Journal of Combinatorics*, 33(5):969–979, 2012. EuroComb ’09. `doi:10.1016/j.ejc.2011.09.014`.

7 Martin Fürer and Huiwen Yu. Space saving by dynamic algebraization. In *9th International Computer Science Symposium in Russia, CSR 2014*, volume 8476 of *Lecture Notes in Computer Science*, pages 375–388. Springer, 2014. `doi:10.1007/978-3-319-06686-8_29`.

8 Jakub Gajarský, Stephan Kreutzer, Jaroslav Nešetřil, Patrice Ossona de Mendez, Michal Pilipczuk, Sebastian Siebertz, and Szymon Toruńczyk. First-order interpretations of bounded expansion classes. *ACM Trans. Comput. Log.*, 21(4):29:1–29:41, 2020. `doi:10.1145/3382093`.

9 Martin Grohe and Stephan Kreutzer. Methods for algorithmic meta theorems. In *AMS-ASL Joint Special Session on Model Theoretic Methods in Finite Combinatorics*, volume 558 of *Contemporary Mathematics*, pages 181–206. American Mathematical Society, 2009.

10 Falko Hegerfeld and Stefan Kratsch. Solving connectivity problems parameterized by treedepth in single-exponential time and polynomial space. In *37th International Symposium on Theoretical Aspects of Computer Science, STACS 2020*, volume 154 of *LIPIcs*, pages 29:1–29:16. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.STACS.2020.29`.

11 Tuukka Korhonen. A single-exponential time 2-approximation algorithm for treewidth. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021*, pages 184–192. IEEE, 2021. `doi:10.1109/FOCS52979.2021.00026`.

12 Łukasz Kowalik, Marcin Mucha, Wojciech Nadara, Marcin Pilipczuk, Manuel Sorge, and Piotr Wygocki. The PACE 2020 Parameterized Algorithms and Computational Experiments challenge: Treedepth. In *15th International Symposium on Parameterized and Exact Computation, IPEC 2020*, volume 180 of *LIPIcs*, pages 37:1–37:18. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.IPEC.2020.37`.

13 Wojciech Nadara, Michał Pilipczuk, and Marcin Smulewicz. Computing treedepth in polynomial space and linear fpt time. *CoRR*, abs/2205.02656, 2022. `doi:10.48550/arXiv.2205.02656`.

14 Jesper Nederlof, Michał Pilipczuk, Céline M. F. Swennenhuis, and Karol Węgrzycki. Hamiltonian Cycle parameterized by treedepth in single exponential time and polynomial space. In *46th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 12301 of *Lecture Notes in Computer Science*, pages 27–39. Springer, 2020. `doi:10.1007/978-3-030-60440-0_3`.

15 Jaroslav Nešetřil and Patrice Ossona de Mendez. Grad and classes with bounded expansion II. Algorithmic aspects. *Eur. J. Comb.*, 29(3):777–791, 2008. `doi:10.1016/j.ejc.2006.07.014`.

16 Jaroslav Nešetřil and Patrice Ossona de Mendez. *Sparsity — Graphs, Structures, and Algorithms*, volume 28 of *Algorithms and combinatorics*. Springer, 2012. `doi:10.1007/978-3-642-27875-4`.

17 Jaroslav Nešetřil and Patrice Ossona de Mendez. On low tree-depth decompositions. *Graphs Comb.*, 31(6):1941–1963, 2015. `doi:10.1007/s00373-015-1569-7`.

18 Jaroslav Nešetřil and Patrice Ossona de Mendez. A distributed low tree-depth decomposition algorithm for bounded expansion classes. *Distributed Comput.*, 29(1):39–49, 2016. doi:10.1007/s00446-015-0251-x.

19 Michael P. O'Brien and Blair D. Sullivan. Experimental evaluation of counting subgraph isomorphisms in classes of bounded expansion. *CoRR*, abs/1712.06690, 2017. arXiv:1712.06690.

20 Michał Pilipczuk and Sebastian Siebertz. Polynomial bounds for centered colorings on proper minor-closed graph classes. *J. Comb. Theory, Ser. B*, 151:111–147, 2021. doi:10.1016/j.jctb.2021.06.002.

21 Michał Pilipczuk, Sebastian Siebertz, and Szymon Toruńczyk. Parameterized circuit complexity of model-checking on sparse structures. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*, pages 789–798. ACM, 2018. doi:10.1145/3209108.3209136.

22 Michał Pilipczuk and Marcin Wrochna. On space efficiency of algorithms working on structural decompositions of graphs. *ACM Trans. Comput. Theory*, 9(4):18:1–18:36, 2018. doi:10.1145/3154856.

23 Felix Reidl, Peter Rossmanith, Fernando Sánchez Villaamil, and Somnath Sikdar. A faster parameterized algorithm for treedepth. In *41st International Colloquium on Automata, Languages, and Programming, ICALP 2014*, volume 8572 of *Lecture Notes in Computer Science*, pages 931–942. Springer, 2014. doi:10.1007/978-3-662-43948-7_77.