Efficient fully dynamic elimination forests with applications to detecting long paths and cycles*

Jiehua Chen† Wojciech Czerwiński‡ Yann Disser§ Andreas Emil Feldmann¶
Danny Hermelin‖ Wojciech Nadara‡ Marcin Pilipczuk‡ Michał Pilipczuk‡
Manuel Sorge‡ Bartłomiej Wróblewski‡ Anna Zych-Pawlewicz‡

Abstract

We present a data structure that in a dynamic graph of treedepth at most d, which is modified over time by edge insertions and deletions, maintains an optimum-height elimination forest. The data structure achieves worst-case update time $2^{\mathcal{O}(d^2)}$, which matches the best known parameter dependency in the running time of a static fpt algorithm for computing the treedepth of a graph. This improves a result of Dvořák et al. [ESA 2014], who for the same problem achieved update time $f(d)$ for some non-elementary (i.e. tower-exponential) function f. As a by-product, we improve known upper bounds on the sizes of minimal obstructions for having treedepth d from doubly-exponential in d to $d^{\mathcal{O}(d)}$.

As applications, we design new fully dynamic parameterized data structures for detecting long paths and cycles in general graphs. More precisely, for a fixed parameter k and a dynamic graph G, modified over time by edge insertions and deletions, our data structures maintain answers to the following queries:

- Does G contain a simple path on k vertices?
- Does G contain a simple cycle on at least k vertices?

In the first case, the data structure achieves amortized update time $2^{\mathcal{O}(k^2)}$. In the second case, the amortized update time is $2^{\mathcal{O}(k^4)} + \mathcal{O}(k \log n)$. In both cases we assume access to a dictionary on the edges of G.

1 Introduction

In this paper we work with dynamic data structures for graph problems. The usual setting is as follows. We are given a graph G that has an invariant vertex set, but is modified over time by edge insertions and edge deletions. The goal is to design a data structure that efficiently maintains G under such modifications, while supporting queries about some properties of interest in G. We would like to optimize the worst-case or amortized guarantees on both the update time and the query time offered by the data structure.

Classically, the research on dynamic data structures concentrates on problems that, in the static setting, are polynomial-time solvable, such as testing connectivity and maintaining minimum weight spanning trees [28, 29, 50, 32, 23, 35, 37, 40, 51, 41, 22], testing higher connectivity [28, 29, 31, 24, 22, 34], maintaining maximum matchings [46, 26, 7, 8], testing planarity [22, 30], or maintaining the distance matrix of the graph [16, 48, 1, 27]. In this work we study problems that in the classic sense are NP-hard, but are considered tractable from the point of view of *parameterized complexity*. In this paradigm, the usual goal is to design a *fixed-parameter tractable* (fpt) algorithm with running time of the form $f(k) \cdot n^{\mathcal{O}(1)}$, where n is the total input size and k is a *parameter* — an auxiliary quantitative measure of the hardness of an instance. As function f is allowed to be super-polynomial, this enables us to confine the combinatorial explosion, (seemingly) inherent in all NP-hard problems, to the specific parameter under

*This work is the result of research conducted within research project number 2017/26/D/ST6/00264 financed by National Science Centre (Anna Zych-Pawlewicz). Andreas Emil Feldmann was supported by the Czech Science Foundation GAČR (grant #17-10090Y), and by the Center for Foundations of Modern Computer Science (Charles Univ. project UNCE/SCI/004).
This work is a part of projects that have received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme: Grant Agreements no. 714704 (W. Nadara, Ma. Pilipczuk, M. Sorge) and no. 677651 (Mi. Pilipczuk).
The full version of this extended abstract can be found at [13].

†TU Wien, Austria, jiehua.chen@tuwien.ac.at.
‡University of Warsaw, Poland, {wczerwin, w.nadara, marcin.pilipczuk, michal.pilipczuk, manuel.sorge, anka}@mimuw.edu.pl, bw371883@students.mimuw.edu.pl
§TU Darmstadt, Germany, disser@mathematik.tu-darmstadt.de.
¶Charles University in Prague, Czechia, feldmann.a.e@gmail.com.
‖Ben-Gurion University of the Negev, Israel, hermelin@bgu.ac.il.

study.

The idea of parameterized measurement of complexity can be also applied to dynamic data structures: just use auxiliary parameters in upper bounds on the update and query times, allowing exponential dependence in case the considered problem is NP-hard in the classic sense. Despite the naturalness of this concept, so far there has been little systematic work on parameterized dynamic data structures. Here, the main point of reference is the work of Alman et al. [2], who considered a large set of problems fundamental for parameterized complexity, such as Vertex Cover, Hitting Set, Feedback Vertex Set, or Long Path, and delivered a number of parameterized dynamic data structures for them. Following the earlier work of Iwata and Oka [33], many results of Alman et al. are based on dynamization of standard techniques of parameterized algorithms, such as branching, kernelization, or color-coding.

Apart from [2, 33], we are aware of several scattered works on parameterized dynamic data structures; some of them will be mentioned later on. However, roughly speaking, parameterized dynamic data structures present in the literature achieve update and query times of the forms (here, k is always the solution size):

- $f(k)$, i.e., independent of the input size n (examples: Vertex Cover, d-Hitting Set for fixed d [2, 6]);
- $f(k) \cdot \log^c n$, where c is a universal constant (examples: Feedback Vertex Set, Long Path [2]);
- $\log^{f(k)} n$ (example: counting k-vertex patterns in sparse graphs [20]).

Moreover, there are problems for which already for some constant value of k, every dynamic data structure requires $\Omega(n^\delta)$ update time or $\Omega(n^\delta)$ query time, for some $\delta > 0$. As shown by Alman et al. [2], under certain assumptions from fine-grained complexity, the directed variant of Long Path falls into this category.

This charts an intriguing and still largely unexplored complexity landscape. In order to make this area well-founded, we are particularly interested in developing new techniques for designing parameterized data structures, specific to the dynamic setting. We contribute to this direction by developing decomposition-based techniques and applying them to the Long Path and Long Cycle problems.

Dynamic treedepth. In this work we explore a new approach to the design of parameterized data structures, which is based on *elimination forests* and the parameter *treedepth*. Here, an *elimination forest* of a graph G is a rooted forest F on the same vertex set as G, where for every edge uv of G, either u is an ancestor of v or vice versa. The *treedepth* of G is the minimum possible height of an elimination forest of G.

Treedepth can be regarded as a variant of treewidth where instead of the width of a decomposition, we measure its height; in fact, the treedepth of a graph is never larger than its treewidth. The importance of treedepth in the hierarchy of parameters has been gradually realized throughout the recent years. It is a central parameter in the theory of sparse graphs (see [42] for an overview), it has interesting combinatorial properties related to obstructions [15, 19, 36], and it was rendered an important dividing line from the points of view of logic [21] and of space/time complexity trade-offs [44]. In this work we are interested in the dynamic aspects of treedepth, and our first contribution is the following result.

Theorem 1.1. *Suppose G is a dynamic graph on n vertices that is updated by edge insertions and edge removals, subject to a promise that the treedepth of G never exceeds d. Then there is a data structure that, under such updates, maintains a minimum-height elimination forest of G using $2^{\mathcal{O}(d^2)}$ time per update in the worst case. Upon receiving an edge insertion that would break the promise, the data structure does not carry out the insertion and reports this fact. The data structure uses $\mathcal{O}(d \cdot n)$ memory.*

In fact, we are not the first to consider dynamic data structures for graphs of bounded treedepth. This problem was considered by Dvořák et al. [18], who gave a data structure with the same functionality as that provided by Theorem 1.1, but achieving update time $f(d)$ for some non-elementary function f. Recall that this means that $f(d)$ is tower-exponential: it is not bounded by the t-fold exponential function, for any constant t. The starting idea for our design of the data structure of Theorem 1.1, which will be further called the *dynamic treedepth data structure*, lies in the general strategy proposed by Dvořák et al. [18]. However, we rely on a new, deeper understanding of the combinatorics of treedepth and implement updates in a completely different way, which results in the improved update time of $2^{\mathcal{O}(d^2)}$. We include a comprehensive comparison of the approaches later in this introduction.

The $2^{\mathcal{O}(d^2)}$ update time offered by Theorem 1.1 reaches a certain limit. Namely, the fastest known static fpt algorithm for computing the treedepth of a graph, due to Reidl et al. [45], runs in time $2^{\mathcal{O}(d^2)} \cdot n$, where d is the value of the treedepth. Thus, achieving $2^{o(d^2)}$ update time in Theorem 1.1 would automatically improve the result of Reidl et al. to a $2^{o(d^2)} \cdot n$-time static algorithm, by introducing edges one by one. Interestingly, in the proof of Theorem 1.1, the $2^{\mathcal{O}(d^2)}$ update time in fact originates from applying the algorithm of Reidl et al. [45] as a black-box to a graph of size $d^{\mathcal{O}(d)}$. This is the only bottleneck preventing the improvement of

the $2^{\mathcal{O}(d^2)}$ update time. So we can actually conclude that improving this factor in the dynamic setting is *equivalent* to improving it in the static setting (up to the next bottleneck of $d^{\mathcal{O}(d)}$).

As a by-product of the combinatorial analysis leading to Theorem 1.1, we also give improved bounds on the sizes of minimal obstructions for having treedepth d. More precisely, we say that a graph G is a *minimal obstruction for treedepth* d if its treedepth is larger than d, but every proper induced subgraph of G has treedepth at most d. Note that every graph of treedepth larger than d contains some minimal obstruction for treedepth d as an induced subgraph, hence such obstructions are minimal "witnesses" for having large treedepth. Dvořák et al. [19] proved that every minimal obstruction for treedepth d has at most $2^{2^{d-1}}$ vertices, and they gave a construction of an obstruction with 2^d vertices. They also hypothesized that, in fact, every minimal obstruction for treedepth d has at most 2^d vertices. We get closer to this conjecture by showing an improved upper bound of $d^{\mathcal{O}(d)}$.

Detecting paths and cycles. We showcase the potential of the dynamic treedepth data structure by using it to design fully dynamic data structures for the Long Path and Long Cycle problems. In these problems, for a given undirected graph G and parameter k, the task is to decide whether G contains a path on k vertices or a cycle on at least k vertices, respectively. The following theorem summarizes our results.

Theorem 1.2. *Let k be a fixed parameter. Suppose G is a dynamic graph on n vertices, updated by edge insertions and edge deletions, and we are given access to a dictionary on the edges of G using δ_s memory with operations taking amortized time bounded by δ_t. Then there are data structures that, upon such updates, maintain the answers to the queries:*

- *Does G contain a simple path on k vertices?*
- *Does G contain a simple cycle on at least k vertices?*

In the first case, the data structure achieves amortized update time $2^{\mathcal{O}(k^2)} + \mathcal{O}(\delta_t)$ and uses $(n \cdot 2^{\mathcal{O}(k \log k)} + \delta_s)$ memory. In the second case, the amortized update time is $2^{\mathcal{O}(k^4)} + k^{\mathcal{O}(k^2)} \cdot \delta_t + \mathcal{O}(k \log n)$, and the memory usage is $(n \cdot 2^{\mathcal{O}(k^2 \log k)} + \delta_s)$.

Note that Theorem 1.2 concerns general graphs, not just graphs of bounded treedepth. In Theorem 1.2 we do not specify the query time, because we consider decision problems: the answer to the query is recomputed upon every update and can be later be provided in constant time. Also, we assume access to a dictionary on the edges of the graph. There are several ways of implementing such a dictionary that differ in trade-offs between time/space complexity and allowing amortization or randomization. For instance, the simplest solution — an adjacency matrix — achieves worst-case constant operation time at the cost of quadratic space complexity, while dynamic perfect hashing [17] gives linear space complexity, but guarantees only *expected amortized* constant time per operation. .

Long Path occupies a central position in parameterized complexity theory due to serving as the main protagonist in the development of several fundamental techniques: representative sets [39], treewidth-based win-win approaches [10], color-coding [3], algebraic coding or monomial testing [9, 38, 49], and kernelization lower bounds [11]. Long Cycle is less prominent in comparison, but is known to be fpt even in the directed variant [52]. As mentioned above, Long Path in the dynamic setting was already considered by Alman et al. [2]. By dynamizing the standard color-coding approach [3], they designed a data structure that uses $k! \cdot 2^{\mathcal{O}(k)} \cdot \mathsf{DC}(n)$ time per update, where $\mathsf{DC}(n)$ denotes the query/update time for a data structure maintaining dynamic connectivity. There are several implementations of dynamic connectivity, yet they all achieve an update time that is polylogarithmic in n, and actually there is an $\Omega(\log n)$ lower bound in the cell-probe model [43]; see the discussion in [2]. Thus, while Theorem 1.2 offers worse parametric factor of the update time compared to the data structure of Alman et al. [2], it completely removes the dependence on the size of the graph, which seems difficult in the approach used in [2]. We are not aware of any previous work on dynamic data structures for the Long Cycle problem.

Techniques behind Theorem 1.2. At first glance, it may seem surprising how the dynamic treedepth data structure can be helpful in designing data structures for Long Path and Long Cycle, because these data structures should work on an arbitrary dynamic graph, without any promise about the treedepth. Here, we use the following well-known connection (see e.g. [42, Proposition 6.1]): a graph of treedepth at least k always contains a path on k vertices. Hence, the answer to Long Path is non-trivial *only* if the treedepth is smaller than k; otherwise it is trivially positive. To capitalize on this observation, we use a technique of postponing invariant-breaking insertions, introduced by Eppstein et al. [22] in the context of planarity testing. Effectively, this enables us to focus on the case when the treedepth of the maintained dynamic graph is at all times bounded by k, at the cost of allowing amortization in the update time guarantees.

Next, we show that the dynamic treedepth data structure can be conveniently enriched with all sorts of dynamic programming procedures on elimination forests, so that in the dynamic setting we may maintain their

tables upon edge insertions and deletions. By doing this for the standard dynamic programming procedure for Long Path, we complete the proof of the first point of Theorem 1.2.

When working out this part of the argument, we make effort to introduce a convenient language for formulating dynamic programming on elimination forests that combines well with the dynamic treedepth data structure. This is because we expect these parts of our work to be of a wider applicability. In fact, we consider this to be one of the most important conceptual messages of this paper: dynamic programming on graphs of bounded treedepth can be efficiently maintained in the dynamic setting, and this is a *technique* for the design of parameterized data structures.

We remark that the data structure of Dvořák et al. [18] can be similarly combined with dynamic programming. In fact, they show that for every fixed problem definable in *Monadic Second Order logic* MSO_2 (see [14, Section 7.4.1] for an introduction), the answer to this problem can be maintained together with the dynamic treedepth data structure within the same complexity; that is, with update time $f(d)$ for a non-elementary function f. This is also the case for our data structure. Since the Long Path problem is expressible in MSO_2, it is possible to derive a data structure for dynamic Long Path with amortized update time $f(k)$, for a non-elementary function f, by combining the result of Dvořák et al. [18] with the technique of Eppstein et al. [22]. Here, k is the requested vertex count of the path.

We move on to the second point of Theorem 1.2 — the data structure for Long Cycle. This requires further ideas. The main issue is that the connection with treedepth a priori fails: as witnessed by paths, there are graphs of arbitrary large treedepth and no cycles at all. However, to some extent the approach can be salvaged: it can be shown (see [42, Proposition 6.2]) that every *biconnected* graph of treedepth at least k^2 contains a simple cycle on at least k vertices. We use this combinatorial observation as follows.

Due to the technique of postponing insertions, we may assume that the maintained graph G at all times does not contain a simple cycle on at least k vertices. Then the abovementioned combinatorial fact implies that every biconnected component of G has treedepth at most k^2. Therefore, our data structure maintains a partition of G into biconnected components, and for each biconnected component H of G we maintain an elimination forest F_H of H of height at most k^2. Roughly speaking, for maintaining the partition into biconnected components, we use the top trees data structure of Alstrup et al. [4, 5], which introduces the $\mathcal{O}(k \log n)$ factor to the update time. The forests F_H for biconnected components H are maintained using the dynamic treedepth data structures for $d = k^2$. Observe that, upon edge insertions and removals, the biconnected components of the graph may merge or split. For this, we need to design appropriate merge and split procedures for the dynamic treedepth data structures. Fortunately, our understanding of the combinatorics of treedepth allows this, at the cost of significant technical effort.

Lower bounds. Observe that the update time offered by Theorem 1.2 for the Long Cycle problem contains an $\mathcal{O}(\log n)$ factor. This is in fact unavoidable: a data structure for detecting simple cycles of length at least 3 (aka just cycles) can be used for maintaining dynamic connectivity in forests, for which there is an $\Omega(\log n)$ lower bound in the cell-probe model [43]. See Corollary 12.5 in the full version [13] for a formal derivation of this result. Thus, there is a qualitative difference between Long Path and Long Cycle in the dynamic setting: the first problem admits a data structure with amortized update time independent of n, while in the second factors linear in $\log n$ are necessary in the update time guarantees.

Here, let us point out another curious application of the data structure offered by Theorem 1.1. Using it, it is very easy to implement connectivity queries (whether given vertices u and v are in the same connected component) in time $\mathcal{O}(d)$: it will be always the case that the maintained forest F has one tree per each connected component of G, so it suffices to check whether u and v are in the same tree of F, which can be done by following parent pointers to respective roots. This gives a data structure for dynamic connectivity in graphs of treedepth at most d with update time $2^{\mathcal{O}(d^2)}$ and query time $\mathcal{O}(d)$. On the other hand, the $\Omega(\log n)$ lower bound for dynamic connectivity of Demaine and Pătraşcu [43] applies even to forests of paths, which can be thought of the simplest classes that do *not* have bounded treedepth. This means that in some sense, the possibility of maintaining dynamic connectivity with update and query time independent of n is tightly linked with assuming a bound on the treedepth of the considered dynamic graph.

A different lower bound methodology was proposed by Alman et al. [2]. Among other results, they proved that any data structure for the directed variant of Long Path for $k = 5$ has to assume $\Omega(n^\delta)$ query time, or $\Omega(n^\delta)$ update time, or $\Omega(n^{1+\delta})$ initialization time on an edgeless graph, for some $\delta > 0$. This lower bound is conditional, subject to a hypothesis called *ℓ-layered reachability oracle (ℓLRO) Conjecture*, which in turn is implied by the Triangle Conjecture and by the 3SUM Conjecture — assumptions commonly adopted in

fine-grained complexity. Using this technique, we give analogous lower bounds for the following variations of the considered problems:

- dynamic undirected 5-Path, where instead of asking for any 5-path, we look for a 5-path with a specified pair of endpoints; and
- dynamic undirected Exact-5-Cycle, where we ask for the existence of a cycle on *exactly* 5 vertices, instead of *at least* 5 vertices.

See Theorem 12.3 in the full version [13] for a formal statement.Note that in the static setting, all the variations mentioned above can be solved in fpt time using color coding or algebraic coding. Thus, the tractability domain for Long Path and related problems is much narrower in the dynamic setting. It seems that the combinatorial links with treedepth, heavily exploited in our approach, are a necessary ingredient without which not only the technique breaks, but the problems actually become provably hard.

Other works on parameterized dynamic data structures. To give a broader background, we review some results on dynamic data structures for parameterized problems that are not directly relevant to the motivation of our work.

Dvořák and Tůma [20] investigated the problem of counting occurrences (as induced subgraphs) of a fixed pattern graph H in a dynamic graph G that is assumed to be sparse (formally, always belongs to a fixed class of bounded expansion $\mathcal{C}$). They gave a data structure that maintains such a count with amortized update time $\mathcal{O}(\log^c n)$, where the constant c depends both on H and on the class $\mathcal{C}$. As classes of bounded treedepth have bounded expansion (see [42]), by taking H to be a path on k vertices we obtain a data structure for the dynamic k-path problem in graphs of treedepth smaller than k with amortized polylogarithmic update time, where the degree of the polylogarithm depends on k. Note that this result is significantly weaker than the one provided by us and by Alman et al. [2], though it is obtained using very different tools. The result of Dvořák and Tůma [20] is based on a data structure of Brodal and Fagerberg [12] for maintaining a bounded-outdegree orientation of a graph of bounded degeneracy, which also can be considered a dynamic parameterized data structure.

The dynamic setting for parameterized vertex cover and other vertex-deletion problems was first considered by Iwata and Oka [33]. As explained in Section 1, this work was continued and significantly extended by Alman et al. [2]. More recent advances include dynamic kernels for hitting and packing problems in set systems with very low update times [6], and work on monitoring timed automata in data streams [25]. Also, Schmidt et al. [47] investigated a combination of parameterization and the concept of DynFO. This setting is, however, somewhat different, as the main focus is on performing updates that can be described using simple logical formulas, and not necessarily executable efficiently in the classic sense.

Comparison with Dvořák et al. [18]. We present a quick overview of the approach that Dvořák et al. [18] used in their dynamic treedepth data structure. While doing this, we explain how our techniques differ and improve upon this approach. In this overview, we assume familiarity with MSO_2 and basic understanding of MSO_2-types.

We assume the following setting. We have a dynamic graph G, a fixed MSO_2 sentence φ, and a parameter d so that the treedepth of G is promised to always be upper bounded by d at all times. Our goal is to maintain a fully dynamic data structure $\mathbb{D}_\varphi[F, G]$ that maintains a recursively optimal elimination forest F of G and an answer to the query whether $G \models \varphi$. Let q be the maximum of d and the quantifier rank (i.e. maximum number of nested quantifiers) in φ. Note that thus, q is *at least* d.

With each vertex $u \in V(G)$, let us associate a graph G_u defined as in Section 2.2: G_u has vertex set $\mathsf{SReach}_{F,G}(u) \cup \mathsf{desc}_F(u)$ and edge set comprising of all the edges of G with at least one endpoint in $\mathsf{desc}_F(u)$. The idea is to associate with each vertex u the *type* of G_u, which is a piece of information that concisely describes all the properties of G_u needed both for the task of computing the treedepth, and for verifying satisfaction of φ. In the work of Dvořák et al. [18], this type is the MSO_2 type of G_u of quantifier rank q. The number of such types is bounded by a function of q only, but this function is non-elementary: it is a tower of height q, which is not smaller than d.

We note that in [18], this is presented somewhat differently. Namely, the type of G_u is maintained implicitly by storing a bounded-size *S-code*: a representantive subgraph of G_u having the same MSO_2-type of quantifier rank q as G_u, obtained by trimming superfluous subtrees in F.

Now, basic compositionality and idempotence properties of MSO_2 imply that in order to compute the type of G_u, it suffices to know the multiset of types of graphs G_v for all children v of u in F. Moreover, there is a threshold τ depending on d and φ such that within this multiset, each type appearing more than τ times can be treated as if it appeared exactly τ times. This can be related to the discussion of compositionality and idempotence in Section 2.2. Thus, for every vertex u one can compute the type of G_u from the types associated with the children in constant time, assuming we know the multiplicity of each type among the children up to the

threshold τ. This applies even though the number of children is unbounded. Intuitively, this allows efficient recomputation of the types upon modifications of the forest F, through a bucketing approach similar to that presented in Section 2.2.

Thus, the design of the data structure itself in [18] is similar to ours: every vertex remembers all its children, but these children are partitioned into buckets (represented in [18] using *merge nodes*) according to their types. Note that thus, the number of buckets per vertex is bounded by a non-elementary function of $q \geq d$. The basic idea of achieving update time independent of n is the same: during modifications of the elimination forest, we operate on whole buckets. Thus, re-attaching a whole bucket at a different place of the elimination forest can be done using a single operation in constant time.

The implementation of updates in [18] is, however, very different to ours and works roughly as follows. Suppose F is a tree for simplicity. First, one finds a *candidate new root*: a vertex that may be the new root of an optimal elimination tree after the update. Now comes the main trick: being a candidate root can be expressed by a (quite complicated) MSO_2 formula of quantifier rank d, hence we can use the types of rank $q \geq d$, stored in the data structure before the update, we can locate a candidate root. Once the new root is located, we iteratively move the new root up the tree. During this process we need to fix a bounded number of subtrees, which can done by recursion, because each of the trees that needs to be fixed is of height at least one smaller. All in all, this update thus uses a fairly convoluted recursion scheme, where the number of recursive calls heavily depends on the number of types that the data structure keeps track of.

Thus, the aspect that contributes the most to the time complexity is the number of types on which the data structure relies, which directly corresponds to the number of buckets stored per vertex. As we explained above, Dvořák et al. [18] use MSO_2 types of a quantifier rank $q \geq d$, whose number is a tower of exponentials of height q. Note that the assumption that $q \geq d$ is necessary to be able to efficiently evaluate the MSO_2 query locating a new root, which is needed in the update operation.

In our data structure, we are much more frugal when defining types of vertices for the purpose of maintaining a recursively optimal elimination forest. More precisely, we show that it is enough to classify each vertex u according to (1) the treedepth of the subgraph induced by the descendants of u, including u; and (2) the set of ancestors of u that are adjacent to u or any of its descendants. This gives only $d \cdot 2^d$ different types. Moreover, we perform the update in a different way than Dvořák et al.: to construct an elimination forest $\widehat{F}$ of the updated graph, we extract a *core* $K \subseteq V(G)$ of size $d^{\mathcal{O}(d)}$ that contains both endpoints of the updated edge, compute an optimum elimination forest F^K of $G[K]$ using the static algorithm of Reidl et al. [45] in time $2^{\mathcal{O}(d^2)}$, and construct $\widehat{F}$ by re-attaching parts of F lying outside of K to F^K. While this method is conceptually simpler than the approach used in [18], justifying the correctness requires a quite deep and technical dive into the combinatorics of treedepth and of elimination forests. This analysis is explained in Section 2.1. We remark that the concept behind the construction of the cores is analogous to that behind the construction of the S-codes in [18], but we execute it so that the size of the core is much smaller. We also use the cores in a quite different way.

As far as augmentation of the data structure with a dynamic programming procedure is concerned, this is automatic in the approach of Dvořák et al. [18] for MSO_2-expressible problems. Namely, the data structure anyway stores all the information about MSO_2-types up to quantifier rank q, so the answers to all boolean MSO_2 queries up to this quantifier rank are explicitly maintained. This, of course, comes at the cost of a huge explosion of complexity due to maintaining a partition into types that is potentially much finer than needed for the problem we are interested in. The language of configuration schemes and its implementation in the dynamic treedepth data structure via *mugs*, which we present in [13], is designed to remedy this complexity explosion. Namely, it allows one to design a dynamic programming procedure and automatically combine it with the dynamic treedepth data structure so that the overhead in the update time paid for augmentation is polynomial in the number of states.

In the following, we present a concise overview of the reasoning leading to our main results, focusing on explaining the key ideas rather than technical details. These details can be found in the full version [13].

2 Overview

2.1 Treedepth, elimination forests, and cores

In this subsection we give an overview of the material presented in the full version [13]. Our first goal is to obtain a fine combinatorial understanding of elimination forests of optimum height, so that we will be able to efficiently recompute them upon edge insertions and deletions.

Recall that an *elimination forest* (or equivalently treedepth decomposition) of a graph G is a rooted forest on the same vertex set as G satisfying the following property: for every edge uv of G, either u is an ancestor of v in F, or vice versa. Note that edges of F do

not need to be present in G. Elimination forests are graph decompositions underlying the parameter *treedepth*, defined as the minimum possible height of an elimination forest. Note that an elimination forest of a connected graph is necessarily a tree.

We will work with elimination forests that are in some sense also "locally optimal", as explained in the following definition. Here, for a forest F and vertex $u \in V(F)$, by F_u we denote the subtree of F induced by u and all its descendants. Also, $\mathsf{desc}_F(u)$ denotes the set of descendants of u in F, including u itself.

DEFINITION 2.1. *An elimination forest F of a graph G is* recursively optimal *if for every vertex u, the graph $G[\mathsf{desc}_F(u)]$ is connected and has treedepth equal to the height of F_u.*

In other words, in a recursively optimal elimination forest F, each subtree F_u is an optimum-height elimination tree of $G[\mathsf{desc}_F(u)]$. Thus, the height of a recursively optimal elimination forest F always matches the treedepth of the graph, as F needs to optimally decompose each connected component in a separate tree. It is easy to see that every graph has a recursively optimal elimination forest, and, up to technical details, such a forest can be computed in time $2^{\mathcal{O}(d^2)} \cdot n^{\mathcal{O}(1)}$ using the algorithm of Reidl et al. [45].

From now on, let us fix a graph G and assume, for simplicity, that G is connected. Let T be a recursively optimal elimination tree of G, say of height d. For a vertex u, we define the *strong reachability set* $\mathsf{SReach}(u)$ as the set $N_G(\mathsf{desc}_T(u))$, that is, $\mathsf{SReach}(u)$ consists of all (strict) ancestors of u that have neighbors among $\mathsf{desc}_T(u)$. The intuition is that $\mathsf{SReach}(u)$ is the set to which the subtree T_u is "attached" in T. In other words, if we temporarily removed T_u from T and wanted to attach it back, then the optimal way would be to find the deepest vertex m of $\mathsf{SReach}(u)$ and attach T_u by making u a child of m. In this way, the conditions in the definition of an elimination forest are satisfied, while T_u is attached as high as possible.

A *prefix* of T is an ancestor-closed subset of vertices of T. For a non-empty prefix K of T, an *appendix* of K is a vertex that does not belong to K, but whose parent already belongs to K. The set of appendices of K will be denoted by $\mathsf{App}(K)$. The following definition is the cornerstone of our analysis.

DEFINITION 2.2. *Let $q \in \mathbb{N}$. A non-empty prefix K of T is a* q-core *(of (G,T)) if the following property holds: for every appendix $a \in \mathsf{App}(K)$ and subset $X \subseteq \mathsf{SReach}(a)$ of size at most 2, a has at least q distinct siblings w in T such that $w \in K$, $X \subseteq \mathsf{SReach}(w)$, and $\mathsf{height}(T_w) \geq \mathsf{height}(T_a)$.*

Before we continue, let us verify that we can always find very small cores.

LEMMA 2.1. *For each $q \geq 2$, there is a q-core of size at most $(qd)^{\mathcal{O}(d)}$.*

Proof. [Sketch] Consider the following recursive marking procedure that can be applied to a vertex $u \in V(G)$. For each set X consisting of at most 2 ancestors of u (including u), consider all the children w of u satisfying $X \subseteq \mathsf{SReach}(u)$, and mark q of them with the largest values of $\mathsf{height}(T_w)$, or all of them if their number is smaller than q. Then apply the procedure recursively on each marked child of u. It is straightforward to see that if K comprises of all vertices that got marked after applying the procedure to the root of T, then K is a q-core. Since for a marked vertex u we also mark at most $q \cdot (1 + d + \binom{d}{2}) = \mathcal{O}(qd^2)$ children of u, and T has height at most d, it follows that $|K| \leq (qd)^{\mathcal{O}(d)}$. ∎

Let us now explain the idea behind the definition of a core. Suppose K is a q-core, $a \in \mathsf{App}(K)$, and w is a sibling of a satisfying the property from the definition for some $X = \{x, y\} \subseteq \mathsf{SReach}(a)$. Since T is recursively optimal, $G[\mathsf{desc}_T(w)]$ is connected. As $x, y \in \mathsf{SReach}(w) = N_G(\mathsf{desc}_T(w))$, we conclude that in G there is a path $P_w^{x,y}$ such that $P_w^{x,y}$ has endpoints x and y, and all the internal vertices of $P_w^{x,y}$ belong to $\mathsf{desc}_T(w)$. As we have q such siblings w, we can find q such paths $P_w^{x,y}$, and they will be pairwise internally vertex-disjoint. It can now easily be seen that, provided $q \geq d$, such a set of q paths forces that in *every* elimination tree of G of depth at most d, x and y have to be in the ancestor-descendant relation. Since none of the paths $P_w^{x,y}$ intersects $\mathsf{desc}_T(a)$, this conclusion can be drawn even if we removed all the vertices of $\mathsf{desc}_T(a)$ from G. This reasoning can be applied for every pair $\{x, y\} \subseteq \mathsf{SReach}(a)$. After fixing technical details, this amounts to the following statement.

LEMMA 2.2. *Suppose K is a d-core and T^K is any elimination tree of the graph $G[K]$ of height at most d. Then for each $a \in \mathsf{App}(K)$, the set $\mathsf{SReach}(a)$ is* straight *in T^K, that is, all the vertices of $\mathsf{SReach}(a)$ lie on one root-to-leaf path in T^K.*

Supposing that K is a d-core, let $R = T - K$ be the rooted forest obtained by removing all the vertices of K from T; see Figure 1. Note that R is an elimination forest of the graph $G - K$. Then the conclusion of Lemma 2.2 means for *every* elimination tree T^K of $G[K]$ of height at most d (possibly very different from $T[K]$), R is *attachable* to T^K in the following sense: for each tree S of R, the set $N_G(V(S))$ is straight in T^K. Recalling

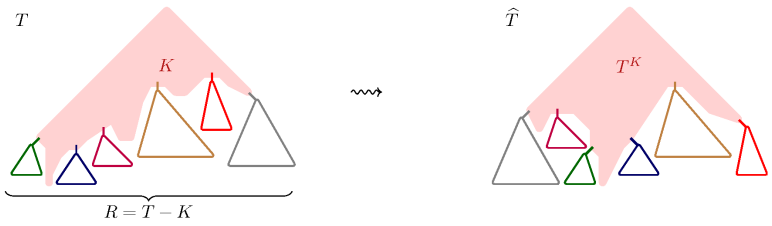

Figure 1: Treedepth cores and their replaceable elimination trees.

our previous intuition, this suggests that we can compute a new elimination forest $\widehat{T}$ of G by attaching R "below" T^K as follows (note that $\mathsf{App}(K)$ coincides with the set of roots of R). For each $a \in \mathsf{App}(K)$, we let m be the deepest vertex of $\mathsf{SReach}(a)$ in T^K, and we attach the tree $R_a = T_a$ by making a a child of m. It is straightforward to see that if R is attachable to T^K, then the tree $\widehat{T}$ obtained in this manner is indeed an elimination tree of T; we shall call it the *extension* of T^K via R.

We see that $\widehat{T}$ constructed as above is indeed an elimination forest of G, but so far we cannot say much about its height. Fortunately, from the definition of a core we can also derive useful properties in this respect. In particular, this is why in this definition we insisted that for each of the distinguished siblings w of a, the height of F_w is not smaller than the height of F_a.

The intuition is as follows. Suppose K is a $(d+1)$-core and T^K is a recursively optimal elimination forest of $G[K]$ of height at most d. Consider any $a \in \mathsf{App}(K)$. By Lemma 2.2, the set $\mathsf{SReach}(a)$ is straight in T^K. Let then m be the vertex of $\mathsf{SReach}(a)$ that is the deepest in T^K. Then, by the definition of the core we can find a set $W \subseteq K$ consisting of $d+1$ siblings w of a such that

$$\begin{aligned} m \in \mathsf{SReach}(w) \quad & \text{and} \\ \mathsf{td}(G[\mathsf{desc}_T(w)]) &= \mathsf{height}(T_w) \\ &\geq \mathsf{height}(T_a) = \mathsf{td}(G[\mathsf{desc}_T(a)]). \end{aligned}$$

Here, the first and the last equality follows from the recursive optimality of T. Through a fairly complicated inductive scheme, we can show for one of the trees T_w for $w \in W$, the graph $G[K \cap \mathsf{desc}_T(w)]$ has same treedepth as $G[\mathsf{desc}_T(w)]$ and its vertex set is fully contained in a subtree T^K_x for some x that is a child of m in T^K. This witnesses that $\mathsf{height}(T^K_x) \geq \mathsf{height}(T_w) \geq \mathsf{height}(T_a)$. We infer that attaching T_a below m in the construction of $\widehat{T}$ — the extension of T^K via $R = T - K$ — *cannot* increase the height of $\widehat{T}$ above $\mathsf{height}(T^K)$. This is because after the attachment, there anyway is a subtree rooted at a sibling of a whose height is not smaller than the height of T_a. We may conclude the following.

LEMMA 2.3. *Let K be a $(d+1)$-core and let T^K be any elimination tree of $G[K]$ of height at most d. Let $R = T - K$ and let $\widehat{T}$ be the extension of T^K via R (which is well-defined by Lemma 2.2). Then $\widehat{T}$ is a recursively optimal elimination tree of G and the height of $\widehat{T}$ is equal to the height of T^K.*

Recall that in the first place, we were interested in recomputing a recursively optimal elimination forest of a graph under edge insertions and edge deletions. Let then H be a graph obtained from G by either inserting an edge uv, or deleting an edge uv. By following the same reasoning that led us to Lemmas 2.2 and 2.3, but additionally keeping track of the modified edge uv, we can prove the following.

LEMMA 2.4. *Let K be a $(d+2)$-core of (G,T) that includes both u and v, and let T^K be any elimination tree of $H[K]$ of height at most d. Let $R = T - K$. Then R is attachable to T^K. Moreover, if $\widehat{T}$ is the extension of T^K via R, then $\widehat{T}$ is a recursively optimal elimination forest of H whose height is equal to the height of T^K.*

Note that in Lemma 2.4 we require that K is a $(d+2)$-core, while Lemma 2.3 only assumed that K is a $(d+1)$-core. This is because some of the witnessing structures, for instance paths $P^{x,y}_w$ considered in the reasoning leading to Lemma 2.2, might be affected by the removal of the edge uv. However, as this is just a single edge, adding 1 to the requirement on the core suffices for the argument to go through.

Lemma 2.4 suggests the following strategy for recomputing a recursively optimal elimination tree upon inserting or deleting an edge uv. Here, H is the updated graph.

- Using the procedure described in the proof of Lemma 2.1, compute a $(d+2)$-core K of T of size $d^{\mathcal{O}(d)}$. By modifying this procedure slightly, we may make sure that $u, v \in K$.
- Using the static algorithm of Reidl et al. [45], compute a recursively optimal elimination forest T^K of $H[K]$. Since the treedepth of $H[K]$ does not exceed the treedepth of H, which in turn does not

exceed $d+1$, this takes time $2^{\mathcal{O}(d^2)}$. Observe that if the treedepth of $H[K]$ turns out to be larger than d, then the same can be concluded about H.

- Letting $R = T - K$, compute $\widehat{T}$: the extension of T^K via R.

Then Lemma 2.4 asserts that $\widehat{T}$ is a recursively optimal elimination forests of H.

Note that this procedure readily can be implemented as a static algorithm, but the idea seems useful in the dynamic setting as well. This is because the forest R — which constitutes a vast majority of the graph, provided $d \ll n$ — first gets detached from T and then gets attached below T^K while keeping its structure intact. In the next section, our goal will be to implement this detachment and attachment so that the internal data about R does not need to be updated at all.

Observe that from Lemma 2.3 we may immediately derive the following conclusion: the subgraph induced by a d-core inherits the treedepth of the original graph.

LEMMA 2.5. *Let K be a $(d+1)$-core. Then the treedepth of $G[K]$ is equal to the treedepth of G.*

Proof. Let T^K be an elimination forest of $G[K]$ of minimum height. By Lemma 2.3, there exists an elimination forest of G of height equal to the height of T^K. This means that the treedepth of G is not larger than the treedepth of $G[K]$. As the reverse inequality is obvious, the lemma follows. ∎

We note that in reality, we prove (the formal analogs of) Lemmas 2.3 and Lemmas 2.5 in the reverse order, as (a generalization of) Lemma 2.5 is needed in the inductive scheme used in the proof of Lemma 2.3.

From Lemmas 2.1 and 2.5 we may now immediately derive the improved bounds on minimal obstructions for bounded treedepth. Recall here that a graph G is a minimal obstruction for treedepth d if the treedepth of G is larger than d, but every proper induced subgraph of G has treedepth at most d.

THEOREM 2.1. *Every minimal obstruction for treedepth d has at most $d^{\mathcal{O}(d)}$ vertices.*

Proof. Let G be a minimal obstruction for treedepth d. Clearly, G is connected. As removing one vertex decreases the treedepth by at most 1, the treedepth of G is equal to $d+1$. Let T be a recursively optimal elimination tree of G; then T has height $d+1$. By Lemma 2.1, we may find a $(d+1)$-core K of (G,T) of size $d^{\mathcal{O}(d)}$. By Lemma 2.5, the treedepth of $G[K]$ is $d+1$. Since G is a minimal obstruction, we necessarily have $K = V(G)$. So G has $d^{\mathcal{O}(d)}$ vertices. ∎

2.2 Dynamic treedepth data structure In this subsection we explain the proof of Theorem 1.1. The idea is to design the dynamic treedepth data structure so that the detachment/attachment strategy described in the previous subsection can be implemented efficiently. We note that the internal organization of information in our data structure roughly follows the general strategy of Dvořák et al. [18], but the approach we use to implement the update methods, which is based on the analysis of cores that we developed in the previous section, is completely new and different from [18]. This is the part of the reasoning that leads to the improvement.

Suppose that G is the considered graph, say connected for simplicity, and T is an elimination tree of G of depth at most d. Suppose further that G is updated by inserting or deleting an edge uv, and H is the updated graph. As outlined in the previous subsection, we should compute a $(d+2)$-core K of (G,T) of size $d^{\mathcal{O}(d)}$, recompute a recursively optimal elimination forest T^K of $H[K]$ using a static algorithm, and reattach $R := T - K$ below T^K. Consider any appendix $a \in \mathsf{App}(K)$ and recall that reattaching $R_a = T_a$ boils down to making a a child of the vertex of $\mathsf{SReach}(a)$ that is the deepest in T^K. Now comes the main observation: for any other appendix $a' \in \mathsf{App}(K)$ satisfying $\mathsf{SReach}(a') = \mathsf{SReach}(a)$, the tree $R_{a'} = T_{a'}$ will be attached at exactly the same place as R_a. Therefore, we can treat all trees R_a with the same $\mathsf{SReach}(a)$ as one "batch", which will be detached from T and reattached to $\widehat{T}$ concurrently. Here, it is not hard to see that all the trees of this batch have the same parent in T, which obviously belongs to K.

We now implement this idea algorithmically. The tree T is stored as follows. For every vertex u, we remember the parent of u in T, $\mathsf{SReach}(u)$, and $\mathsf{NeiUp}(u) := \mathsf{SReach}(u) \cap N_G(u)$; the last set is used to represent the edge set of G. Further, u remembers all its children, but these children are partitioned into *buckets* as follows: for each $X \subseteq \mathsf{SReach}(u) \cup \{u\}$ and $i \leq d$, we store the bucket

$$\mathsf{B}[u,X,i] := \{\, v \in \mathsf{children}(u) \,:\, \mathsf{SReach}(v) = X \text{ and } \mathsf{height}(T_v) = i \,\}.$$

Thus, buckets $\mathsf{B}[u,\cdot,\cdot]$ form a partition of the children of u and there are at most $2^d \cdot d$ buckets associated with u. Buckets are represented using doubly-linked lists.

Inserting or deleting the edge uv can now be implemented as follows:

- Construct a $(d+2)$-core K of size $d^{\mathcal{O}(d)}$ that includes u and v. Having access to buckets, this can be done by simulating the marking procedure presented in Lemma 2.1 in time $d^{\mathcal{O}(d)}$.
- Apply the static algorithm of Reidl et al. [45] to compute a recursively optimal elimination tree T^K

of $H[K]$. This step takes time $2^{\mathcal{O}(d^2)} \cdot |K|^{\mathcal{O}(1)} = 2^{\mathcal{O}(d^2)}$ and is the only bottleneck: all the other steps take time $d^{\mathcal{O}(d)}$. Also, if it turns out that $\mathsf{height}(T^K) > d$, then the treedepth of H exceeds d and the update should be rejected.

- Remove all vertices of K from all the buckets. This can be done in time $\mathcal{O}(|K|)$ by remembering, for each vertex, a pointer to a list element representing it in the bucket to which it belongs.
- For each $u \in K$, $X \subseteq \mathsf{SReach}(u) \cup \{u\}$, and $i \leq d$, rename the bucket $\mathsf{B}[u, X, i]$ to $\mathsf{B}[m, X, i]$, where m is the vertex of $\mathsf{SReach}(u)$ that is the deepest in T^K. Note that there are at most $|K| \cdot 2^d \cdot d = d^{\mathcal{O}(d)}$ buckets that need to be renamed in this way.
- Recompute the information for the vertices of K and place them in appropriate buckets. This can be done in time $2^{\mathcal{O}(d)} \cdot |K|^{\mathcal{O}(1)} = d^{\mathcal{O}(d)}$ by a bottom-up traversal of T^K.

The key observation is that in such an implementation, the following assertion holds: for each renamed bucket $\mathsf{B}[u, X, i]$, all the values stored for vertices of trees R_a for $a \in \mathsf{B}[u, X, i]$ do not need to be updated at all. The only exception are the parent pointers for vertices $a \in \mathsf{B}[u, X, i]$, which after renaming should all point to the new parent m. This can be easily remedied by storing one parent pointer per bucket, and changing it in a single operation. This concludes the proof of Theorem 1.1.

Configuration schemes. As mentioned in Section 1, the dynamic treedepth data structure can be conveniently augmented to maintain a run of a dynamic programming procedure on the stored elimination forest. This applies to a wide range of dynamic programming procedures, in particular those obtained for MSO_2-expressible problems through the classic connection with tree automata. In [13] we present a general formalism of *configuration schemes* that can be used to formulate such dynamic programming procedures. To keep the overview simple, we now explain how this idea applies to Long Path.

Suppose T is the maintained elimination tree and consider any $u \in V(G)$. Let $X = \mathsf{SReach}(u)$ and let G_u be the subgraph of G such that the vertex set of G_u is $X \cup \mathsf{desc}_T(u)$, while the edge set of G_u comprises of all the edges of G that have an endpoint in $\mathsf{desc}_T(u)$. Note that, thus, X forms an independent set in G_u. We now define a set $\mathfrak{C}(X)$ of *configurations* on X: a configuration c is a pair (F, j), where F is a *linear forest* (i.e. a forest of paths) on vertex set $X \cup \{s, t\}$, where s, t are special vertices, and j is an integer not larger than k (the requested vertex count of the path). Note that $|\mathfrak{C}(X)| \leq |X|^{\mathcal{O}(|X|)} \cdot (k+1) \leq d^{\mathcal{O}(d)} \cdot k$. Configuration $c = (F, j)$ is *realizable* in G_u if in G_u there is a family of paths $\{P_e : e \in E(F)\}$ of total length j such that the paths P_e are disjoint apart from endpoints in X, and the endpoints of P_e match the endpoints of e. Here, the special vertices s, t can be replaced with any vertices in G_u. Then with each $u \in V(G)$, we can associate the set $\mathsf{conf}(G_u) \subseteq \mathfrak{C}(X)$ comprising configurations realizable in G_u. Whether G contains a k-path can be determined by checking whether $\mathsf{conf}(G_r)$, where r is the root of T, contains configuration $(F_{st}, k-1)$, where F_{st} has only one edge: st.

This configuration scheme has two important properties:

- *Compositionality*: $\mathsf{conf}(G_u)$ can be computed from the multiset $\{\{\mathsf{conf}(G_v) : v \in \mathsf{children}(u)\}\}$.
- *Idempotence*: There is a threshold τ (equal to $d+2$) such that for the computation above, it is immaterial whether a configuration is realized in τ or more children of u (see [13] for a formal definition).

We show that these two basic properties alone are sufficient for augmenting the dynamic treedepth data structure so that with each $u \in V(G)$, we also implicitly store $\mathsf{conf}(G_u)$. This of course introduces factors depending on the configuration scheme to the update time, but in case of Long Path and assuming $k \leq d$, these factors are dominated by the $2^{\mathcal{O}(d^2)}$ update time of the data structure.

The augmentation is done as follows. For every bucket $\mathsf{B}[u, X, i]$ stored in the data structure, we additionally store a *mug* $\mathsf{B}[u, X, i, c]$ for each configuration $c \in \mathfrak{C}(X)$. This mug comprises all $v \in \mathsf{B}[u, X, i]$ for which $c \in \mathsf{conf}(G_v)$, and is organized as a doubly-linked list (a sublist of $\mathsf{B}[u, X, i]$). Note that each $v \in \mathsf{B}[u, X, i]$ can appear in multiple mugs, but the number of mugs is bounded by $|\mathfrak{C}(X)|$, which depends only on d and the configuration scheme in question. The set $\mathsf{conf}(G_v)$ corresponds to the set of mugs to which v belongs. It is now not hard to maintain the mugs during updates using the assumptions of compositionality and idempotence, similarly as we do for buckets.

2.3 Postponing insertions Using all the tools prepared so far for $d = k - 1$, we can implement a fully dynamic data structure that for a graph G, promised to be always of treedepth smaller than k, maintains an elimination forest of G of height smaller than k together with the answer to the query whether G contains a k-path. Upon receiving an invariant-breaking edge insertion, the data structure rejects the update and reports this. We now use the technique of Eppstein et al. [22] to turn this into a data structure working without the promise.

We maintain the dynamic treedepth data structure $\mathbb{D}$ described above, which stores a subgraph G' of G. Additionally, we have a queue Q of edges whose insertions

are postponed. We maintain the invariants: G consists of G' and all the edges stored in Q; G' has treedepth smaller than k; and if Q is non-empty, then $G' + e$ has treedepth at least k, where e is the edge at the front of Q. Thus, if Q is empty then $G' = G$. The query about a k-path can be implemented as follows: if Q is empty then we can simply query $\mathbb{D}$, and otherwise the answer is **true**, because G has treedepth at least k. When inserting an edge, we either try to insert it into $\mathbb{D}$ in case Q is empty, or we push it at the back of Q otherwise. The former case may result in rejecting the insertion and pushing the edge into Q. Finally, when deleting an edge we either delete it from $\mathbb{D}$ or from Q, depending where it is currently stored. We may now need to perform a clean-up: iteratively pop an edge from the front of Q and insert it into $\mathbb{D}$. This can take large worst-case time, but it is not hard to see that the amortized time remains constant. We use the dictionary to quickly locate edges within Q.

2.4 Detecting cycles Finally, we show how the whole machinery can be put into motion to handle also the Long Cycle problem.

Recall that a biconnected graph of treedepth at least k^2 necessarily contains a simple cycle on at least k vertices [42, Proposition 6.2]. Hence, if G does not contain a simple cycle on at least k vertices — and using the technique of Eppstein et al. [22] we can focus on this case — then every biconnected component of G has treedepth smaller than k^2. Therefore, in our data structure we maintain the partition $\mathcal{N}$ of G into biconnected components and, for each $H \in \mathcal{N}$, the dynamic treedepth data structure $\mathbb{D}[H]$ for $d = k^2$, which stores H together with some recursively optimal elimination tree T_H of height at most d.

To efficiently handle the partition $\mathcal{N}$ upon updates, we maintain a spanning forest Υ of G using the top tree data structure of Alstrup et al. [4, 5]. This data structure supports adding and removing edges from Υ in time $\mathcal{O}(\log n)$, as well as the following two queries about a pair of vertices u, v:

- What is the distance between u and v in Υ?
- If u, v are in the same connected component of Υ, return the path in Υ between u and v.

These queries take time $\mathcal{O}(\log n)$ and $\mathcal{O}(\ell \log n)$, respectively, where ℓ is the length of the reported path.

Consider now the operation of inserting an edge uv. First, we check what is the distance between u and v in Υ. If u and v are in different connected components of Υ, and therefore also of G, then we add uv to Υ and we add a new biconnected component consisting only of uv. If u and v are in the same connected component of Υ, but the distance between them in Υ is at least $k-1$, then the insertion should be rejected: uv would close a simple cycle of length at least k. Otherwise, in time $\mathcal{O}(k \log n)$ we can retrieve a path $P \subseteq \Upsilon$ with endpoints u and v, and this path has length smaller than $k-1$.

It may happen that the edges of P belong to different biconnected components of G. It is easy to see that then, the following should happen to the partition $\mathcal{N}$ after the insertion of uv: all the biconnected components containing edges of P get merged into one biconnected component. To carry this out, we iterate through the (at most $k-2$) edges on P and if two consecutive edges xy and yz belong to different biconnected components H_1 and H_2, then we merge H_1 and H_2. This requires merging the data structures $\mathbb{D}[H_1]$ and $\mathbb{D}[H_2]$, and in particular computing a recursively optimal elimination forest of the union of H_1 and H_2. We show that this can be done using the toolbox of cores as follows. Let T_1 and T_2 be the elimination trees of H_1 and H_2 stored in $\mathbb{D}[H_1]$ and $\mathbb{D}[H_2]$, respectively.

- First, we compute $(d+1)$-cores K_1 and K_2 of (H_1, T_1) and (H_2, T_2), respectively, each of size $d^{\mathcal{O}(d)} = k^{\mathcal{O}(k^2)}$. We make sure that $\{x, y\} \subseteq K_1$ and $\{y, z\} \subseteq K_2$.
- Letting $K \coloneqq K_1 \cup K_2$, we compute a recursively optimal elimination forest T^K of $G[K]$ using the static algorithm of Reidl et al. [45]. This takes time $2^{\mathcal{O}(d^2)} = 2^{\mathcal{O}(k^4)}$.
- We construct an elimination forest $\widehat{T}$ of $H_1 \cup H_2$ by attaching both the forests $T_1 - K_1$ and $T_2 - K_2$ below T^K, as in the extension operation.

It can be argued that $\widehat{T}$ constructed in this manner is a recursively optimal elimination forest of $H_1 \cup H_2$. Moreover, all of this can be done in time $2^{\mathcal{O}(d^2)} = 2^{\mathcal{O}(k^4)}$ using our representation of the dynamic treedepth data structure. Note that since the length of P is smaller than $k-1$, we perform at most $k-3$ such merges.

The operation of edge deletion is essentially symmetric: we need to split biconnected components instead of merging them, which can be done analogously. However, there is one issue: the deleted edge uv may belong to Υ, in which case it is pointless to query Υ for a u-to-v path P along which the splits should be performed. Fortunately, we show that in this case, we can retrieve a suitable path P from the data structure $\mathbb{D}[H]$, where H is the biconnected component that contains uv. Note that P has length smaller than k, for otherwise together with uv it would constitute a simple cycle of length at least k. Another caveat is that in order to maintain the invariant that Υ is spanning, after deleting uv from Υ we may need to find a replacement edge and insert it into Υ. Again fortunately, the retrieved path P provides at most $k-2$ candidates for such a replacement edge.

3 Conclusions

In this work we presented a fully dynamic data structure that for a dynamic graph G, promised to be of treedepth at most d at all times, maintains an elimination forest of G of optimum height. The data structure offers $2^{\mathcal{O}(d^2)}$ update time in the worst case. We used this result to give data structures for the dynamic variants of the Long Path and the Long Cycle problems. For Long Path, the data structure offers amortized update time $2^{\mathcal{O}(k^2)} + \mathcal{O}(\tau)$, where τ is the amortized operation time in a dictionary on the edges of the maintained graph. For Long Cycle, the amortized update time is $2^{\mathcal{O}(k^4)} + k^{\mathcal{O}(k^2)} \cdot \tau + \mathcal{O}(k \log n)$.

As argued in Section 1, the results for Long Path and Long Cycle are tight as far as the dependence on n is concerned. However, there is a lot of room for improvements of the parametric factor, that is, the dependency on k. In particular, the data structure for Long Path of Alman et al. [2] achieves a better parametric factor of $k^{\mathcal{O}(k)}$, at the cost of allowing a factor that is polylogarithmic in n. Would it be possible to obtain amortized update time $k^{\mathcal{O}(k)}$, without any additional factors depending on n? Perhaps more interestingly, static fpt algorithms for Long Path, designed for instance using color-coding [3] or algebraic coding [9, 38, 49], achieve parametric factor $2^{\mathcal{O}(k)}$ in their running times. Can one design a data structure for dynamic Long Path with amortized update time $2^{\mathcal{O}(k)} \cdot \log^c n$ for some constant c, or even $2^{\mathcal{O}(k)}$? Similar questions can be also asked about Long Cycle, where the $2^{\mathcal{O}(k^4)}$ factor seems even more amenable to improvements.

The $2^{\mathcal{O}(k^2)}$ update time offered by our dynamic data structure for Long Path is a direct consequence of the $2^{\mathcal{O}(d^2)}$ update time of the dynamic treedepth data structure. As we argued in Section 1, improving this factor is equivalent to improving the parametric factor in the complexity of the static fpt algorithm of Reidl et al. [45] for computing the treedepth of a graph. However, note that for an application to the Long Path problem, we do not necessarily need to maintain an elimination forest of optimum height; a constant-factor approximation would perfectly suffice. Unfortunately, approximation algorithms for treedepth remain largely unexplored even in the static setting, which brings us to an old question (see e.g. [15]): is there a constant-factor approximation algorithm for treedepth with running time $2^{o(d^2)} \cdot n^{\mathcal{O}(1)}$, where d is the value of the treedepth?

Finally, we hope that our work might give some insight into the problem of maintaining an approximate tree decomposition of a dynamic graph of bounded treewidth. Here, even achieving polylogarithmic-time updates would be very interesting. This direction was also mentioned both by Alman et al. [2] and by Dvořák et al. [18].

References

[1] I. Abraham, S. Chechik, and S. Krinninger. Fully dynamic all-pairs shortest paths with worst-case update-time revisited. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, pages 440–452, USA, 2017. Society for Industrial and Applied Mathematics.

[2] J. Alman, M. Mnich, and V. Vassilevska Williams. Dynamic parameterized problems and algorithms. In *Proceedings of the 44th International Colloquium on Automata, Languages, and Programming, ICALP 2017*, volume 80 of *LIPIcs*, pages 41:1–41:16. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2017.

[3] N. Alon, R. Yuster, and U. Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.

[4] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming, ICALP 1997*, volume 1256 of *Lecture Notes in Computer Science*, pages 270–280. Springer, 1997.

[5] S. Alstrup, J. Holm, K. D. Lichtenberg, and M. Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, Oct. 2005.

[6] M. Bannach, Z. Heinrich, R. Reischuk, and T. Tantau. Dynamic kernels for hitting sets and set packing. *Electronic Colloquium on Computational Complexity (ECCC)*, 26:146, 2019.

[7] S. Bhattacharya, M. Henzinger, and D. Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '16, pages 398–411, New York, NY, USA, 2016. Association for Computing Machinery.

[8] S. Bhattacharya, M. Henzinger, and D. Nanongkai. Fully dynamic approximate maximum matching and minimum vertex cover in o(log3 n) worst case update time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, pages 470–489, USA, 2017. Society for Industrial and Applied Mathematics.

[9] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Narrow sieves for parameterized paths and packings. *J. Comput. Syst. Sci.*, 87:119–139, 2017.

[10] H. L. Bodlaender. On linear time minor tests with depth-first search. *J. Algorithms*, 14(1):1–23, 1993.

[11] H. L. Bodlaender, R. G. Downey, M. R. Fellows, and D. Hermelin. On problems without polynomial kernels. *J. Comput. Syst. Sci.*, 75(8):423–434, 2009.

[12] G. S. Brodal and R. Fagerberg. Dynamic representation of sparse graphs. In *6th International Workshop on Algorithms and Data Structures, WADS 1999*, volume

1663 of *Lecture Notes in Computer Science*, pages 342–351. Springer, 1999.

[13] J. Chen, W. Czerwiński, Y. Disser, A. E. Feldmann, D. Hermelin, W. Nadara, M. Pilipczuk, M. Pilipczuk, M. Sorge, B. Wróblewski, and A. Zych-Pawlewicz. Efficient fully dynamic elimination forests with applications to detecting long paths and cycles. arXiv:2006.00571, 2020.

[14] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer, 2015.

[15] W. Czerwiński, W. Nadara, and M. Pilipczuk. Improved bounds for the excluded-minor approximation of treedepth. In *Proceedings of the 27th Annual European Symposium on Algorithms, ESA 2019*, volume 144 of *LIPIcs*, pages 34:1–34:13. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2019.

[16] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, Nov. 2004.

[17] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science, FOCS 1988*, pages 524–531. IEEE Computer Society, 1988.

[18] Z. Dvořák, M. Kupec, and V. Tůma. A dynamic data structure for MSO properties in graphs with bounded tree-depth. In *Proceedings of the 22nd Annual European Symposium on Algorithms, ESA 2014*, volume 8737 of *Lecture Notes in Computer Science*, pages 334–345. Springer, 2014.

[19] Z. Dvořák, A. C. Giannopoulou, and D. M. Thilikos. Forbidden graphs for tree-depth. *Eur. J. Comb.*, 33(5):969–979, 2012.

[20] Z. Dvořák and V. Tůma. A dynamic data structure for counting subgraphs in sparse graphs. In *Proceedings of the 13th International Symposium on Algorithms and Data Structures, WADS 2013*, volume 8037 of *Lecture Notes in Computer Science*, pages 304–315. Springer, 2013.

[21] M. Elberfeld, M. Grohe, and T. Tantau. Where First-Order and Monadic Second-Order logic coincide. *ACM Trans. Comput. Log.*, 17(4):25:1–25:18, 2016.

[22] D. Eppstein, Z. Galil, G. F. Italiano, and T. H. Spencer. Separator based sparsification. I. Planary testing and minimum spanning trees. *J. Comput. Syst. Sci.*, 52(1):3–27, 1996.

[23] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 252–257, New York, NY, USA, 1983. ACM.

[24] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.*, 26(2):484–538, Apr. 1997.

[25] A. Grez, F. Mazowiecki, M. Pilipczuk, G. Puppis, and C. Riveros. The monitoring problem for timed automata. *CoRR*, abs/2002.07049, 2020.

[26] M. Gupta and R. Peng. Fully Dynamic $(1 + \epsilon)$-Approximate Matchings. In *FOCS*. IEEE Computer Soc., 2013.

[27] M. P. Gutenberg and C. Wulff-Nilsen. *Fully-Dynamic All-Pairs Shortest Paths: Improved Worst-Case Time and Space Bounds*, pages 2562–2574.

[28] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '95, pages 519–527, New York, NY, USA, 1995. Association for Computing Machinery.

[29] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic Deterministic Fully-dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-edge, and Biconnectivity. *Journal of the ACM*, 48(4):723–760, July 2001.

[30] J. Holm and E. Rotenberg. Fully-dynamic planarity testing in polylogarithmic time. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2020, pages 167–180, New York, NY, USA, 2020. Association for Computing Machinery.

[31] J. Holm, E. Rotenberg, and M. Thorup. Dynamic bridge-finding in õ(log2 n) amortized time. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '18, pages 35–52, USA, 2018. Society for Industrial and Applied Mathematics.

[32] S.-E. Huang, D. Huang, T. Kopelowitz, and S. Pettie. Fully dynamic connectivity in o(log n(log log n)2) amortized expected time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, pages 510–520, Philadelphia, PA, USA, 2017. Society for Industrial and Applied Mathematics.

[33] Y. Iwata and K. Oka. Fast dynamic graph algorithms for parameterized problems. In *Proceedings of the 14th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2014*, volume 8503 of *Lecture Notes in Computer Science*, pages 241–252. Springer, 2014.

[34] W. Jin and X. Sun. Fully dynamic c-edge connectivity in subpolynomial time, 2020.

[35] B. M. Kapron, V. King, and B. Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, pages 1131–1142, USA, 2013. Society for Industrial and Applied Mathematics.

[36] K. Kawarabayashi and B. Rossman. A polynomial excluded-minor approximation of treedepth. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 234–246. SIAM, 2018.

[37] C. Kejlberg-Rasmussen, T. Kopelowitz, S. Pettie, and M. Thorup. Faster worst case deterministic dynamic connectivity. In *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus,*

Denmark, pages 53:1–53:15, 2016.

[38] I. Koutis. Faster algebraic algorithms for path and packing problems. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, ICALP 2008*, volume 5125 of *Lecture Notes in Computer Science*, pages 575–586. Springer, 2008.

[39] B. Monien. How to find long paths efficiently. In G. Ausiello and M. Lucertini, editors, *Analysis and Design of Algorithms for Combinatorial Problems*, volume 109 of *North-Holland Mathematics Studies*, pages 239–254. North-Holland, 1985.

[40] D. Nanongkai and T. Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and o(n$^{1/2 - \epsilon}$)-time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1122–1129, 2017.

[41] D. Nanongkai, T. Saranurak, and C. Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 950–961, 2017.

[42] J. Nešetřil and P. Ossona de Mendez. *Sparsity — Graphs, Structures, and Algorithms*, volume 28 of *Algorithms and combinatorics*. Springer, 2012.

[43] M. Patrascu and E. D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.

[44] M. Pilipczuk and M. Wrochna. On space efficiency of algorithms working on structural decompositions of graphs. *ACM Trans. Comput. Theory*, 9(4):18:1–18:36, 2018.

[45] F. Reidl, P. Rossmanith, F. Sánchez Villaamil, and S. Sikdar. A faster parameterized algorithm for treedepth. In *Proceedings of the 41st International Colloquium Automata, Languages, and Programming, ICALP 2014*, volume 8572 of *Lecture Notes in Computer Science*, pages 931–942. Springer, 2014.

[46] P. Sankowski. Faster dynamic matchings and vertex connectivity. In *SODA*, pages 118–126, 2007.

[47] J. Schmidt, T. Schwentick, N. Vortmeier, T. Zeume, and I. Kokkinis. Dynamic complexity meets parameterised algorithms. In *Proceedings of the 28th EACSL Annual Conference on Computer Science Logic, CSL 2020*, volume 152 of *LIPIcs*, pages 36:1–36:17. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2020.

[48] M. Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 112–119, New York, NY, USA, 2005. Association for Computing Machinery.

[49] R. Williams. Finding paths of length k in $O^\star(2^k)$ time. *Information Processing Letters*, 109(6):315–318, 2009.

[50] C. Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, pages 1757–1769, USA, 2013. Society for Industrial and Applied Mathematics.

[51] C. Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 1130–1143, 2017.

[52] M. Zehavi. A randomized algorithm for long directed cycle. *Inf. Process. Lett.*, 116(6):419–422, 2016.