On Dynamic Parameterized $k$-Path

Jiehua Chen† Wojciech Czerwiński‡ Yann Disser§ Andreas Emil Feldmann†

Danny Hermelin† Wojciech Nadara‡ Marcin Pilipczuk‡ Michał Pilipczuk‡

Manuel Sorge‡ Bartłomiej Wróblewski‡ Anna Zych-Pawlewicz‡

June 2, 2020

Abstract

We present a data structure that for a dynamic graph $G$, which is updated by edge insertions and removals, maintains the answer to the query whether $G$ contains a simple path on $k$ vertices with amortized update time $2^{O(k^2)}$, assuming access to a dictionary on the edges of $G$. Underlying this result lies a data structure that maintains an optimum-depth elimination forest in a dynamic graph of treedepth at most $d$ with update time $2^{O(d^2)}$. This improves a result of Dvořák et al. [ESA 2014], who for the same problem achieved update time $f(d)$ for a non-elementary function $f$. 

†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 (J. Chen, Mi. Pilipczuk). This work is also the result of research conducted within research project number 2017/26/D/ST6/00264 financed by National Science Centre of Poland (Anna Zych-Pawlewicz). Andreas Emil Feldmann was supported by the Czech Science Foundation GACR (grant #17-10090Y), and by the Center for Foundations of Modern Computer Science (Charles Univ. project UNCE/SCI/004).

‡University of Warsaw, Poland, {wczerwin,w.nadara,{marcin,michal}.pilipczuk,manuel.sorge,anka}@mimuw.edu.pl, bw371883@students.mimuw.edu.pl

§TU Darmstadt, Germany, disser@mathematik.tu-darmstadt.de.

TU Wien, Austria, jiehua.chen@tuwien.ac.at.

Ben-Gurion University of the Negev, Israel, hermelin@bgu.ac.il.
1 Introduction

The \textit{k}-path problem is probably one of the most intensively studied problems of parameterized complexity: given a graph \(G\) (directed or undirected, depending on the variant) and an integer \(k\), decide whether \(G\) contains a simple path on \(k\) vertices (not necessarily induced). The first fixed-parameter algorithm for \(k\)-path (both directed and undirected) was given by Monien \cite{Monien} as early as in 1985. This algorithm runs in time \(O(k! \cdot nm)\) and is an elegant example of the use of \textit{representative sets}, which by now became an important part of the parameterized complexity toolbox, see \cite[Chapter 12]{Fomin2019}.

Next, Bodlaender \cite{Bodlaender} proposed an algorithm with running time \(O(k! \cdot 2^k \cdot n)\), however working only for the undirected variant of the problem. The idea is to construct a depth-first-search (DFS) forest \(F\) of the graph. If \(F\) has depth at least \(k\) then it immediately uncovers a \(k\)-path. Otherwise, \(F\) can be turned into a tree decomposition of width at most \(k\), on which a dynamic programming algorithm running in time \(O(k! \cdot 2^k \cdot n)\) can be employed. Thus, Bodlaender’s approach \cite{Bodlaender} was precursive for contemporary decomposition techniques that are widespread in parameterized complexity; see \cite[Chapter 7]{Fomin2019}.

Later, \(k\)-path was again in the spotlight due to the work of Alon et al. \cite{Alon}, who introduced the \textit{color-coding} technique in order to give a randomized \((2e)^k \cdot n^{O(1)}\)-time algorithm for the problem, both in the undirected and the directed variant. This algorithm can be derandomized within time complexity \(2^{O(k)} \cdot n^{O(1)}\), and in fact it was the first algorithm that achieved a single-exponential parametric factor of the form \(2^{O(k)}\). The technique of color-coding turned out to be widely applicable in many other settings and became one of the most fundamental tools in the design of parameterized algorithms; see \cite[Chapter 5]{Fomin2019}.

Finally, the search for efficient algorithms for \(k\)-path was pivotal in the development of algebraic techniques in parameterized complexity. Koutis \cite{Koutis} and Williams \cite{Williams28} gave randomized algorithms for directed \(k\)-path with running times \(2^{3k/2} \cdot n^{O(1)}\) and \(2^k \cdot n^{O(1)}\), respectively; see also \cite{Bodlaender2010} for a more modern presentation. These developments laid foundations for \textit{algebraic coding} and \textit{monomial testing}, which by now are the core algebraic techniques of parameterized complexity; see \cite[Chapter 7]{Fomin2019}. While for undirected \(k\)-path, Björklund \cite{Bjorklund2009} improved the running time to \(1.66^k \cdot n^{O(1)}\), the question whether there is a \(c^k \cdot n^{O(1)}\)-time algorithm for directed \(k\)-path for any \(c < 2\) remains one of the most tantalizing open questions in the area.

\textbf{Dynamic \(k\)-path.} As witnessed by the discussion above, the \(k\)-path problem has served as the main protagonist in the development of several fundamental directions in parameterized complexity. In this work we are interested in another view, namely that of \textit{dynamic algorithms}. Consider the following setting. We are given a graph \(G\) whose vertex set remains invariant, but the edge set is modified over time by insertions and deletions. We wish to always be able to answer the following query: does \(G\) contain a simple path on \(k\) vertices? The goal is to develop a data structure that maintains this answer using as little time per update as possible, where the complexity is measured both in terms of \(n\) and \(k\).

This question was first investigated by Alman et al. \cite{Alman}, who presented a data structure working for the undirected variant that uses \(k! \cdot 2^{O(k)} \cdot DC(n)\) time per update and \(2^{O(k)} \cdot \log n\) time per query, where \(DC(n)\) denotes the query/update time for a data structure maintaining dynamic connectivity. In general, there are several possibilities for an implementation of dynamic connectivity, yet they all achieve an update time that is polylogarithmic in \(n\); see the discussion in \cite{Alman}. The data structure of Alman et al. \cite{Alman} is based on dynamization of the standard color-coding approach. Thus, the polylogarithmic factors — originating both from dynamic connectivity and from color-coding — seem hard to avoid. We remark that, for the directed variant of the problem, Alman et al. \cite{Alman} showed that under plausible assumptions from fine-grained complexity, any dynamic data structure needs update time \(\Omega(n^\delta)\) for some \(\delta > 0\), already for \(k = 5\). This
shows a qualitative difference between the directed and the undirected variants in the dynamic setting.

This raises a question: is there a data structure for the dynamic undirected $k$-path problem that would offer update time depending only on $k$, without any factors polylogarithmic in $n$?

**Our contribution.** In this work we answer this question affirmatively by proving the following result.

**Theorem 1 (Main result, stated informally).** There is a data structure that for a dynamic undirected graph $G$ updated by edge insertions and deletions, maintains whether $G$ contains a simple path on $k$ vertices with amortized update time $2^{O(k^2)}$. The data structure assumes access to a dictionary on the edges of $G$.

Note that Theorem 1 is stated slightly informally, as we did not specify what we exactly mean by access to a dictionary on the edges of $G$. This is a technical assumption that the edge set of $G$ can be indexed so that one can efficiently access a given edge and store some record associated with it. A simple implementation of such a dictionary would be an adjacency matrix; then each dictionary operation can be carried out in constant time. Of course the drawback is the quadratic initialization time and space complexity, but there are also more efficient possibilities: for instance dynamic perfect hashing [11] would give linear space complexity and initialization time while offering dictionary operations with amortized expected $O(1)$ time. We discuss dictionaries in Section 7; the formal statement of Theorem 1, given as Theorem 4, can be found there as well.

**Our techniques.** Instead of relying on color-coding, our idea is to resort to the earlier decomposition approach of Bodlaender [6]. In modern terms, this approach can be explained as follows. 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 observation is that every DFS forest of a graph is an elimination forest. Hence, if by applying a DFS we obtain an elimination forest $F$ whose depth is at least $k$, then we immediately see a $k$-path in the graph. Otherwise, we can employ a standard dynamic programming procedure on this forest to decide whether $G$ contains a $k$-path.

This reasoning shows an important connection between the $k$-path problem and the notion of the treedepth of a graph, which is the smallest possible depth of an elimination forest. Namely, if $G$ does not contain a $k$-path, then the treedepth of $G$ is smaller than $k$. Contrapositively, if the treedepth of $G$ is larger than $k$, then $G$ certainly contains a $k$-path. This suggests the following strategy for the dynamic $k$-path problem: design a data structure that maintains a bounded-depth elimination forest of a dynamic graph, together with tables of a suitable dynamic programming algorithm for $k$-path.

This problem —maintaining an elimination forest of a bounded treedepth graph dynamically modified by edge insertions and deletions — has been considered by Dvořák et al. [12]. They gave a data structure that maintains an elimination forest of optimum depth with update time $f(d)$, for some function $f$, provided there is a promise that the treedepth never exceeds $d$. Moreover, the data structure may maintain the answer to any fixed problem definable in Monadic Second Order logic MSO$_2$ within the same update time.

Since containing a $k$-path can be expressed using an MSO$_2$ sentence of size bounded by a function of $k$, all of this combined gives a data structure for the dynamic $k$-path problem with update time $f(k)$, however working only under the following assumption: the treedepth of the graph in question always stays lower than $k$. Note that if this invariant ceases to hold, then we know for sure that a $k$-path exists, but the data structure should also handle a situation when the treedepth fluctuates above and below $k$. We show that the promise can indeed be lifted using a simple technique of postponing invariant-breaking insertions in an auxiliary queue, which was used by Eppstein et al. [15] in the context of planarity testing. This comes at the cost of introducing amortization. Also, this is the only place where we actually use the assumption
that we are given access to a dictionary on the edge set of the graph. We remark that this idea can also be applied to two data structures given by Alman et al. [1] — for the Edge Clique Cover and Point Line Cover problems — to remove the assumption about being given a promise that the considered dynamic instance always contains a solution of bounded size.

All of this combined gives a data structure for the dynamic $k$-path problem with amortized update time $f(k)$, for some function $f$. Unfortunately, the obtained function $f$ depends on $k$ in a non-elementary way; it is a tower of height depending on $k$. This is not only caused by encoding the $k$-path problem in MSO$_2$, which could be avoided by hand-crafting an explicit dynamic programming procedure. The more serious issue is that the implementation of the data structure of Dvořák et al. [12] actually uses answers to some quite complicated MSO$_2$ queries in order to perform updates in the maintained elimination forest. Hence, the non-elementary update time is needed also for maintaining the forest itself.

Now, our approach can be summarized as follows. We present a new data structure for the problem considered by Dvořák et al. [12] that offers an improved update time of $2^{O(d^2)}$. This data structure can be combined with the standard dynamic programming approach to $k$-path to give a data structure for the $k$-path problem in a dynamic graph of treedepth smaller than $k$ with update time $2^{O(k^2)}$. Combining this with the technique of postponing insertions gives our main result, Theorem 1.

The bulk of our work is devoted to the new data structure for maintaining an optimal elimination forest of a dynamic graph of bounded treedepth. Admittedly, our approach is heavily inspired by the general strategy of Dvořák et al. [12], hence let us provide a comprehensive comparison.

Comparison with Dvořák et al. [12]. Let $F$ be an elimination forest of a graph $G$. With any vertex $u \in V(G)$ we can associate a graph $G_u$ defined as the subgraph of $G$ induced by all the ancestors and descendants of $u$ in $F$. 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 the MSO$_2$-expressible property $\varphi$ we are interested in. In the work of Dvořák et al. [12], this type is the MSO$_2$ type of $G_u$ of sufficiently high quantifier rank $q$, depending on both $d$ and $\varphi$. We note that in [12], 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 $G_u$, 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 $\tau$ depending on $d$ and $\varphi$ such that within this multiset, each type appearing more than $\tau$ times can be treated as if it appeared exactly $\tau$ times. Thus, for any vertex $u$ one can compute the type of $G_u$ from the types associated with the children in constant time, even though the number of children is unbounded. This allows efficient recomputation of the types upon modifications of the forest $F$.

The implementation of updates 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 stored in $F$ before the update to 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 using a fairly convoluted recursion scheme.

Thus, the aspect that contributes the most to the time complexity is the number of types that the data structure needs to keep track of. In [12], Dvořák et al. used MSO$_2$ types of a sufficiently high quantifier rank, whose number is bounded by a non-elementary function of $d$ and $\varphi$. In our data structure, we are much more frugal in this aspect: 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 $F'$ of the updated graph, we extract a core $K \subseteq V(G)$ of size $d^{O(d)}$ 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. [25] in time $2^{O(d^2)}$, and construct $F'$ by re-attaching parts of $F$ lying outside of $K$ to $F^K$. While this method is conceptually simpler than the approach used in [12], justifying the correctness requires a quite deep and technical dive into the combinatorics of treedepth and of elimination forests. We remark that the construction of the core is analogous to the construction of the $S$-codes in [12], but is done more carefully so that the size of the core is much smaller.

**Obstructions for treedepth.** We observe that our combinatorial analysis leading to the construction of a core of size $d^{O(d)}$ can also be used to give improved bounds on the sizes of minimal obstructions for having treedepth $d$ with respect to the induced subgraph order. 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 $G$ already 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. [13] proved that every minimal obstruction for treedepth $d$ has at most $2^{d^c}$ 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 $d^{O(d)}$.

**Other related work.** Dvořák and Tůma [14] investigated the problem of counting occurrences (as 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 $O(\log^c n)$, where the constant $c$ depends both on $H$ and on the class $\mathcal{C}$. As classes of bounded treedepth have bounded expansion (see [23]), 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 weaker than that provided by Alman et al. [1], though it is obtained using very different tools.

Besides the works mentioned above [1, 14, 12], we are aware only of a handful of other papers investigating the concept of dynamic parameterized algorithms. While the dynamic setting for parameterized vertex cover and other vertex-deletion problems was first considered by Iwata and Oka [19], the main reference point here is the work of Alman et al. [1], who performed a systematic study of a dozen of fundamental parameterized problems and gave a basic methodology for proving lower bounds. More recent advances include dynamic kernels for hitting and packing problems in set systems with very low update times [3], and work on monitoring timed automata in data streams [17]. Also, Schmidt et al. [26] 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.

**Organization.** In Section 2 we set up notation and recall basic definitions and facts about elimination forests. In Section 3 we work out the combinatorics of cores in elimination forests, while in Section 4 we digress to obtain improved bounds on the sizes of minimal obstructions for treedepth $d$. In Section 5 we design a data structure that maintains an optimal elimination forest in a dynamic graph of bounded
treedepth. In Section 6 we show how this data structure can be enriched for the purpose of maintaining the run of an auxiliary dynamic program, and we show such a dynamic program for the $k$-path problem. In Section 7 we discuss the technique of postponed insertions and gather all the tools to prove the main result. We conclude in Section 8 by discussing open problems.

2 Preliminaries

Graphs. We use standard graph notation. For a graph $G$, by $V(G)$ and $E(G)$ we denote the vertex and the edge set of $G$, respectively. The open and closed neighborhoods of a vertex $u$ are respectively defined as $N_G(u) := \{v: uv \in E(G)\}$ and $N_G[u] := N_G(u) \cup \{u\}$. We extend this notation to sets of vertices as follows: for $X \subseteq V(G)$, we write $N_G[X] := \bigcup_{x \in X} N_G[x]$ and $N_G(X) := N_G[X] - X$. For a subset of vertices $A$ of a graph $G$, the subgraph induced by $A$, denoted $G[A]$, consists of $A$ and all the edges of $G$ with both endpoints in $A$. For a vertex $u \in V(G)$, by $G - u$ we denote the graph obtained from $G$ by removing vertex $u$ and all its incident edges.

Forests. A rooted forest $F$ is a directed acyclic graph in which each vertex $u$ has at most one out-neighbor, called the parent of $u$ and denoted by parent$_F(u)$. A vertex $u$ is a root of $F$ if it has no parent, which we denote by parent$_F(u) = \bot$. The set of roots of a forest $F$ is denoted by roots$_F$. The in-neighbors of a vertex $u$ are the children of $u$ and their set is denoted by children$_F(u)$. Two vertices of $F$ that either are both roots or have the same parent are called siblings.

If a vertex $v$ is reachable from $u$ by a directed path in $F$ then $v$ is an ancestor of $u$ and $u$ is a descendant of $v$. Note that each vertex is its own ancestor and descendant. For a vertex $u$, by anc$_F(u)$ and desc$_F(u)$ we denote the sets of ancestors and descendants of $u$, respectively. By $F_u$ we denote the subtree of $F$ induced by the descendants of $u$. The depth of a vertex $u$ in $F$ is depth$_F(u) := |\text{anc}_F(u)|$ and the height of $F$ is height$(F) := \max_{u \in V(F)} \text{depth}_F(u)$.

A subset of vertices $X \subseteq V(F)$ is straight in $F$ if for all $u, v \in X$, either $u$ is an ancestor of $v$ in $F$ or $v$ is an ancestor of $u$ in $F$. Equivalently, vertices of a straight set lie on one leaf-to-root path in $F$. Here, by a root-to-leaf path in $F$ we mean a path connecting a leaf with the root of some tree in $F$.

A prefix of a forest $F$ is an ancestor-closed subset of vertices, that is, $A \subseteq V(F)$ is a prefix if $u \in A$ implies anc$_F(u) \subseteq A$. The set of appendices of a prefix $A$, denoted Append$_F(A)$, comprises all ancestor-minimal elements of $V(F) - A$, that is, vertices $u \not\in A$ such that either $u \in \text{roots}_F$ or parent$_F(u) \in A$. Note that for all $u \in \text{Append}_F(A)$, we have anc$_F(u) - \{u\} \subseteq A$.

Elimination forests and treedepth. An elimination forest of a graph $G$ is a rooted forest $F$ with $V(F) = V(G)$ such that for every edge $uv \in E(G)$, the set $\{u, v\}$ is straight in $F$. Note that if $G$ is connected then each elimination forest of $G$ has to be a tree. Hence in such a case we may also speak about elimination trees.

If $F$ is an elimination forest of a graph $G$ and $u \in V(G)$, then we define the strong reachability set of $u$:

$$\text{SReach}_{F,G}(u) := N_G(\text{desc}_F(u)).$$

We remark that the name strong reachability set comes from the theory of structural sparsity, where this concept is present and is an analogue of the definition above; see e.g. [20, 18, 29]. Note that for every vertex $u$ we have $\text{SReach}_{F,G}(u) \subseteq \text{anc}_F(u) - \{u\}$. 

5
The treedepth of a graph $G$, denoted $\text{td}(G)$, is the minimum height of an elimination forest of $G$. An elimination forest of $G$ is \textit{optimal} if its height is equal to the treedepth of $G$. We will need a more refined notion of “local” optimality, as expressed next.

**Definition 1.** An elimination forest $F$ of $G$ is recursively \textit{optimal} if for every $u \in V(G)$, we have that:
- the graph $G[\text{desc}_F(u)]$ is connected; and
- the tree $F_u$ is an optimal elimination forest of $G[\text{desc}_F(u)]$.

We remark that Dvořák et al. \cite{12} also use this definition; they call such elimination forests just “optimal”. We will also widely use a weakened version of this definition given below.

**Definition 2.** An elimination forest $F$ of $G$ is recursively \textit{connected} if for every $u \in V(G)$, we have that the graph $G[\text{desc}_F(u)]$ is connected.

Let us point out a simple, yet important detail about recursively connected elimination forests.

**Lemma 1.** Suppose $F$ is a recursively connected elimination forest of a graph $G$. Let $u$ be a vertex of $G$ and $v$ be a child of $u$ in $F$. Then $u \in S\text{Reach}_{F,G}(v)$.

**Proof.** Otherwise, no vertex of $\text{desc}_F(v)$ would have a neighbor in $\text{desc}_F(u) - \text{desc}_F(v)$, so the graph $G[\text{desc}_F(u)]$ would not be connected. \hfill \Box

Clearly, every recursively optimal elimination forest is recursively connected, as we require that explicitly in the definition. Note also that every recursively optimal elimination forest is in particular optimal, as it optimally decomposes each connected component of the graph. As shown by Reidl et al. \cite{25}, an optimal elimination forest of an $n$-vertex graph of treedepth $d$ can be computed in time $2^{O(d^2)} \cdot n$. We can use this algorithm as a black-box to show the following.

**Lemma 2.** Given an $n$-vertex graph $G$ of treedepth $d$, a recursively optimal elimination forest of $G$ can be computed in time $2^{O(d^2)} \cdot n^{O(1)}$.

**Proof.** Clearly, if $G$ is disconnected, then it suffices to run the algorithm on each connected component of $G$ separately and output the union of the obtained trees.

Suppose then that $G$ is connected. Call a vertex $u \in V(G)$ \textit{admissible} if $\text{td}(G - u) < \text{td}(G)$. Observe that such an admissible vertex always exists: if $F$ is any optimal elimination tree of $G$, then the root of $F$ is admissible. Moreover, such an admissible vertex can be found in time $2^{O(d^2)} \cdot n^2$ by testing for each $u \in V(G)$ whether $\text{td}(G - u) < \text{td}(G)$, where each such test can be done in time $2^{O(d^2)} \cdot n$ using the algorithm of Reidl et al. \cite{25}. Finally, if $u$ is admissible, then it is easy to see that a recursively optimal elimination tree of $G$ can be obtained by taking a recursively optimal elimination forest $F'$ of $G - u$ and adding $u$ as the new root, that is, making all the roots of $F'$ into children of $u$. Thus, after finding an admissible vertex $u$, one can recurse into $G - u$ and adjust the obtained elimination forest as above.

Since at every step we use $2^{O(d^2)} \cdot n^2$ time to find an admissible vertex, it is straightforward to see that the algorithm runs in time $2^{O(d^2)} \cdot n^3$. \hfill \Box

Observe that every DFS forest of a graph $G$ — a forest of calls of depth-first search started in an arbitrary vertex of each connected component of $G$ — is actually an elimination forest of $G$. Note also that if $G$ does not contain a simple path on $k$ vertices, then every DFS forest of $G$ has depth smaller than $k$, because a root-to-leaf path in a DFS forest is a simple path in the graph. Hence, we have the following.
Lemma 3 ([23], Proposition 6.1). If a graph $G$ does not contain a simple path on $k$ vertices, then $\text{td}(G) < k$.

We remark that in a weak sense, also the converse implication holds: if $G$ contains a simple path on $2^k$ vertices, then $\text{td}(G) > k$; see the discussion before Proposition 6.1 in [23]. We will not use this fact though.

Finally, we will use some basic properties of elimination forests related to the connectivity in graphs. First, the following fact is well-known.

Lemma 4. Suppose that $F$ is an elimination forest of a graph $G$ and $A \subseteq V(G)$ is such that $G[A]$ is connected. Then there exists $x \in A$ such that $A \subseteq \text{desc}_F(x)$.

Proof. Let $x$ be any vertex of $A$ that minimizes $\text{depth}_F(x)$. Let $B := A \cap \text{desc}_F(x)$ and suppose, for contradiction, that $A - B$ is non-empty. Observe that every vertex $y \in A - B$ can be neither an ancestor of $x$—by the minimality of $x$—nor a descendant of $x$—for it would be included in $B$. Hence, for each $y \in A - B$ and $z \in B$, the set $\{y, z\}$ is not straight in $F$. As $F$ is an elimination forest of $G$, this implies that there is no edge between $B$ and $A - B$. Since $B$ is non-empty due to containing $x$, this is a contradiction to the assumption that $G[A]$ is connected.

Next, vertices $x$ and $y$ of a graph $G$ are $k$-vertex-connected in $G$ if there exists $k$ internally vertex-disjoint paths with endpoints $x$ and $y$. We will use the following simple claim.

Lemma 5. Suppose that $x$ and $y$ are $d$-vertex-connected in a graph $G$. Then for every elimination forest $F$ of $G$ of depth at most $d$, the set $\{x, y\}$ is straight in $F$.

Proof. If $\{x, y\}$ was not straight in $F$, then every path connecting $x$ and $y$ would have to contain an internal vertex that belongs to $\text{anc}_F(x) \cap \text{anc}_F(y)$. Since $|\text{anc}_F(x) \cap \text{anc}_F(y)| < \text{height}(F) \leq d$, this implies that there cannot be $d$ internally vertex-disjoint paths connecting $x$ and $y$ in $G$, a contradiction.

3 Treedepth cores

We now introduce the most important definition in this work: the core of an elimination forest of a graph. Intuitively, this is a relatively small subset of vertices that retains all the relevant connectivity information that is essential for, say, treedepth computation.

Definition 3. Suppose that $F$ is an elimination forest of a graph $G$. For $q \in \mathbb{N}$, a prefix $K$ of $F$ is called a $q$-core of $(G, F)$ if the following condition holds for every vertex $u \in \text{App}_F(K)$: for each $X \subseteq \text{SReach}_{F,G}(u)$ with $|X| \leq 2$, there exist at least $q$ siblings $w$ of $u$ such that $w \in K$, $X \subseteq \text{SReach}_{F,G}(w)$, and $\text{height}(F_w) \geq \text{height}(F_u)$.

See Figure 1 for an illustration.

Before we proceed further, let us observe that in recursively optimal elimination forests we can always find $q$-cores of size bounded by a function of $q$ and the height. In the following, for a set $A$, by $(\frac{A}{\leq 2})$ we denote the set of all subsets of $A$ of size at most 2.

Lemma 6. Let $F$ be a recursively optimal elimination forest of a graph $G$ and let $d$ be the height of $F$. Then for every $q \in \mathbb{N}$, there is a $q$-core $K$ of $(G, F)$ such that

$$|K| \leq q \cdot \frac{(q(d^2 + 1))^d - 1}{q(d^2 + 1) - 1}.$$
K will be convenient: if $K$.

Then, provided that $K$.

Suppose that $K$.

Lemma 7. Suppose that $F$ is a recursively connected elimination forest of a graph $G$ and $K$ is a 1-core of $(G, F)$. Then for every $u \in K$, we have that

(i) $SReach_{F,G}(u) = N_{G[K]}(K \cap \text{desc}_F(u))$ and

(ii) the graph $G[K \cap \text{desc}_F(u)]$ is connected.
Proof. We first prove (i) by induction on height$(F_u)$. The base case when height$(F_u) = 1$ is trivial: then $K \cap \text{desc}_F(u) = \text{desc}_F(u) = \{u\}$ and $\text{anc}_F(u) \subseteq K$, hence $\text{SReach}_{F,G}(u) = N_G(u) = N_{G[K]}(u) = N_{G[K]}(K \cap \text{desc}_F(u))$.

Suppose now that height$(F_u) > 1$. The inclusion $\text{SReach}_{F,G}(u) \supseteq N_{G[K]}(K \cap \text{desc}_F(u))$ is obvious, so it remains to show the following: if some $v \in \text{anc}_F(u), v \neq u$, has a neighbor in $\text{desc}_F(u)$, then $v$ has also a neighbor in $K \cap \text{desc}_F(u)$. For this, let $a$ be a neighbor of $v$ in $\text{desc}_F(u)$. If $a = u$ then we are done, because $u \in K$. Otherwise $a \in \text{desc}_F(w)$ for some $w \in \text{children}_F(u)$, which implies that $v \in \text{SReach}_{F,G}(w)$. As $K$ is a 1-core in $F$, there exists $w' \in \text{children}_F(u) \cap K$ such that $v \in \text{SReach}_{F,G}(w')$. By applying the induction hypothesis to $w'$ we infer that $\text{SReach}_{F,G}(w') = N_{G[K]}(K \cap \text{desc}_F(w'))$, hence $v$ has a neighbor $a'$ in $K \cap \text{desc}_F(w')$. Then also $a' \in K \cap \text{desc}_F(u)$; this completes the proof of (i).

We now move to the proof of (ii), which we again perform by induction on height$(F_u)$. As before, the base case when height$(F_u) = 1$ is trivial, as then $G[K \cap \text{desc}_F(u)]$ consists of one vertex $u$. So suppose height$(F_u) > 1$. Since $F$ is recursively connected, by Lemma 1 $u$ has a neighbor in each of the sets $\text{desc}_F(w)$ for $w \in \text{children}_F(u)$. Consider any $w \in \text{children}_F(u)$ such that $K \cap \text{desc}_F(w) \neq \emptyset$. Since $K$ is a prefix of $F$, we have $w \in K$. By applying (i) and the induction hypothesis to $w$, we infer that $u$ has a neighbor in $K \cap \text{desc}_F(w)$ and $G[K \cap \text{desc}_F(w)]$ is connected. Thus, $G[K \cap \text{desc}_F(u)]$ consists of a disjoint union of connected graphs, plus there is vertex $u$ which has neighbors in each of these connected graphs. Hence $G[K \cap \text{desc}_F(u)]$ is connected as well. This proves (ii).

In subsequent lemmas we will often consider another graph that is obtained from $G$ by a minor modification within a given core $K$: we may add or remove some edges with both endpoints in $K$, but the total number of removals is bounded by $\ell \in \mathbb{N}$. For this, we introduce the following definition: for a graph $G$, elimination forest $F$ of $G$, and a prefix $K$ of $F$, a graph $H$ is a $(K, \ell)$-restricted extension of $G$ if:

- $V(H) = V(G)$;
- $E(H) = (K) = E(G) - (K)$; and
- $|E(G) - E(H)| \leq \ell$.

Note that the second condition above means that edges from the symmetric difference of $E(H)$ and $E(G)$ have both endpoints in $K$, hence for every $u \notin K$ we have $N_H(u) = N_G(u)$.

We now present the following lemma, which intuitively provides good "re-attachment points" for trees obtained by removing a core from an elimination forest.

**Lemma 8.** Let $F$ be a recursively connected elimination forest of a graph $G$ such that $F$ has height at most $d$. Let $K$ be a $(d + \ell)$-core of $(G, F)$, for some $\ell \geq 0$. Suppose that $H$ is a $(K, \ell)$-restricted extension of $G$ and let $F^K$ be any recursively connected elimination forest of $H[K]$ of height at most $d$. Then for every $u \in \text{App}_{F}(K)$, the set $\text{SReach}_{F,G}(u)$ is straight in $F^K$.

**Proof.** Consider any pair of distinct vertices $x, y \in \text{SReach}_{F,G}(u)$. Denote $W := W_K(u, \{x, y\})$; then $|W| \geq d + \ell$. Consider any $w \in W$; recall that $w \in K$ and $x, y \in \text{SReach}_{F,G}(w)$. By Lemma 7, the graph $G[K \cap \text{desc}_F(w)]$ is connected and both $x$ and $y$ have neighbors in $K \cap \text{desc}_F(w)$ in $G[K]$. This implies that in $G[K]$ there is a path $P_{xy}^w$ with endpoints $x$ and $y$ such that every internal vertex of $P_{xy}^w$ belongs to $\text{desc}_F(w)$. Since the sets in $\{\text{desc}_F(w) : w \in W\}$ are pairwise disjoint, this shows that $x$ and $y$ are $(d + \ell)$-vertex-connected in $G[K]$. Since $|E(G) - E(H)| \leq \ell$, at most $\ell$ of the paths $P_{xy}^w$ may contain an edge that does not appear in $H$, hence $x$ and $y$ are $d$-vertex-connected in $H[K]$. By Lemma 5 we infer that $\{x, y\}$ is straight in the elimination forest $F^K$. Since $x$ and $y$ were chosen arbitrarily from $\text{SReach}_{F,G}(u)$, we conclude that $\text{SReach}_{F,G}(u)$ is straight in $F^K$, as claimed.

From Lemma 8 we can derive the following claim: restricting an elimination forest to a $d$-core preserves the treedepth of each subgraph induced by a subtree.
**Lemma 9.** Let $F$ be a recursively optimal elimination forest of a graph $G$ such that $F$ is of height at most $d$, and let $K$ be a $d$-core of $(G, F)$. Then for every $v \in K$, we have $\text{td}(G[K \cap \text{desc}_F(v)]) = \text{td}(G[\text{desc}_F(v)])$.

**Proof.** Observe that for every $v \in K$, we have that $F_v$ is a recursively optimal elimination forest of $G[\text{desc}_F(v)]$ and $K \cap V(F_v)$ is a $d$-core for $(G[\text{desc}_F(v)], F_v)$. Hence, it suffices to prove the lemma for the case when $F$ is a tree and $v$ is the root of $F$. Indeed, if we succeed in this, then applying the statement for this case to each $v \in K$ yields the general statement of the lemma.

We proceed by induction on $\text{height}(F)$. The base case when $\text{height}(F) = 1$ is trivial. For the inductive step, we may assume that

$$\text{td}(G[K \cap \text{desc}_F(z)]) = \text{td}(G[\text{desc}_F(z)])$$

for each $z \in K \setminus \{v\}$.

We need to prove that $\text{td}(G[K]) = \text{td}(G)$. Clearly $\text{td}(G[K]) \leq \text{td}(G)$, so for contradiction suppose that $\text{td}(G[K]) < \text{td}(G)$. Let $F^K$ be an elimination forest of $G[K]$ of height strictly smaller than $\text{td}(G)$; in particular, $\text{height}(F^K) < d$. Our goal is to construct an elimination forest $F'$ of $G$ of height equal to the height of $F^K$, which will be a contradiction.

Consider any $u \in \text{App}_F(K)$. By Lemma 8 applied to $H = G$, the set $M := \text{SReach}_{F,G}(u)$ is straight in $F^K$. Note here that since $u \neq v$ (due to $u \notin K$ and $v \in K$) and $v$ is the only root of $F$, $u$ has a parent in $F$, which by Lemma 1 belongs to $M$. In particular $M \neq \emptyset$. Let then $m$ be the vertex of $M$ that is the deepest in $F^K$; this is well-defined because $M$ is straight in $F^K$. Further, let $\hat{M} := \text{anc}_{F,K}(m)$; as height($F^K$) < $d$ and $M$ is straight in $F^K$, we have $M \subseteq \hat{M}$ and $|M| < d$.

Let $W := W_K(u, \{m\})$; then $|W| \geq d$ as $K$ is a $d$-core. Since $|W| \geq d$, there exists $w \in W$ such that $\text{desc}_F(w) \cap \hat{M} = \emptyset$. Recall that $m \in \text{SReach}_{F,G}(w)$ and $\text{height}(F_w) \geq \text{height}(F_u)$ by the definition of $W$.

By Lemma 7, graph $G[K \cap \text{desc}_F(w)]$ is connected and $\text{N}_G[K](K \cap \text{desc}_F(w)) = \text{SReach}_{F,G}(w)$. As $G[K \cap \text{desc}_F(w)]$ is connected and $F^K$ is an elimination forest of $G[K]$, by Lemma 4 there exists $x \in K \cap \text{desc}_F(w)$ such that $K \cap \text{desc}_F(w) \subseteq \text{desc}_{F^K}(x)$. Note that $x \notin \hat{M}$ because $\text{desc}_F(w) \cap \hat{M} = \emptyset$. On the other hand, since $m \in \text{SReach}_{F,G}(w) = \text{N}_G[K](K \cap \text{desc}_F(w))$, $m$ has a neighbor in the set $K \cap \text{desc}_F(w) \subseteq \text{desc}_{F^K}(x)$. As $F^K$ is an elimination forest of $G[K]$, this implies that $x \in \text{desc}_{F^K}(m) \setminus \{m\}$. Since $K \cap \text{desc}_F(w) \subseteq \text{desc}_{F^K}(x)$ and $M \subseteq \text{anc}_{F,K}(m)$, we conclude that in $F^K$, all the vertices of $K \cap \text{desc}_F(w)$ are descendants of all the vertices of $M$, hence also of all the vertices of $\hat{M}$. In particular, $|\hat{M}| + \text{td}(G[K \cap \text{desc}_F(w)]) \leq \text{height}(F^K)$.

Since $w \neq v$, applying (1) with $z = w$ yields $\text{td}(G[K \cap \text{desc}_F(w)]) = \text{td}(G[\text{desc}_F(w)])$. Moreover, observe that we have $\text{td}(G[\text{desc}_F(w)]) = \text{height}(F_w) \geq \text{height}(F_u) = \text{td}(G[\text{desc}_F(u)])$, where the equalities follow from the recursive optimality of $F$. By combining these, we find that

$$|\hat{M}| + \text{height}(F_u) \leq \text{height}(F^K).$$

(2)

We now construct a new elimination forest $F'$ of $G$ as follows. Start with $F' = F^K$ and, for every $v \in \text{App}_F(K)$, insert the tree $F_v$ into $F'$ by making $u$ a child of $m$, defined as the deepest in $F^K$ vertex of $\text{SReach}_{F,G}(u)$. It is straightforward to see that $F'$ constructed this way is an elimination forest of $G$. Moreover, by (2) we conclude that $\text{height}(F') = \text{height}(F^K)$ (recall that $M \subseteq \hat{M}$). As $\text{height}(F^K) < \text{td}(G)$, this is a contradiction, and the inductive step is proved.

We now gather all the tools established so far to prove the following key statement. Intuitively, it says that if a graph $G$ is modified into its $(K, \ell)$-restricted extension $H$, where $K$ is a $q$-core for a sufficiently
Suppose that Lemma 10. Let $\mathcal{F}$ be a recursively optimal elimination forest of a graph $G$ of height at most $d$, and let $K$ be a $(d + \ell + 1)$-core of $(G, \mathcal{F})$ for some $\ell \geq 0$. Let $H$ be a $(K, \ell)$-restricted extension of $G$ and let $\mathcal{F}^K$ be any recursively optimal elimination forest of $H[K]$ of height at most $d$. Construct a forest $\mathcal{F}'$ as follows: start from $\mathcal{F}' = \mathcal{F}^K$ and, for every $u \in \text{App}_{\mathcal{F}}(K)$, insert the tree $\mathcal{F}_u$ into $\mathcal{F}'$ by making $u$ a child of the deepest (in $\mathcal{F}^K$) vertex of $\text{SReach}_{\mathcal{F},G}(u)$ (which is straight in $\mathcal{F}^K$ by Lemma 8), or by making $u$ a root in case $\text{SReach}_{\mathcal{F},G}(u) = \emptyset$. Then $\mathcal{F}'$ is a recursively optimal elimination forest of $H$ of height at most $d$. Moreover, for every $u \in K$, we have $\text{SReach}_{\mathcal{F}',H}(u) = \text{SReach}_{\mathcal{F}^K,H[K]}(u)$ and height($\mathcal{F}_u'$) = height($\mathcal{F}_u^K$).

**Proof.** We first prove the following claim. The reasoning is similar to that used in the proof of Lemma 9.

**Claim 1.** Suppose that $\mathcal{F}^K$ is a recursively connected (and not necessarily optimal) elimination forest of $H[K]$ of depth at most $d$ and the rest of the assumptions are as in Lemma 10. For every $u \in \text{App}_{\mathcal{F}}(K)$ there exists $z \in K$ such that $u$ and $z$ are siblings in $\mathcal{F}'$ and height($\mathcal{F}_z^K$) $\geq$ height($\mathcal{F}_u$). Moreover, for any given $p \in \text{SReach}_{\mathcal{F},H}(u)$, $z$ can be chosen so that in addition $p \in \text{SReach}_{\mathcal{F}^K,H[K]}(z)$.

**Proof.** Let us first consider the corner case when $u \in \text{roots}_{\mathcal{F}}$. Note that then $\text{SReach}_{\mathcal{F},G}(u) = \emptyset$, so there is no $p$ to be chosen and the second part of the statement holds vacuously. Let $W := W_K(u, \emptyset)$; then $|W| \geq d + \ell + 1$. By Lemmas 7 and 9, for every $w \in W$ we have that $G[K \cap \text{desc}_F(w)]$ is connected and $\text{td}(G[K \cap \text{desc}_F(w)]) = \text{height}(\mathcal{F}_w)$. Moreover, since $H$ is a $(K, \ell)$-restricted extension of $G$ and $|W| \geq d + \ell + 1$, there exists a vertex $w \in W$ such that no edge of $E(G) - E(H)$ is incident to any vertex of $\text{desc}_F(w)$, implying that $H[K \cap \text{desc}_F(w)]$ is a subgraph of $G[K \cap \text{desc}_F(w)]$. Hence, $H[K \cap \text{desc}_F(w)]$ is connected as well and $\text{td}(H[K \cap \text{desc}_F(w)]) \geq \text{td}(G[K \cap \text{desc}_F(w)])$. As $H[K \cap \text{desc}_F(w)]$ is connected
and $F^K$ is an elimination forest of $H[K]$, due to Lemma 4, there exists $z \in K \cap \text{desc}_F(w)$ such that $K \cap \text{desc}_F(w)$ is entirely contained in $F^K_z$. We conclude that

$$\text{height}(F^K_z) \geq \text{td}(H[K \cap \text{desc}_F(w)]) \geq \text{td}(G[K \cap \text{desc}_F(w)]) = \text{height}(F_w) \geq \text{height}(F_u).$$

We conclude by observing that both $u$ and $z$ are roots of $F'$, hence they are siblings in $F'$. Thus, $z$ satisfies all the requested properties.

We proceed to the proof of the main case when $u \notin \text{roots}_F$. Let $M := \text{SReach}_{G}(u)$. By Lemma 8, $M$ is straight in $F^K$. Moreover, as $u \notin \text{roots}_F$, from Lemma 1 we infer that $\text{parent}_F(u) \in M$, hence in particular $M \neq \emptyset$. Let then $m$ be the vertex of $M$ that is the deepest in $F^K$. Further, let $\hat{M} = \text{anc}_{F^K}(m)$. Then, as $M$ is straight in $F^K$, we have $M \subseteq \hat{M}$ and $|\hat{M}| \leq \text{height}(F^K) \leq d$. In the remainder of the proof we in addition fix any $p \in M$ and we look for a vertex $z$ that in addition to being a sibling of $u$ in $F'$ and satisfying $\text{height}(F^K_z) \geq \text{height}(F_u)$, also satisfies $p \in \text{SReach}_{F^K \cdot H[K]}(z)$.

Let $W := W_K(u, \{m, p\})$; then $|W| \geq d + \ell + 1$. Observe that among the vertices $w \in W$, for at most $\ell$ of them there may exist an edge in $E(G) - E(H)$ that is incident to a vertex of $\text{desc}_F(w)$. This leaves us with a set $W' \subseteq W$ of size at least $d + 1$ such that for each $w \in W'$, we have

- $E(H[\text{desc}_F(w)]) \supseteq E(G[\text{desc}_F(w)])$ and
- $N_H(\text{desc}_F(w)) \supseteq N_G(\text{desc}_F(w)) = \text{SReach}_{F,G}(w) \supseteq \{m, p\}$.

As $|W'| > d$ and $|\hat{M}| \leq d$, there exists a vertex $w \in W'$ such that $\text{desc}_F(w) \cap \hat{M} = \emptyset$.

By Lemma 7 applied to $w$, we infer that the graph $G[K \cap \text{desc}_F(w)]$ is connected and moreover $N_{G[K]}(K \cap \text{desc}_F(w)) = \text{SReach}_{F,G}(w)$. As vertices of $\text{desc}_F(w)$ are not incident to the edges of $E(G) - E(H)$, the graph $H[K \cap \text{desc}_F(w)]$ is a supergraph of $G[K \cap \text{desc}_F(w)]$, hence $H[K \cap \text{desc}_F(w)]$ is connected as well. Moreover, we have $N_{H[K]}(K \cap \text{desc}_F(w)) \supseteq N_{G[K]}(K \cap \text{desc}_F(w)) = \text{SReach}_{F,G}(w)$. As $m \in \text{SReach}_{F,G}(w)$, this means that in the graph $H[K]$, $m$ has a neighbor in $K \cap \text{desc}_F(w)$.

Since $H[K \cap \text{desc}_F(w)]$ is connected and $F^K$ is an elimination forest of $H[K]$, by Lemma 4 there exists a vertex $x \in K \cap \text{desc}_F(w)$ such that all vertices of $K \cap \text{desc}_F(w)$ are descendants of $x$ in $F^K$. Since $\text{desc}_F(w) \cap \hat{M} = \emptyset$, we in particular have $x \notin \hat{M} = \text{anc}_{F^K}(m)$. As $F^K$ is an elimination forest of $H[K]$ and in this graph, $m$ has a neighbor in $K \cap \text{desc}_F(w) \subseteq \text{desc}_{F^K}(x)$, we conclude that $x$ is a strict descendant of $m$ in $F^K$. By the construction of $F'$, we can find a vertex $z \in K$ that in $F'$ is a sibling of $u$ and an ancestor of $x$. Note that $K \cap \text{desc}_F(w) \subseteq \text{desc}_{F^K}(x) \subseteq \text{desc}_{F^K}(z)$. Moreover, we have

$$\text{SReach}_{F^K \cdot H[K]}(z) = N_{H[K]}(\text{desc}_{F^K}(z)) \supseteq N_{H[K]}(K \cap \text{desc}_F(w)) \cap \hat{M} \supseteq \text{SReach}_{F,G}(w) \cap M \supseteq \{m, p\}.$$  

It now remains to observe that

$$\text{height}(F^K_z) \geq \text{td}(H[\text{desc}_{F^K}(z)]) \geq \text{td}(H[K \cap \text{desc}_F(w)]) \geq \text{td}(G[K \cap \text{desc}_F(w)]) = \text{td}(G[\text{desc}_F(w)]) = \text{height}(F_w) \geq \text{height}(F_u).$$

Here, (3) follows from the fact that $F^K_z$ is an elimination forest for $H[\text{desc}_{F^K}(z)]$, (4) follows from Lemma 9, and (5) follows from the recursive optimality of $F$. \hfill \blacksquare
We proceed to the proof of the lemma. We need to show that the constructed forest $F'$ is a recursively optimal elimination forest of $H$ of depth at most $d$. This consists of checking three properties:

- $F'$ is an elimination forest of $H$;
- $\text{height}(F') \leq d$; and
- $F'$ is recursively optimal.

We verify these in order.

**Claim 2.** $F'$ is an elimination forest of $H$.

**Proof.** We need to prove that for every edge $ab \in E(H)$, $\{a, b\}$ is straight in $F'$. If both $a, b \in K$, then this follows from the fact that $F^K$ is an elimination forest of $H[K]$. If both $a, b \notin K$, then $ab \in E(G)$ implies that $a, b$ are both contained in a tree $F_u$ for some $u \in \text{App}_F(K)$. Then $\{a, b\}$ is straight in $F_u$ because $F$ is an elimination forest of $G$, hence $\{a, b\}$ is also straight in $F'$.

Finally, if say $a \in K$ and $b \notin K$, then $b$ is contained in $F_u$ for some $u \in \text{App}_F(K)$. Because $ab \in E(G)$, we have $a \in N_G(\text{desc}_F(u)) = S\text{Reach}_{F,G}(u)$. By the construction of $F'$, in $F'$ all the vertices of $F_u$ are descendants of all the vertices of $S\text{Reach}_{F,G}(u)$. In particular $b$ is a descendant of $a$, which concludes the proof of the claim.

**Claim 3.** $\text{height}(F') \leq d$.

**Proof.** By Claim 1 we immediately get that $\text{height}(F') = \text{height}(F^K) \leq d$.

With the next two claims we verify that $F'$ is recursively optimal.

**Claim 4.** For every $u \in V(G)$, the graph $H[\text{desc}_{F'}(u)]$ is connected.

**Proof.** If $u \notin K$, then $\text{desc}_{F'}(u) = \text{desc}_F(u)$ and $H[\text{desc}_F(u)] = G[\text{desc}_F(u)]$, so the claim follows immediately from the recursive optimality of $F$.

Suppose then that $u \in K$. By the recursive optimality of $F^K$, we have that the graph $H[\text{desc}_{F^K}(u)] = H[K \cap \text{desc}_{F'}(u)]$ is connected. Observe that the connected components of $H[\text{desc}_{F'}(u)] - K$ are exactly the graphs $G[\text{desc}_F(w)]$ for those $w \in \text{App}_F(K)$ that satisfy $w \in \text{desc}_{F'}(u)$. By the construction of $F'$, and in particular the choice of the attachment point for $w$, for each such $w$ we have that $S\text{Reach}_{F,G}(w) \cap K \cap \text{desc}_{F'}(u)$. In particular, some vertex of $\text{desc}_F(w)$ has a neighbor in $K \cap \text{desc}_{F'}(u)$ (in $G$, and equivalently in $H$). We conclude that every connected component of $H[\text{desc}_{F'}(u)] - K$ contains a vertex adjacent to $\text{desc}_{F'}(u) \cap K$ and $H[K \cap \text{desc}_{F'}(u)]$ itself is connected, so $H[\text{desc}_{F'}(u)]$ is connected as well.

**Claim 5.** For every $u \in V(G)$, $\text{height}(F'_u) = \text{td}(H[\text{desc}_{F'}(u)])$.

**Proof.** Again, if $u \notin K$, then $F'_u = F_u$ and $H[\text{desc}_F(u)] = G[\text{desc}_F(u)]$, so the claim follows immediately from the recursive optimality of $F$.

For $u \in K$, by Claim 1 and the recursive optimality of $F^K$ we have

$$\text{height}(F'_u) = \text{height}(F^K_u) = \text{td}(H[\text{desc}_{F^K}(u)]) = \text{td}(H[K \cap \text{desc}_{F'}(u)]) \leq \text{td}(H[\text{desc}_{F'}(u)]).$$

However $\text{height}(F'_u) \geq \text{td}(H[\text{desc}_{F'}(u)])$ by the definition of treedepth, so we are done.

We prove the final assertion of the lemma in the following claim.
Claim 6. For every \( u \in K \), \( SReach_{F',H}(u) = SReach_{F^K,H[K]}(u) \) and \( \text{height}(F'_u) = \text{height}(F^K_u) \).

Proof. The second assertion follows immediately from Claim 1: adding trees \( F_u \) for \( u \in \text{App}_F(K) \) when constructing \( F' \) cannot increase the height of any subtree of \( F^K \). For the first assertion, it suffices to show that \( SReach_{F',H}(u) \subseteq SReach_{F^K,H[K]}(u) \), as the reverse inclusion is obvious. Take any \( p \in SReach_{F',H}(u) \) and let \( w \) be any neighbor of \( p \) in \( \text{desc}_{F'}(u) \). If \( w \in K \) then we are done, so suppose otherwise. Then \( w \in \text{desc}_{F'}(a) \) for some \( a \in \text{App}_F(K) \), \( a \in \text{desc}_F(u) \). Note that since \( w \) and \( p \) are adjacent in \( H \) and \( w \notin K \), we in fact have \( p \in \text{Ng}(\text{desc}_F(a)) = SReach_{F,G}(a) \). By Claim 1, there exists a sibling \( z \) of \( a \) in \( F' \) such that \( z \in K \) and \( p \in SReach_{F^K,H[K]}(z) \). Since \( z \) is also a descendant of \( u \) in \( F^K \), we conclude that indeed \( p \in SReach_{F^K,H[K]}(u) \). As \( p \) was chosen arbitrarily from \( SReach_{F',H}(u) \), this shows that \( SReach_{F',H}(u) \subseteq SReach_{F^K,H[K]}(u) \) and finishes the proof. \( \square \)

As Claims 2, 3, 4, 5, and 6 verify all the postulated assertions, this finishes the proof of Lemma 10. \( \square \)

4 Obstructions for treedepth

In this section we consider bounds on the sizes of induced subgraphs that are obstructions for having low treedepth, as explained formally through the following notion.

Definition 4. A graph \( G \) is a minimal obstruction for treedepth \( d \) if \( \text{td}(G) > d \), but \( \text{td}(G - v) \leq d \) for each \( v \in V(G) \).

As mentioned in Section 1, Dvořák et al. [13] proved the following result.

Theorem 2 ([13]). Let \( d \in \mathbb{N} \). Then every minimal obstruction for treedepth \( d \) has at most \( 2^{2d-1} \) vertices. Furthermore, there exists a minimal obstruction for treedepth \( d \) that has \( 2^d \) vertices.

In fact, the lower bound of \( 2^d \) provided by Theorem 2 is obtained by showing that every acyclic minimal obstruction for treedepth \( d \) has exactly \( 2^d \) vertices, and such obstructions can be precisely characterized by means of an inductive construction. This led Dvořák et al. [13] to conjecture that in fact every minimal obstruction for treedepth \( d \) has at most \( 2^d \) vertices. We now show that from the combinatorial analysis presented in the previous section we can derive an upper bound with asymptotic growth \( d^{O(d)} \). While this still leaves a gap to the conjectured value of \( 2^d \), the new estimate is dramatically lower than the doubly-exponential upper bound provided in [13].

Theorem 3. If \( G \) is a minimal obstruction for treedepth \( d \), then the vertex count of \( G \) is at most

\[
(d+1) \cdot \frac{(d+1)((d+1)^2 + 1)}{(d+1)((d+1)^2 + 1) - 1} \in d^{O(d)}.
\]

Proof. Since \( G \) is a minimal obstruction for treedepth \( d \), \( G \) is connected and \( \text{td}(G) = d + 1 \). Let \( F \) be a recursively optimal elimination tree of \( G \); then \( \text{height}(F) = d + 1 \). Let \( r \) be the root of \( F \). By Lemma 6, we can find a \( (d + 1) \)-core \( K \) of \( (G, F) \) of size at most \( M(d) \), where \( M(d) \) is the bound provided in the theorem statement. Clearly, \( r \in K \). Applying Lemma 9 to \( v = r \), we find that \( \text{td}(G[K]) = \text{td}(G) \). Since \( G \) is a minimal obstruction for treedepth \( d \), this means that \( K = V(G) \), implying \( |V(G)| \leq M(d) \). \( \square \)

We remark that a more careful analysis of the bounds used in Lemma 6 yields a slightly better upper bound than the one claimed in Theorem 3, however with the same asymptotic growth of \( d^{O(d)} \). It remains open whether this can be improved to an upper bound of the form \( 2^{O(d)} \).
5 Data structure

In this section we present our data structure for maintaining a recursively optimal elimination forest of a dynamic graph of bounded treedepth. Before we proceed to the details, let us clarify the computation model. We assume the standard word RAM model of computation with words of bitlength $O(\log n)$, where $n$ is the vertex count of the input graph. Further, we assume that the vertices’ identifiers fit into single machine words, hence they take unit space and can be operated on in constant time. Edges are represented as pairs of vertices.

In all our data structures we assume that the initialization method takes the number $n$ as part of the input and constructs the data structure for an edgeless graph on $n$ vertices. Of course, if one wishes to initialize the structure for a graph given on input, it suffices to initialize the edgeless graph of appropriate order and add all the edges by a repeated use of the insertion method.

Description of the data structure. We now present a data structure $\mathbb{D}[F,G]$ that stores an elimination forest $F$ of a graph $G$. We will always assume that $F$ is recursively optimal and its depth is bounded by a given parameter $d$.

In $\mathbb{D}[F,G]$, each vertex $u$ is associated with a record consisting of:

- a pointer toParent$(u)$ which points to a memory cell that stores parent$_F(u)$;
- a set SReach$_{F,G}(u)$;
- a set NeiUp$_{F,G}(u)$ equal to NeiUp$_{F,G}(u) := N_G(u) \cap$ SReach$_{F,G}(u)$;
- a number height$(u)$ equal to height$(F_u)$;
- for each $X \subseteq$ SReach$_{F,G}(u) \cup \{u\}$ and $i \in \{1, \ldots, d\}$, the bucket

$$B[u,X,i] := \{w : w \in \text{children}_F(u), \text{SReach}_{F,G}(w) = X, \text{height}(F_w) = i\}.$$ 

Note that the buckets $B[u,X,i]$ form a partition of the children of $u$.

Sets SReach$_{F,G}(u)$, NeiUp$_{F,G}(u)$, as well as all the buckets $B[u,X,i]$ are stored as doubly linked lists, where a doubly linked list is represented as a pair of pointers: to its first and last element. This representation is not essential for SReach$_{F,G}(u)$ and NeiUp$_{F,G}(u)$, as these sets have sizes at most $d$ anyway, but is important for the buckets, as their sizes are unbounded.

In addition to the above, we assume that for every bucket $B[u,X,i]$ there is a single memory cell $p[u,X,i]$ that stores $u$, and that toParent$(u)$ points to $p[u,X,i]$ for each $w \in B[u,X,i]$. That is, all elements of $B[u,X,i]$ point to the same memory cell $p[u,X,i]$ for storing the information on their parent. Thus, changing the parent of the whole bucket can be done in constant time.

Furthermore, we store also the roots of $F$ in buckets as follows. It is simpler to think of the buckets (and to implement our algorithms) on a tree rather than a forest. We hence introduce an artificial symbol $\bot$, which represents an artificial root connecting all trees in $F$, i.e., all roots in $F$ are treated as its children. Now for each $i \in \{1, \ldots, d\}$, we create a bucket

$$B[\bot,\emptyset,i] := \{r : r \in \text{roots}_F, \text{height}(F_r) = i\}.$$ 

Thus, the buckets $B[\bot,\emptyset,i]$ form a partition of roots$_F$. These buckets are not associated with any vertex of $G$: they are stored at the global level in $\mathbb{D}[F,G]$, again as doubly linked lists. Also, with each of these buckets we associate a memory cell $p[\bot,\emptyset,i]$ which stores $\bot$ and is pointed to by toParent$(w)$ for all $w \in B[\bot,\emptyset,i]$.

Thus, each vertex $w \in V(G)$ is stored in exactly one bucket, namely

$$w \in B[\text{parent}_F(w), \text{SReach}_{F,G}(w), \text{height}(F_w)].$$
In addition to the record described in the beginning, for every vertex \( w \) we store a pointer to the list element corresponding to \( w \) in the doubly linked list representing the bucket in which \( w \) resides. Note that this allows removing \( w \) from this bucket in constant time.

This concludes the description of the data structure \( \mathbb{D}[F, G] \). It is clear that the initialization for an edgeless graph \( G \) can be done in \( O(n) \) time, as one only needs to initialize \( O(n) \) empty buckets.

We note that the edges of the graph \( G \) are stored in \( \mathbb{D}[F, G] \) implicitly: given \( u, v \in V(G) \), to verify whether \( u \) and \( v \) are adjacent in \( G \) it suffices to check whether \( u \in \text{NeiUp}_{F,G}(v) \) or \( v \in \text{NeiUp}_{F,G}(u) \), which can be done in \( \mathcal{O}(d) \) time. Thus, one can think of \( \mathbb{D}[F, G] \) as an implicit representation of \( G \) as well. Also, in the following we will repeatedly use the fact that, given \( u \in V(G) \) and access to \( \mathbb{D}[F, G] \), the set \( \text{anc}_{F}(u) \) can be computed in \( \mathcal{O}(d) \) time by iteratively following parent pointers from \( u \).

**Extracting cores.** We now show that, given the data structure \( \mathbb{D}[F, G] \), one can efficiently extract small cores from it. The argument essentially boils down to implementing the procedure outlined in the proof of Lemma 6 using access to \( \mathbb{D}[F, G] \).

**Lemma 11.** Suppose we have access to a data structure \( \mathbb{D}[F, G] \) that stores a recursively optimal elimination forest \( F \) of a graph \( G \) of height at most \( d \). Then one can implement a method \( \text{core}(L, q) \) which, given \( L \subseteq V(G) \) and \( q \in \mathbb{N} \), in \( (d + q + |L|)^{\mathcal{O}(d)} \) time computes a \( q \)-core \( K \) of \( (G, F) \) satisfying \( L \subseteq K \) and \( |K| \leq (d + q + |L|)^{\mathcal{O}(d)} \). Moreover, within the same running time one can also construct the graph \( G[K] \).

**Proof.** The algorithm is presented using pseudocode as Algorithm 1 in Appendix A. We implement it as method \( \text{core}(L, q) \) of \( \mathbb{D}[F, G] \); this method, given \( L \) and \( q \), outputs a \( q \)-core \( K \) with the desired properties.

The first step of \( \text{core}(L, q) \) is to compute the ancestor closure \( \hat{L} := \text{anc}_{F}(L) \); this can be done in \( \mathcal{O}(d|L|) \) time. Next, we call a recursive method \( \text{recCore}(\hat{L}, q, u) \), presented using pseudocode as Algorithm 2 in Appendix A. This method is a slight generalization of the procedure \( \text{recCore}(q, u) \) outlined in the proof of Lemma 6, where we are additionally given the set \( \hat{L} \) that should be included in the computed core. Precisely, the method \( \text{recCore}(\hat{L}, q, u) \) is given a vertex \( u \) and should output a \( q \)-core of \( (G[\text{desc}_{F}(u)], F[u]) \) that contains \( \hat{L} \cap \text{desc}_{F}(u) \); this output is represented as a doubly linked list. In the initial call, the vertex \( u \) is subsituted with the marked \( \bot \), and we follow the convention that \( \text{desc}_{F}(\bot) = V(G) \) and \( F_{\bot} = F \).

The method \( \text{recCore}(\hat{L}, q, u) \) is implemented as follows. The first step is to gather a list \( R \) consisting of children of \( u \) (or roots of \( F \) in case \( u = \bot \) ) into which the construction of the core should recurse; the implementation is in Lines 1-15 of Algorithm 2. Into this list we first include all vertices \( w \in \hat{L} \) for which \( \text{parent}_{F}(w) = u \); this can be easily done in \( \mathcal{O}(|\hat{L}|) = \mathcal{O}(d|L|) \) time. Next, for each \( X \subseteq \mathcal{SReach}_{F,G}(u) \cup \{u\} \) such that \( |X| \leq 2 \), we consider all vertices \( w \in \text{children}_{F}(u) \) (or \( w \in \text{roots}_{F} \) if \( u = \bot \) ) satisfying \( \mathcal{SReach}_{F}(w) \supseteq X \). From those vertices we add to \( R \) any \( q \) with the highest value of \( \text{height}(F_{w}) \), or all of them if their total number is smaller than \( q \). Note that, for each \( X \), this can be done in time \( \mathcal{O}(q(q + |L|) + d \cdot 2^{d}) \) by inspecting all the buckets \( B[u, Y, i] \) satisfying \( Y \supseteq X \) in any order with non-increasing \( i \), and iteratively including vertices from them until a total number of \( q \) vertices has been included. In order to avoid repetitions on the list, whenever inserting a new vertex into \( R \), we check whether it has not been included before; this takes time \( \mathcal{O}(q + |L|) \).

Once the list \( R \) is constructed, method \( \text{recCore}(\hat{L}, q, u) \) is applied recursively to each \( w \in R \). The return list is the concatenation of all the lists obtained from the recursion, with \( u \) appended in addition.

It is clear that the algorithm correctly constructs a \( q \)-core \( K \) of \( (G, F) \) which contains \( L \). As for the bound on \( |K| \), observe that in procedure \( \text{recCore}(\hat{L}, q, u) \), the total number of vertices included in the list \( R \) is bounded by \( q \cdot \left( \sum_{L \leq 2} \text{height}(F_{w}) \right) + |L| \leq q(d^{2} + 1) + |L| =: k \). The recursion depth is bounded by the
height of $F$, which is at most $d$, hence the total number of nodes in 
the recursion tree is bounded by

$$k^0 + k^1 + \ldots + k^d \in (d + q + |L|)^{O(d)}.$$ 

Observe that for each node of the recursion tree exactly one vertex is 
added to $K$. Thus we conclude that $|K| \leq (d + q + |L|)^{O(d)}$, as claimed. 
Finally, the internal computation for each node takes time 
$O(q(g + |L|) + d \cdot 2^d)$, which is asymptotically dominated by the bound 
on $|K|$. Hence, the total running time is $(d + q + |L|)^{O(d)}$.

In order to construct $G[K]$ from $K$ within the same 
asymptotic running time, it suffices to observe that 
the edge set of $G[K]$ is exactly $\bigcup_{u \in K} \{uv: v \in \text{NeiUp}_{F,G}(u)\}$, so it can be constructed in $O(d \cdot |K|)$ time 
given access to $\mathbb{D}[F,G]$.

\section*{Updates.}

We now show that the data structure can be maintained under edge insertions and removals.
The idea is as follows: first use Lemma 11 to extract a small core $K$ that contains both endpoints of 
the updated edge, then run the static algorithm of Lemma 2 to compute a recursively optimal elimination forest 
$F^K$ of the updated $G[K]$, and finally re-attach all the trees of $F - K$ to $F^K$ as in Lemma 10.

\section*{Lemma 12.}

Suppose that we have access to a data structure $\mathbb{D}[F,G]$ that stores a recursively optimal elimination 
forest $F$ of a graph $G$ such that $F$ has height at most $d$. Let $H$ be a graph obtained from $G$ by 
inserting or removing a single edge, given as input. Then one can in $2^{O(d^2)}$ time either conclude that 
$\text{td}(H) > d$, or modify $\mathbb{D}[F,G]$ to obtain a data structure $\mathbb{D}[F',H]$ where $F'$ is some recursively optimal elimination forest of $H$ 
of height at most $d$.

\section*{Proof.}

We only present the implementation of edge insertion, and at the end we discuss the tiny modifications 
needed for the implementation of edge removal. The corresponding method $\text{insert}(uv)$, where $uv$ is 
the new edge, is presented using pseudocode as Algorithm 3 in Appendix A. We now explain the 
consecutive steps. Let $H := G + uv$, that is, $H$ is obtained from $G$ by adding the edge $uv$.

First, we apply the method $\text{core}(d + 1, \{u,v\})$ provided by Lemma 11 to construct a $(d + 1)$-core $K$ 
of $(G,F)$ that contains both endpoints of $uv$. Also, we construct the graph $G[K]$. Note that $|K| \leq d^{O(d)}$ 
and this step takes $d^{O(d)}$ time.

We add the edge $uv$ to $G[K]$, thus obtaining the graph $H[K]$. Note that, since $\text{td}(G) \leq d$, we have 
$\text{td}(H) \leq d + 1$, hence also $\text{td}(H[K]) \leq d + 1$. We apply the algorithm of Lemma 2 to $H[K]$ to compute a recursively optimal elimination forest $F^K$ of $H[K]$. Note that this step takes $2^{O(d^2)} \cdot |K|^{O(1)} = 2^{O(d^2)}$ time. 
If $\text{height}(F^K) > d$ then $\text{td}(H) \geq \text{td}(H[K]) = \text{height}(F^K) > d$, hence we may terminate the algorithm 
and conclude that $\text{td}(H) > d$. So from now on we assume that $\text{height}(F^K) \leq d$.

As $H$ is a $(K,0)$-restricted extension of $G$, it remains to update the data structure $\mathbb{D}[F,G]$ to the 
structure $\mathbb{D}[F',H]$, where $F'$ is the recursively optimal elimination forest of $H$ constructed from $F^K$ 
and trees $\{F_u: u \in \text{App}_F(K)\}$ as described in Lemma 10. It will be easy to see that this part of the algorithm runs 
in total time $O(|K|)$.

First, we remove all the vertices of $K$ from all the buckets. This can be done in total time $O(|K|)$ by 
removing each vertex $v \in K$ from the bucket that it belongs to; recall here that $v$ stores a pointer to the list 
element representing it in this bucket. We then start inspecting all the buckets $B[u,X,i]$ for $u \in K \cup \{\bot\}$, 
$X \subseteq \text{SReach}_{F,G}(u) \cup \{u\}$, and $i \in \{1, \ldots, d\}$. If after the removal the bucket $B[u,X,i]$ remains nonempty, 
then it contains some vertex $a \in \text{App}(K)$. By Lemma 8, the set $X := \text{SReach}_{F,G}(a)$ has to be straight in 
$F^K$. Hence, we may compute the vertex $u' \in X$ that is the deepest in $F^K$ among the vertices of $X$; in case $X = \emptyset$, we set $u' = \bot$. Now, we can model inserting all the trees $F_a$ for $a \in B[u,X,i]$ into the forest 
$F'$ as follows: we simply rename $B[u,X,i]$ to $B[u',X,i]$ and $p[u,X,i]$ to $p[u',X,i]$, and we change the
value stored in \( p[u', X, i] \) to \( u' \). Note that in the pseudocode in Algorithm 3, this is done in Lines 11-17 by creating temporary buckets \( B'[\cdot, \cdot, \cdot] \) (originally empty) and memory cells \( p'[\cdot, \cdot, \cdot] \) to which we rename the respective old objects. Then we compose the internal data of \( \mathbb{D}[F', H] \) from these temporary objects, see Lines 18-20.

Here, let us point out one important detail: in the above operations, it will never be the case that two different buckets \( B[u_1, X, i] \) and \( B[u_2, X, i] \) will be renamed to the same new bucket \( B'[u', X, i] \). This is because the following assertion always holds in a recursively optimal forest \( F \) of a graph \( G \): for each vertex \( v \), either \( \text{SReach}_{F,G}(v) = \emptyset \), in which case \( v \) is a root, or the parent of \( v \) is equal to the deepest vertex of \( \text{SReach}_{F,G}(v) \). Indeed, this follows from the connectivity of \( G[\text{desc}_F(v')] \), where \( v' \) is the parent of \( v \). Hence, if \( B[u_1, X, i] \) and \( B[u_2, X, i] \) were simultaneously nonempty, then both \( u_1 \) and \( u_2 \) would be the deepest vertex of \( X \) in \( F \), implying \( u_1 = u_2 \).

We proceed with the description of the update operation. The key observation is that, at this point, all the data stored for every vertex \( w \notin K \), that is, \( \text{SReach}(w), \text{NeiUp}(w), \text{height}(w) \), the buckets \( B[w, \cdot, \cdot] \), the cells \( p[w, \cdot, \cdot] \), and the value stored in \( \text{toParent}(w) \), is exactly as it should be in the data structure \( \mathbb{D}[F', H] \). Therefore, it remains to update the data for all \( u \in K \). This update consists of the following. For each \( u \in K \), we add \( u \) to the bucket \( B[\text{parent}_F K(u), \text{SReach}_F K, H[K](u), \text{height}(F^K_u)] \) and make \( \text{toParent}(u) \) point to the cell \( p[\text{parent}_F K(u), \text{SReach}_F K, H[K](u), \text{height}(F^K_u)] \). Next, we update \( \text{SReach}(u) := \text{SReach}_F K, H[K](u), \text{NeiUp}(u) := \text{SReach}_F K, H[K](u) \cap N_{H[K]}(u), \text{and height}(u) := \text{height}(F^K_u) \). Note that the correctness of these updates follows from the last statement of Lemma 10. Also, they can be carried out in time polynomial in \( |K| \), because computing the right hand side of the above assignments only requires investigating the graph \( H[K] \) and its elimination forest \( F^K \).

This concludes the implementation of an edge insertion. For the implementation of an edge removal, we follow exactly the same strategy with the exception that we compute a \( (d + 2) \)-core \( K \) instead of a \( (d + 1) \)-core, because \( G - uv \) is then a \( (K, 1) \)-restricted extension of \( G \) instead of a \( (K, 0) \)-restricted extension. Also, there is no need of checking whether the height of \( F^K \) is larger than \( d \), because it is bounded by \( \text{td}((G - uv)[K]) \leq \text{td}(G - uv) \leq \text{td}(G) \leq d \). \( \square \)

### 6 Dynamic dynamic programming

In this section we show how the data structure presented in Section 5 can be enriched so that together with an elimination forest, it also maintains a run of an auxiliary dynamic program on this forest. For this, we need to understand dynamic programming on elimination forests in an abstract way, which we do using a formalism of configuration schemes. This formalism is based on the classic algebraic approach to graph decompositions, and in particular on the idea of recognizing finite-state properties of graphs of bounded treewidth through homomorphisms into finite algebras. This direction has been intensively developed in the 90s, see the book of Courcelle and Engelfriet [8] for a broad introduction. The definitional layer that we use is closest to the one used in the work of Bodlaender et al. [7].

**Boundaried graphs.** We shall assume that all vertices in all the considered graphs come from some countable reservoir of vertices \( \Omega \), say \( \Omega = \mathbb{N} \). A boundaried graph is a graph \( G \) together with a subset of vertices \( X \subseteq V(G) \), called the boundary. On boundaried graphs we can introduce two basic operators:

- For a boundaried graph \( (G, X) \) and \( x \in X \), the operator forget yields \( \text{forget}((G, X), x) := (G, X \setminus \{x\}) \).

---

18
• For two boundaried graphs \((G_1, X_1)\) and \((G_2, X_2)\) such that \(V(G_1) \cap V(G_2) = X_1 \cap X_2\), the union operator \(\oplus\) yields

\[
(G_1, X_1) \oplus (G_2, X_2) := ((V(G_1) \cup V(G_2), E(G_1) \cup E(G_2)), X_1 \cup X_2).
\]

In other words, we take the disjoint union of \(G_1\) and \(G_2\) and, for each \(x \in X_1 \cap X_2\), we fuse the copy of \(x\) from \(G_1\) with the copy of \(x\) from \(G_2\). The boundary of the output graph is \(X_1 \cup X_2\).

Note that the operator \(\text{forget}((G, X), x)\) is defined only if \(x \in X\), while the operator \((G_1, X_1) \oplus (G_2, X_2)\) is defined only if \(V(G_1) \cap V(G_2) = X_1 \cap X_2\). It is useful to think of this as of an abstract algebra on boundaried graphs, endowed with operations \(\text{forget}\) and \(\oplus\).

**Configuration schemes.** We now introduce the language of configuration schemes, which intuitively is a way of assigning each boundaried graph \((G, X)\) a set of configurations \(\text{conf}(G, X)\) which contains all the essential information about the behaviour of \((G, X)\) with respect to some computational problem.

A configuration scheme is a pair of mappings \((\mathfrak{C}, \text{conf})\) such that with any set of vertices \(X \subseteq \Omega\) we may associate a set of configurations \(\mathfrak{C}(X)\), and with every boundaried graph \((G, X)\) we may associate a subset of configurations \(\text{conf}(G, X) \subseteq \mathfrak{C}(X)\) realized by \((G, X)\). For an example what this might be, see the first paragraph of the proof of Lemma 14. We will need several properties of configuration schemes, the first of which is composability.

We will say that such a configuration scheme \((\mathfrak{C}, \text{conf})\) is composable if the following conditions hold.

- For a boundaried graph \((G, X)\) and \(x \in X\), \(\text{conf}(\text{forget}((G, X), x))\) is uniquely determined by \(\text{conf}(G, X)\) and \(x\).
- For boundaried graphs \((G_1, X_1)\) and \((G_2, X_2)\) whose union is defined, \(\text{conf}((G_1, X_1) \oplus (G_2, X_2))\) is uniquely determined by \(\text{conf}(G_1, X_1)\) and \(\text{conf}(G_2, X_2)\).

Letting \(\Xi := \bigcup_{X \subseteq \Omega} 2^{\mathfrak{C}(X)}\), the above conditions are equivalent to saying that there exist operators

\[
\text{forget}: \Xi \times \Omega \to \Xi \quad \text{and} \quad \oplus: \Xi \times \Xi \to \Xi
\]

such that the following assertions hold:

- Operator \(\text{forget}(\cdot, \cdot)\) is defined only on pairs of the form \((C, x)\) such that \(C \in 2^{\mathfrak{C}(X)}\) and \(x \in X\) for some \(X \subseteq \Omega\); then \(\text{forget}(C, x) \in 2^{\mathfrak{C}(X - \{x\})}\).
- For every boundaried graph \((G, X)\) and \(x \in X\), we have

\[
\text{conf}(\text{forget}((G, X), x)) = \text{forget}(\text{conf}(G, X), x).
\]

- If \(C_1 \in 2^{\mathfrak{C}(X_1)}\) and \(C_2 \in 2^{\mathfrak{C}(X_2)}\) for some \(X_1, X_2 \subseteq \Omega\), then \(C_1 \oplus C_2 \in 2^{\mathfrak{C}(X_1 \cup X_2)}\).
- For all boundaried graphs \((G_1, X_1)\) and \((G_2, X_2)\) whose union is defined, we have

\[
\text{conf}((G_1, X_1) \oplus (G_2, X_2)) = \text{conf}(G_1, X_1) \oplus \text{conf}(G_2, X_2).
\]

In other words, \(\text{conf}\) is a homomorphism from the algebra of boundaried graphs endowed with operations \(\text{forget}\) and \(\oplus\) to the algebra on \(\Xi\) with the same operations. Note that \(\oplus\) is commutative and associative on boundaried graphs, hence it should also have these properties on sets of configurations.

We will say that \((\mathfrak{C}, \text{conf})\) is efficiently composable if there is a non-decreasing computable function \(\zeta: \mathbb{N} \to \mathbb{N}\) such that

- For every \(X \subseteq \Omega\) we have \(|\mathfrak{C}(X)| \leq \zeta(|X|)\) and given \(X\), one can compute \(\mathfrak{C}(X)\) in time \(\zeta(|X|)^O(1)\).
- Given a boundaried graph \((G, X)\) with \(|V(G)| \leq 2\) and \(X = V(G)\), one can compute \(\text{conf}(G, X)\) in constant time.
- Given $X \subseteq \Omega$, $x \in X$, and $C \subseteq \mathcal{E}(X)$, one can compute $\text{forget}(C, x)$ in time $\zeta(|X|)^{O(1)}$.
- Given $X_1, X_2 \subseteq \Omega$, $C_1 \subseteq \mathcal{E}(X_1)$, and $C_2 \subseteq \mathcal{E}(X_2)$, one can compute $C_1 \oplus C_2$ in time $\zeta(|X|)^{O(1)}$.

Finally, we will say that $(\mathcal{E}, \text{conf})$ is idempotent if there is a non-decreasing computable function $\tau : \mathbb{N} \to \mathbb{N}$ such that the following condition holds:

- Consider any $X \subseteq \Omega$ and a multiset $\mathcal{M}$ whose elements are subsets of $\mathcal{E}(X)$. Suppose in $\mathcal{M}$ there is $C \subseteq \mathcal{E}(X)$ such that for each $c \in C$, there are at least $\tau(|X|)$ elements $D \in \mathcal{M} - \{C\}$ such that $c \in D$. Then

$$\bigoplus_{D \in \mathcal{M} - \{C\}} D = \bigoplus_{D \in \mathcal{M}} D.$$ 

A configuration scheme $(\mathcal{E}, \text{conf})$ that is both efficiently composable and idempotent shall be called tractable. The corresponding functions $\zeta$ and $\tau$ shall be called the witnesses of tractability of $(\mathcal{E}, \text{conf})$.

**Recognizing properties.** A graph property is simply a set $\Pi$ consisting of graphs, interpreted as graphs that admit $\Pi$. A graph property $\Pi$ is recognized by a configuration scheme $(\mathcal{E}, \text{conf})$ if there exists a subset of final configurations $F \subseteq \mathcal{E}(\emptyset)$ such that for every graph $G$, we have

$$G \in \Pi \quad \text{if and only if} \quad \text{conf}(G, \emptyset) \cap F \neq \emptyset.$$ 

As mentioned before, the formalism introduced above should be standard for a reader familiar with the work on graph algebras and recognition of MSO-definable languages of graphs. Recall here that MSO$_2$ is a logic on graphs that extends the standard first order logic FO by allowing quantification over subsets of vertices and over subsets of edges. A graph property $\Pi$ is MSO$_2$-definable if there exists a sentence $\varphi$ of MSO$_2$ such that for every graph $G$, $\varphi$ holds in $G$ if and only if $G \in \Pi$. Then the classic connection between graph algebras and MSO$_2$ yields the following statement. As this is not the main focus of our work, we only give a sketch of the proof, but both the statement and the tools are standard.

**Lemma 13.** Every MSO$_2$-definable graph property is tractable.

**Proof (Sketch).** Let us first recall some basic definitions and facts from logic. The quantifier rank of a formula is the maximum number of nested quantifiers in it. For $X \subseteq \Omega$, let MSO$_2[X]$ be the standard MSO$_2$ logic on graphs where in the signature, apart from the standard relations for encoding graphs, we also have each $x \in X$ introduced as a constant. It is known that for every $q \in \mathbb{N}$ and $X$, there is only a finite number of pairwise non-equivalent MSO$_2[X]$-statements of quantifier rank at most $q$. Moreover, given $q$ and $X$, one can compute a set $\text{St}^q(X)$ consisting of one MSO$_2[X]$-statement of quantifier rank at most $q$ per each class of equivalence. Thus, $|\text{St}^q(X)|$ is bounded by a computable function of $q$ and $X$. Then, with every boundaried graph $(G, X)$ we can associate its $q$-type $\text{tp}^q(G, X) \subseteq \text{St}^q(X)$, which just comprises all those statements $\psi \in \text{St}^q(X)$ that are satisfied in $(G, X)$.

We move to the sketch of the proof. Let $\Pi$ be the property in question, and let $\varphi$ be the MSO$_2$-statement defining $\Pi$. Let $q$ be the quantifier rank of $\varphi$. We define the following configuration scheme $(\mathcal{E}, \text{conf})$:

- for $X \subseteq \Omega$, we let $\mathcal{E}(X) := \text{St}^q(X)$; and
- for a boundaried graph $(G, X)$, we let $\text{conf}(G, X) := \text{tp}^q(G, X)$.

The fact that this configuration scheme is efficiently composable and idempotent is a standard fact about MSO$_2$ logic, which can be proved e.g. using Ehrenfeucht-Fraïssé games. By taking the set of final configurations to be $F := \{\varphi\}$, we see that this configuration scheme recognizes $\Pi$. 

Lemma 13 gives tractability of any MSO$_2$-definable graph property, however the witnesses $\zeta, \tau$ of this tractability are non-elementary functions. We now give an explicit configuration scheme for the property of our interest: containing a $k$-path.
Lemma 14. For \( k \in \mathbb{N} \), let \( \Pi_k \) be the graph property comprising all graphs that contain a simple path on \( k \) vertices. Then \( \Pi_k \) is recognized by a tractable configuration scheme with witnesses \( \zeta(x) = k \cdot 2^{x+1} \cdot x! \) and \( \tau(x) = x + k \).

Proof. We first define the configuration scheme. Let \( s, t \) be two new special vertices, not belonging to \( \Omega \); intuitively, in our scheme these will be markers representing the beginning and the end of the constructed path. Recall that a linear forest is an acyclic undirected graph whose every component is a path. For \( X \subseteq \Omega \), we define the set of configurations \( \mathcal{C}(X) \) as follows: \( \mathcal{C}(X) \) consists of all pairs \((H, i)\) such that:

- \( H \) is a linear forest with vertex set \( X \cup \{s, t\} \), where the degrees of \( s \) and \( t \) are at most 1; and
- \( i \in \{0, 1, \ldots, k - 1\} \).

It is straightforward to see that the number of different linear forests with vertex set of size \( p \) is at most \( 2^{p-1} \cdot p! \): a linear forest can be chosen by selecting a permutation \( \pi = (u_1, \ldots, u_p) \) of the vertex set \((p! \text{ choices}), and then deciding, for each \( i \in \{1, \ldots, p-1\} \), whether \( u_i \) and \( u_{i+1} \) are adjacent (\( 2^{p-1} \text{ choices})

Therefore, we have \( |\mathcal{C}(X)| \leq k \cdot 2^{|X|+1} \cdot |X|! = \zeta(|X|) \), as required.

Now, for a boundaried graph \((G, X)\) and a configuration \((H, i) \in \mathcal{C}(X)\), we shall say that \((H, i)\) is realized in \((G, X)\) if there exists a family of paths \( \{P_e : e \in E(H)\} \) in \( G \) satisfying the following conditions:

- For each \( xy \in E(H) \), \( P_e \) is a path whose one endpoint is \( x \), provided \( x \in X \), and the second endpoint is \( y \), provided \( y \in X \). If \( x \) or \( y \) (or both) belongs to \( \{s, t\} \), then the corresponding endpoint of \( P_e \) can be any vertex of \( V(G) \). Moreover, \( V(P_e) \cap X = \{x, y\} \cap X \).
- For all \( e, e' \in E(H) \), the paths \( P_e \) and \( P_{e'} \) are vertex-disjoint, apart from possibly sharing an endpoint in case \( e \) and \( e' \) share an endpoint.
- The total number of edges on paths \( P_e \) is equal to \( i \).

For a boundaried graph \((G, X)\), let \( \text{conf}(G, X) \subseteq \mathcal{C}(X) \) be the set of configurations realized in \((G, X)\).

Observe that a graph \( G \) contains a simple path on \( k \) vertices if and only if \( \text{conf}(G, \emptyset) \) contains configuration \((H_0, k - 1)\), where \( H_0 \) is the graph on vertex set \( \{s, t\} \) with the edge \( st \) present. Thus, by setting \( F := \{(H_0, k - 1)\} \) we see that the scheme \((\mathcal{C}, \text{conf})\) recognizes \( \Pi_k \). It remains to show that \((\mathcal{C}, \text{conf})\) is suitably efficiently composable and idempotent.

Claim 7. The configuration scheme \((\mathcal{C}, \text{conf})\) is efficiently composable with witness \( \zeta(x) = k \cdot 2^{x+1} \cdot x! \).

Proof. The bound \( |\mathcal{C}(X)| \leq \zeta(|X|) \) has already been shown. Also, for every boundaried graph \((G, X)\) such that \( |V(G)| \leq 2 \) and \( X = V(G) \), the value of \( \text{conf}(G, X) \) can be computed in constant time trivially by the definition. Thus, to prove the claim, it remains to define suitable operators \( \text{forget} \) and \( \oplus \), and to show that they are computable in time \( \zeta(|X|)O(1) \).

We start with defining the \( \text{forget}((\cdot, \cdot)) \) operator. For any boundaried graph \((G, X)\) and \( x \in X \) consider a configuration \((H, i) \in \text{conf}(\text{forget}((G, X), x)) \) and a corresponding family of paths \( \{P_e : e \in E(H)\} \) in \( G \), as described in the definition of \( \text{conf}((\cdot, \cdot)) \). There are three possibilities of how \( x \) can interact with this family of paths:

- \( x \) is an internal vertex of exactly one path \( P_e \). This means that \((H', i) \in \text{conf}(G, X)\), where \( e = yz \) and \( H' = (V(H) \cup \{x\}, E(H) \cup \{xy, xz\} - \{yz\}) \).
- \( x \) is an endpoint of some path \( P_e \). From the definition this is possible if and only if the corresponding endpoint of \( e \) belongs to \( \{s, t\} \); say \( e = ys \), the other case being analogous. Again, this means that \((H', i) \in \text{conf}(G, X)\), where \( H' = (V(H) \cup \{x\}, E(H) \cup \{xy, xs\} - \{ys\}) \).
- \( x \) does not belong to any of these paths. This means that \((V(H) \cup \{x\}, E(H)), i) \in \text{conf}(G, X)\).

These observations show a way of defining \( \text{forget}((\cdot, \cdot)) \). For a linear forest \( H \) and \( x \in V(H) \) such that \( x \) has degree 2 in \( H \), let \( \text{join}(H, x) \) be the graph obtained from \( H \) by adding an edge connecting the neighbors.
of $x$ and removing $x$ itself. Then for $C \subseteq \mathcal{C}(X)$ we can write the definition of forget as follows:

$$
\text{forget}(C, x) := \text{forget}_0(C, x) \cup \text{forget}_2(C, x);
$$

$$
\text{forget}_0(C, x) := \{ (H - x, i) : (H, i) \in C, x \in V(H), \deg_H(x) = 0 \};
$$

$$
\text{forget}_2(C, x) := \{ (\text{join}(H, x), i) : (H, i) \in C, x \in V(H), \deg_H(x) = 2 \}.
$$

Based on the previous observations, it is easy to see that this operator satisfies the desired property: $\text{conf}(\text{forget}((G, X), x)) = \text{conf}(\text{forget}(G, X), x)$ for every boundaried graph $(G, X)$ and $x \in X$. Moreover, forget$(C, x)$ can be computed in time $\zeta(|X|)^{O(1)}$ by inspecting every element of $C$ and applying the formula above; recall here that $|C| \leq \zeta(|X|)$.

We now move to the discussion of the $\oplus$ operator. Consider any pair of boundaried graphs $(G_1, X_1)$ and $(G_2, X_2)$ whose union is defined, and let $(G, X) := (G_1, X_1) \oplus (G_2, X_2)$. Consider a configuration $(H, i) \in \text{conf}(G, X)$ and a corresponding family of paths $\{P_e : e \in E(H)\}$ in $G$. Note that each path $P_e$ is either a single edge, in which case it can be present both in $G_1$ and $G_2$, or it has some internal vertices. In the latter case all internal vertices and edges of $P_e$ must belong either to $G_1$ or to $G_2$. They cannot belong to both, as $P_e$ is internally disjoint with $X$, and in $G$ there is no edge between $V(G_1) - X_1$ and $V(G_2) - X_2$. In particular, this means that each path from the family $\{P_e : e \in E(H)\}$ is either entirely contained in $G_1$ or entirely contained in $G_2$. Based on this observation, we can define $\oplus$ as follows.

Two configurations $(H_1, i_1) \in \mathcal{C}(X_1)$ and $(H_2, i_2) \in \mathcal{C}(X_2)$ are **mergable** if $E(H_1) \cap E(H_2) = \emptyset$, $i_1 + i_2 \leq k - 1$ and the graph $H_1 \oplus H_2 := (V(H_1) \cup V(H_2), E(H_1) \cup E(H_2))$ is a linear forest with $s$ and $t$ having degrees at most 1. Then for $C_1 \subseteq \mathcal{C}(X_1)$ and $C_2 \subseteq \mathcal{C}(X_2)$, we define operator $\oplus$ as follows

$$
C_1 \oplus C_2 := C_1 \cup C_2 \cup \{(H_1 \oplus H_2, i_1 + i_2) : (H_1, i_1) \in C_1, (H_2, i_2) \in C_2, \text{ and they are mergable} \}.
$$

Based on the previous observation, it is easy to see that this operator satisfies the desired property: $\text{conf}((G_1, X_1) \oplus (G_2, X_2)) = \text{conf}(G_1, X_1) \oplus \text{conf}(G_2, X_2)$ for all boundaried graphs $(G_1, X_1)$ and $(G_2, X_2)$ whose union is defined. That the operator $\oplus$ defined above is commutative and associative is obvious. Finally, the value of $C_1 \oplus C_2$ can be computed in time $\zeta(|X|)^{O(1)}$ by inspecting all pairs $((H_1, i_1), (H_2, i_2)) \in C_1 \times C_2$, whose number is bounded by $\zeta(|X_1|)\zeta(|X_2|) \leq \zeta(|X|)^2$, and applying the formula above.

**Claim 8.** The configuration scheme $(\mathcal{C}, \text{conf})$ is idempotent with witness $\tau(x) = x + k$.

**Proof.** Consider any $X \subseteq \Omega$ and a multiset $\mathcal{M}$ whose elements are subsets of $\mathcal{C}(X)$. Suppose that there is $C \in \mathcal{M}$ such that for each $c \in C$, there are at least $|X| + k$ elements $D \in \mathcal{M} - \{C\}$ such that $c \in D$. Let $S := \bigoplus(\mathcal{M} - \{C\})$. We need to prove that $S = S \oplus C$.

That $S \subseteq S \oplus C$ is directly implied by the definition of operator $\oplus$. To prove that $S \oplus C \subseteq S$, let us consider any configuration $(H, i) \in S \oplus C$. From the definition of $\oplus$ there exists a multiset of configurations $M = \{(H_1, i_1), (H_2, i_2), \ldots, (H_m, i_m)\}$ for some $m \leq |\mathcal{M}|$, with each configuration in $M$ chosen from some distinct element of multiset $\mathcal{M}$, such that $H = \bigoplus_{j=1}^m H_j$ and $i = \sum_{j=1}^m i_j$. In particular, graphs $H_j$ for $j \in \{1, \ldots, m\}$ are pairwise mergable.

Note that $H = \bigoplus_{j=1}^m H_j$ is a linear forest on a vertex set of size $|X| + 2$, hence it has at most $|X| + 1$ edges. As graphs $H_j$ have pairwise disjoint edge sets due to being mergable, while $\sum_{j=1}^m i_j = i \leq k - 1$, we conclude the following: if $|M| > |X| + k$, then there is a configuration $(H_j, i_j) \in M$ such that $E(H_j) = \emptyset$ and $i_j = 0$. Note that this configuration can be safely removed from $M$ without changing the union of configurations in $M$. By performing this operation repeatedly, from now on we may assume that $|M| \leq |X| + k$. 

22
Now, if none of the configurations from $M$ were selected from $C$, then $(H, i) \in S$ and we are done. On the other hand, if some configuration $(H_j, i_j)$ was selected from $C$, then by the assumptions on $C$, there is at least one other element $D \in M - \{C\}$, such that no other configuration in $M$ was selected from $D$ and $(H_j, i_j) \in D$. Therefore, we can select $(H_j, i_j)$ from $D$ instead of $C$, which again implies that $(H, i) \in S$. 

Claims 7 and 8 verify that the scheme $(C, \text{conf})$ has all the required properties. This concludes the proof of Lemma 14. 

Maintaining a scheme. We next show that whenever a graph property admits a tractable configuration scheme, it can be efficiently maintained in a dynamic graph of bounded treedepth.

**Lemma 15.** Let $\Pi$ be a graph property that is recognized by a tractable configuration scheme with polynomial-time computable witnesses $\zeta$ and $\tau$. Then there exists a data structure that maintains a dynamic graph $G$ of treedepth at most $d$ under edge insertions and deletions with update time $2^{O(d^2)} \cdot \tau(d)\zeta(d)O(1)$ while offering a query on whether $G \in \Pi$ in time $O(1)$. Upon inserting an edge that breaks the invariant $d \leq d$, the data structure detects this and refuses to carry out the operation. The initialization of the data structure for an edgeless $n$-vertex graph takes time $O(n)$.

**Proof.** We enrich the data structure $\mathbb{D}[F, G]$ presented in Section 5 to the data structure $\mathbb{D}^{\Pi}[F, G]$ implementing the requested functionality. Let $(C, \text{conf})$ be the assumed tractable configuration scheme recognizing $\Pi$. Denote $\zeta := \zeta(d)$ and $\tau := \tau(d)$; note that these can be computed in time polynomial in $d$.

For each $w \in V(G)$, define the graph

$$G_w := \left( \text{desc}_F(w) \cup \text{SReach}_{F,G}(w), \{ e \in E(G) : e \cap \text{desc}_F(w) \neq \emptyset \} \right).$$

In other words, the vertex set of $G_w$ consists of the descendants of $w$ plus $\text{SReach}_{F,G}(w)$, while in the edge set we include only those edges of $G$ that are incident on vertices of $\text{desc}_F(w)$. Thus, $\text{SReach}_{F,G}(w)$ is an independent set in $G_w$. Further, we define the boundaried graph

$$G_w := (G_w, \text{SReach}_{F,G}(w)).$$

In addition to all the data stored in $\mathbb{D}[F, G]$, in $\mathbb{D}^{\Pi}[F, G]$ we include the following. For each vertex $u$, subset $X \subseteq \text{SReach}_{F,G}(u) \cup \{u\}$, $i \in \{1, \ldots, d\}$, and configuration $c \in C(X)$, we store the mug $B[u, X, i, c]$ defined as follows:

$$B[u, X, i, c] := \{ w \in B[u, X, i] : c \in \text{conf}(G_w) \}.$$ 

In other words, the mug $B[u, X, i, c]$ comprises all vertices $w$ from the bucket $B[u, X, i]$ for which $c$ is realized in $G_w$. Note that the mugs are not necessarily disjoint, and some vertices may even not belong to any mug.

Similarly as for the buckets, the mugs are represented as doubly linked lists. In the lists corresponding to mugs we only store indices of the required vertices, representing their copies, while the actual objects corresponding to vertices are stored in the buckets as before. Each vertex object $w$ in bucket $B[u, X, i]$, besides storing pointers to the next and the previous element in $B[u, X, i]$, also stores $|C(X)| \leq \zeta$ additional pointers: for each $c \in C(X)$, we store the pointer to the position of an element in $B[u, X, i, c]$ containing index of $w$, or null pointer if the $w$ does not belong to $B[u, X, i, c]$. Thus, whenever we remove a vertex $w$ from its bucket, we may also remove it from all the mugs to which it belongs in time $O(\zeta)$.

It is again clear that $\mathbb{D}^{\Pi}[F, G]$ can be initialized for an edgeless graph $G$ in $O(n)$ time, because there are additionally $O(n)$ empty mugs to initialize. We now explain how the mugs are maintained upon updates of
$\mathbb{D}^{|F, G|}$. For this, we modify the reasoning from the proof of Lemma 12. Let us focus on edge insertion, as edge removal is again analogous. We construct the core $K$ and the forests $F^K$ and $F'$ exactly as before. We also apply all the operations needed for updating $\mathbb{D}^{|F, G|}$ to $\mathbb{D}^{|F', H|}$ in the same manner. It remains to update the mugs.

The first step in updating $\mathbb{D}^{|F, G|}$ to $\mathbb{D}^{|F', H|}$ was to remove each $u \in K$ from the bucket in which it resides. When performing this operation we also remove $u$ from all the mugs to which $u$ belongs. Note that this boils down to removing the single list element corresponding to $u$ not just from one list representing the bucket containing $u$, but also from at most $\zeta$ lists representing the mugs containing $u$.

The next step in updating $\mathbb{D}^{|F, G|}$ to $\mathbb{D}^{|F', H|}$ was to rename the buckets $B[u, \cdot, \cdot]$ for $u \in K$ so as to model re-attaching trees $F_u$ for $a \in \text{App}_F(K)$ in the construction of $F'$. We apply the same renaming scheme: whenever a bucket $B[u, X, i]$ is renamed to $B[u', X, i]$, this renaming applies also to all the mugs $B[u, X, \cdot, \cdot]$ that are sub-lists of $B[u, X, i]$. Recall that the key observation in the proof of Lemma 12 was that after the renaming, all the data stored for vertices $w \notin K$ did not require updating, and even the buckets $B[u, \cdot, \cdot]$ for $u \in K$ were as they should be in $\mathbb{D}^{|F', H|}$, except they were lacking vertices of $K$. Observe that now, the same also applies to the mugs: mugs $B[w, \cdot, \cdot, \cdot]$ for $w \notin K$ do not require updating, while mugs $B[u, \cdot, \cdot, \cdot]$ for $u \in K \cup \{\bot\}$ are as they should be in $\mathbb{D}^{|F', H|}$, except they lack vertices of $K$. This is because for $w \notin K$ we have $H_w = G_w$, where $H_w$ is the boundary graph defined analogously to $G_w$, but with respect to the graph $H$ and its elimination forest $F'$.

Hence, it remains to update the mugs $B[w, \cdot, \cdot, \cdot]$ for $w \in K \cup \{\bot\}$ by inserting the lacking vertices of $K$. Similarly as in the proof of Lemma 12, for this it suffices to add each $u \in K$ to the mug

$$B[\text{parent}_{F^K}(u), \text{SReach}_{F^K, H[K]}(u), \text{height}(F^K_u), c]$$

for each $c \in \text{conf}(H_u)$. (6)

Note that this can be easily done in time $O(\zeta)$ provided we have computed the set $\text{conf}(H_u)$.

We now present a procedure that updates the mugs $B[w, \cdot, \cdot, \cdot]$ for all $w \in K \cup \{\bot\}$ by processing vertices $u \in K$ in a bottom-up manner over the forest $F^K$. When processing $u$ we assume that all $v \in \text{children}_{F^K}(u)$ have already been inserted in appropriate mugs as prescribed in (6), that is, all the mugs $B[u, \cdot, \cdot, \cdot]$ are already correctly constructed. Based on this we compute $\text{conf}(H_u)$, so that $u$ itself can be inserted to appropriate mugs as prescribed in (6).

First, we construct a set $W$ of vertices with $W \subseteq \text{children}_{F}(u)$ as follows: for each $X \subseteq \text{SReach}(u) \cup \{u\}, i \in \{1, \ldots, d\}$, and $c \in \mathcal{C}(X)$, include in $W$ the first $\tau$ elements of the mug $B[u, X, i, c]$, or all of them in case their number is at most $\tau$. Note that this can be done in time $O(\tau)$ per considered mug, so in total time $O(2^d \cdot d\tau\zeta)$ per vertex $u$. Also, we have $|W| \leq 2^d \cdot d\tau\zeta$. Next, we construct the multiset

$$\mathcal{M} := \{\{\text{conf}(H_v) : v \in W\}\}.$$ 

This can be done in time $O(|W| \cdot \zeta)$ as follows: for each $v \in W$, we deduce $\text{conf}(H_v)$ by inspecting the list element of $v$ and checking in which mugs it is contained.

Let $I_{uu'}$ is the boundaried graph $((\{u, u'\}, \{uu'\}), \{u, u'\})$; that is, it consists only of two boundary vertices $u, u'$ and the edge $uu'$. We finally compute $\text{conf}(H_u)$ using the following formula:

$$\text{conf}(H_u) = \text{forget} \left( \bigoplus_{u' \in \text{NeiUp}(u)} \text{conf}(I_{uu'}) \oplus \bigoplus_{D \in \mathcal{M}} D, u \right).$$

(7)

Note that formula (7) can be computed in time

$$(d + |\mathcal{M}|) \cdot \zeta^{O(1)} = 2^d \cdot d\tau\zeta^{O(1)},$$

24
because it involves $O(d + |M|)$ operations $\oplus$ and forget, plus $O(d)$ operations $\text{conf}(\cdot, \cdot)$ applied to a two-vertex graph: each of these operations can be carried out in time $\zeta^{O(1)}$ due to the efficient composability of $(\mathcal{C}, \text{conf})$. Once $\text{conf}(H_u)$ is computed, the vertex $u$ can be added to appropriate mugs as described in (6) in time $O(\zeta)$. It remains to argue that formula (7) is correct. First, observe that

$$H_u = \text{forget}\left( \bigoplus_{u' \in \text{NeiUp}(u)} \text{conf}(I_{uu'}) \oplus \bigoplus_{v \in \text{children}_{\mathcal{P}}(u)} H_{v}, ~ u \right).$$

Hence, by the composability of $(\mathcal{C}, \text{conf})$ we have

$$\text{conf}(H_u) = \text{forget}\left( \bigoplus_{u' \in \text{NeiUp}(u)} \text{conf}(I_{uu'}) \oplus \bigoplus_{v \in \text{children}_{\mathcal{P}}(u)} \text{conf}(H_v), ~ u \right). \tag{8}$$

However, the construction of $W$ and the idempotence of $(\mathcal{C}, \text{conf})$ implies that

$$\bigoplus_{D \in M} D = \bigoplus_{v \in W} \text{conf}(H_v) = \bigoplus_{v \in \text{children}_{\mathcal{P}}(u)} \text{conf}(H_v). \tag{9}$$

Now (8) and (9) imply the correctness of (7).

This concludes the description of the update operation. To see that the time complexity of the update is as promised, note that the time spent on processing a single $u \in K$ as above is bounded by $2^d \cdot d \tau \zeta^{O(1)}$. Since $|K| \leq 2^{O(d^2)}$, the total time complexity of $2^{O(d^2)} \cdot \tau \zeta^{O(1)}$ follows.

We are left with implementing the query whether the currently stored graph belongs to $\Pi$. In the data structure we will always store a single bit indicating this, so the query can be answered in $O(1)$ time, but the bit has to be updated upon every insertion or deletion. We now explain how to do this. In the previous paragraphs we described the computation of $\text{conf}(H_u)$ for any vertex $u$ based on the access to correctly updated mugs $B[u, \cdot, \cdot, \cdot]$, in time $2^d \cdot d \tau \zeta^{O(1)}$. We can apply the same reasoning for $u = \bot$, and thus compute within the same running time the value $\text{conf}(H_\bot) = \text{conf}(H, \emptyset)$. Now to verify whether $H \in \Pi$, it suffices to check whether one of the (constant number of) final configurations of $(\mathcal{C}, \text{conf})$ is contained $\text{conf}(H, \emptyset)$.

By combining Lemmas 14 and 15 we immediately obtain the following.

**Lemma 16.** There exists a data structure that maintains a dynamic graph $G$ of treedepth smaller than $k$ under edge insertions and deletions with update time $2^{O(k^2)}$ while offering a query on whether $G$ contains a simple path on $k$ vertices in time $O(1)$. Upon inserting an edge that breaks the invariant $\text{td}(G) < k$, the data structure detects this and refuses to carry out the operation. The initialization of the data structure for an edgeless $n$-vertex graph takes $O(n)$ time.

Finally, let us remark that by combining Lemma 15 with Lemma 13 in the same manner, we may recover the result of Dvořák et al. [12]: for every fixed MSO$_2$-definable property $\Pi$, there is a data structure that for a dynamic graph of treedepth at most $d$ maintains the information about $\Pi$-membership with update time $f(d)$, for some function $f$. 

25
7 Postponing insertions and the main result

In this section we combine the results from the previous section, in particular Lemma 16, with the technique of postponing insertions in order to prove our main result: a data structure for the dynamic k-path problem.

For this technique to work, we need to assume access to a dictionary on the edge set of the graph. This is in principle not an unusual assumption when dealing with dynamic graphs. In our case, however, it becomes an issue, as we would like to avoid factors dependent on |V| or |E| in our running times. Nevertheless, as this matter is not the focus of this paper, we briefly describe solutions available in the literature.

A dictionary is a data structure \( \mathbb{L} \) is built on top of some universe of keys \( U \). It stores a dynamically changing set of keys together with records (of constant size) associated with them. To be more precise, \( \mathbb{L} \) offers operations \( \text{insert}(e, r), \text{remove}(e) \) and \( \text{lookup}(e) \), which allow inserting a new key \( e \) with its record \( r \), removing key \( e \) and its record, and looking up the record of \( e \). The performance of \( \mathbb{L} \) depends on two parameters: the size of universe \( |U| \) and the number \( |L| \) of keys currently stored. Note that for our application to the \( k \)-path problem we have \( U = \binom{V}{2} \) where \( V \) is the invariant vertex set. That is, \( |U| = \mathcal{O}(|V|^2) \), while \( |L| = |E| \) is the number of edges in the current graph.

The literature offers, among other, the following implementations that can be applied to our setting:

- adjacency matrix: \( \mathcal{O}(|V|^2) \) space, \( \mathcal{O}(1) \) worst case time per operation;
- perfect hashing (FKS-hashing) \[27\]: \( \mathcal{O}(M) \) space, \( \mathcal{O}(1) \) expected worst case time per operation, where \( M \) is an upper bound on the number of distinct edges that may appear, known a priori;
- dynamic perfect hashing \[11\]: \( \mathcal{O}(|E|) \) space, \( \mathcal{O}(1) \) expected amortized time per operation;
- Y-fast tries \[27\]: \( \mathcal{O}(|E|) \) space, \( \mathcal{O}(\log \log |V|) \) amortized time per operation.

In the sequel, we do not fix any of these possibilities. Instead, we assume that we are given access to a dictionary data structure \( \mathbb{L} \), and we measure the time complexity both in terms of the standard operations and in terms of the number of calls to \( \mathbb{L} \).

Postponing insertions. We now present a generic technique for turning data structures working under structural restrictions into general ones, at the cost of introducing amortization and using a dictionary. As we mentioned, the idea is not new: it was used by Eppstein et al. \[15\] for planarity testing. We formulate the technique in an abstract way, because apart from being useful in our setting, it also applies to some other problems discussed by Alman et al. \[1\].

Suppose \( U \) is some universe. A family of subsets \( \mathcal{F} \subseteq 2^U \) is downward closed if for all \( E \subseteq F \subseteq U \), \( F \in \mathcal{F} \) entails \( E \in \mathcal{F} \). A data structure \( \mathbb{D} \) strongly supports \( \mathcal{F} \) membership if \( \mathbb{D} \) maintains a subset \( X \) of \( U \) under \( \text{insert}(x) \) and \( \text{remove}(x) \) operations (just as in the dictionary, but without the associated records), and in addition to this it offers a boolean query \( \text{member}(\cdot) \) that checks whether \( X \in \mathcal{F} \). We also consider the following weaker notion: a data structure \( \mathbb{D} \) weakly supports \( \mathcal{F} \) membership if, again, it implements \( \text{insert}(x) \) and \( \text{remove}(x) \) operations on a dynamic subset \( X \subseteq U \), but works under the restriction that we always have \( X \in \mathcal{F} \). Whenever performing an \( \text{insert}(\cdot) \) operation would violate the invariant \( X \in \mathcal{F} \), the data structure should detect this and refuse to carry out the operation.

The following lemma shows that data structures supporting weak \( \mathcal{F} \) membership can be turned into ones supporting strong \( \mathcal{F} \) membership at the cost of introducing amortization and allowing access to a dictionary \( \mathbb{L} \) on \( U \). The proof essentially repeats the same argument as \[15, Corollary 1\].

**Lemma 17.** Suppose \( U \) is a universe and we have access to a dictionary \( \mathbb{L} \) on \( U \). Let \( \mathcal{F} \subseteq 2^U \) be downward closed and suppose there is a data structure \( \mathbb{D} \) that weakly supports \( \mathcal{F} \) membership. Then there is a data structure \( \mathbb{D}' \) that strongly supports \( \mathcal{F} \) membership, where each \( \text{member}(\cdot) \) query takes \( \mathcal{O}(1) \) time and each update uses amortized \( \mathcal{O}(1) \) time and amortized \( \mathcal{O}(1) \) calls to operations on \( \mathbb{L} \) and \( \mathbb{D} \).
Proof. \( \mathcal{D}' \) maintains (a copy of) \( \mathcal{D} \) and an additional queue \( Q \), in which \( \mathcal{D}' \) stores elements whose insertion is postponed. We will denote the set stored in \( \mathcal{D}' \) by \( X \), while the sets stored in (the copy of) \( \mathcal{D} \) and in \( Q \) are \( X_\mathcal{D} \) and \( X_Q \), respectively. We maintain the invariant that \( X \) is the disjoint union of \( X_\mathcal{D} \) and \( X_Q \).

The queue \( Q \) is implemented as a doubly linked list. In addition to the above, we maintain a dictionary \( L \) that stores \( X \). In \( L \), the record associated with each \( x \in X \) is either a pointer that points to the list element representing \( x \) in \( Q \), or a marker \( \perp \) in case \( x \in X_\mathcal{D} \). Thus, given \( x \in U \), we may use the \( \text{lookup}(x) \) operation in \( L \) to check whether \( x \) belongs to \( X_\mathcal{D} \) or \( X_Q \) and, if the latter holds, access the corresponding list element on \( Q \). In the following, whenever we insert an element to \( \mathcal{D} \) or \( Q \), we insert it to \( L \) as well (possibly together with a pointer to list element). Same for removals.

We maintain the following invariant:

\((\ast)\) \hspace{1cm} If \( Q \) is non-empty and \( x \) is the front element of \( Q \), then \( X_\mathcal{D} \cup \{x\} \notin F \).

Note that since \( \mathcal{D} \) stores \( X_\mathcal{D} \), we obviously have \( X_\mathcal{D} \in F \). Hence if \( Q \) is empty, then \( X_\mathcal{D} = X \notin F \). On the other hand, if \( Q \) is non-empty, then invariant \((\ast)\) together with upward-closedness of \( F \) implies \( X \notin F \). Thus, the \( \text{member}() \) query amounts to checking whether \( Q \) is empty, which can be done in constant time.

We now explain how updating \( \mathcal{D}' \) works, starting with the \( \text{insert}(x) \) operation on \( \mathcal{D}' \). By applying \( \text{lookup}(x) \) in \( L \), we may further assume that \( x \notin X \). If \( Q \) is not empty, then we push \( x \) to the back of \( Q \). Otherwise, we try to add \( x \) to \( X_\mathcal{D} \) by applying \( \text{insert}(x) \) on \( \mathcal{D} \). If this operation succeeds (i.e. \( X_\mathcal{D} \cup \{x\} \in F \)), then there is nothing more to do. Otherwise, if \( \mathcal{D} \) refuses to insert the element \( x \), then we have a situation where \( X_Q = \emptyset \), \( X_\mathcal{D} \in F \), but \( X_\mathcal{D} \cup \{x\} \notin F \). Hence, we push \( x \) to the back of \( Q \); note that invariant \((\ast)\) is thus satisfied, as \( x \) becomes the only element in \( Q \). This concludes the implementation of \( \text{insert}(x) \) in \( \mathcal{D}' \).

We now move to the \( \text{remove}(x) \) operation in \( \mathcal{D}' \). First, we apply \( \text{lookup}(x) \) operation offered by \( L \). If \( x \notin X \), then there is nothing to do. Otherwise, we have two cases: either \( x \in X_\mathcal{D} \) or \( x \in X_Q \).

If \( x \in X_Q \), then we remove \( x \) from \( Q \); recall here that \( \text{lookup}(x) \) provided us with the pointer to the corresponding list element, so this can be done in constant time. However, at this point invariant \((\ast)\) might have ceased to hold. Hence, we apply the \( \text{flush}() \) operation, defined as follows. We iteratively take the front element \( x \) from \( Q \) and try to insert it to \( \mathcal{D} \) by applying \( \text{insert}(x) \) on \( \mathcal{D} \). If \( x \) gets successfully inserted into \( \mathcal{D} \), we remove \( x \) from \( Q \) and proceed with the iteration. Otherwise, if \( \mathcal{D} \) refuses to insert \( x \), we break the iteration. Thus, the iteration stops when either \( Q \) becomes empty, or the first element \( x \) of \( Q \) satisfies \( X_\mathcal{D} \cup \{x\} \notin F \); so invariant \((\ast)\) is restored.

We are left with the case when \( x \in X_\mathcal{D} \). In this case we remove \( x \) from \( X_\mathcal{D} \) by calling \( \text{remove}(x) \) on \( \mathcal{D} \). Again, as this might have broken invariant \((\ast)\); we restore it by calling the \( \text{flush}() \) operation. This concludes the implementation of \( \text{remove}(x) \) in \( \mathcal{D}' \).

We are left with discussing the complexity. Observe that each operation \( \text{insert}(\cdot) \) uses \( O(1) \) operations on \( \mathcal{D} \) and \( O(1) \) operations on \( L \). Similarly for \( \text{remove}(\cdot) \), except that the application of \( \text{flush}(\cdot) \) may perform an unbounded number of moves of elements from \( X_Q \) to \( X_\mathcal{D} \), each involving \( O(1) \) operations on \( \mathcal{D} \) and \( L \). However, each element inserted to \( \mathcal{D}' \) is moved from \( X_Q \) to \( X_\mathcal{D} \) by \( \text{flush}(\cdot) \) at most once, so the operations used for successful moves from \( X_Q \) to \( X_\mathcal{D} \) can be charged to \( \text{insert}(\cdot) \) previously applied on \( \mathcal{D}' \). It follows that in the amortized sense, each operation in \( \mathcal{D}' \) uses \( O(1) \) time and \( O(1) \) operations on \( \mathcal{D} \) and \( L \).

We remark that Lemma 17 can be applied to two problems considered by Alman et al. [1]: \textsc{Edge Clique Cover} and \textsc{Point Line Cover}. In the first problem, given a graph \( G \) and parameter \( k \), we ask whether the edges of the \( G \) can be covered by at most \( k \) cliques in \( G \). In the second problem, given a set \( S \) of points in the plane and parameter \( k \), we ask whether all these points can be covered using at most \( k \) lines. Alman et al. [1] gave data structures for the dynamic variants of these problems, however working under the promise
that there is always a solution of size at most \( g(k) \). They achieved: update time \( \mathcal{O}(4^{g(k)}) \) and query time \( 2^{O(k')} + \mathcal{O}(16g(k)) \) for Edge Clique Cover; and update time \( \mathcal{O}(g(k)^3) \) and query time \( \mathcal{O}(g(k)g(k)+2) \) for Point Line Cover. It was left open whether the assumption about the promise can be lifted. By combining the data structures of Alman et al. [1] for \( g(k) = k + 1 \) with Lemma 17, we obtain data structures that achieve: amortized update time \( \mathcal{O}(4^k) \) and query time \( 2^{2O(k)} \) for Edge Clique Cover; and amortized update time \( \mathcal{O}(k^3) \) and query time \( 2^{O(k \log k)} \) for Point Line Cover. This assumes access to a dictionary on the edges of the graph in the case of Edge Clique Cover, and on the point set in the case of Point Line Cover.

**Data structure for dynamic \( k \)-path.** We now gather all our tools to prove the main result of this work.

**Theorem 4 (Main result, stated formally).** Suppose \( G \) is a dynamic graph with vertex set \( V \) updated by edge insertions and removals, and we have access to a dictionary \( L \) on the universe \( \binom{V}{2} \). Then there exists a dynamic data structure that under such updates maintains whether \( G \) contains a simple path on \( k \) vertices. Each update takes amortized time \( 2^{O(k^2)} \) and uses amortized \( O(1) \) calls to operations in \( L \), while the initialization for an edgeless \( G \) takes \( O(|V|) \) time.

**Proof.** Letting \( U \coloneqq \binom{V}{2} \), we note that a dynamic graph with vertex set \( V \) can be equivalently treated as a dynamic subset of \( U \). Let then \( Q_k \subseteq 2^U \) comprise all sets \( F \subseteq U \) such that the graph \( (V, F) \) does not contain a simple path on \( k \) vertices. Similarly, let \( T_k \subseteq 2^U \) comprise all sets \( F \subseteq U \) such that the graph \( (V, F) \) has treedepth smaller than \( k \). Note that both \( Q_k \) and \( T_k \) are downward closed and, by Lemma 3, we have \( Q_k \subseteq T_k \).

By Lemma 16, there is a data structure \( D \) that weakly supports \( T_k \) membership with update time \( 2^{O(k^2)} \), and moreover offers \( Q_k \) membership queries in constant time. By Lemma 17, we can now turn \( D \) into a data structure \( D' \) that strongly supports \( T_k \) membership, where each update takes amortized time \( 2^{O(k^2)} \) and uses \( O(1) \) operations on \( L \). Now, we can easily implement \( Q_k \) membership queries in \( D' \) as follows: if the currently maintained set \( X \subseteq U \) does not belong to \( T_k \), then it also does not belong to \( Q_k \), and otherwise we may simply query the data structure \( D \) that is maintained within \( D' \) (see the proof of Lemma 17). Finally, observe that strongly supporting \( Q_k \) membership is equivalent to the requirement requested in the theorem statement.

**8 Conclusions**

In this work we presented a data structure for the dynamic \( k \)-path problem that achieves amortized update time \( 2^{O(k^2)} \). We remark that it is straightforward to modify the data structure so that within the same complexity it also maintains an example \( k \)-path, in case there is any. Also, it is easy to verify that the space usage of the data structure is \( 2^{O(k^2)} \cdot n + \mathcal{O}(m) \).

Observe that while static fixed-parameter algorithms for \( k \)-path achieve a parametric factor of the running time of the form \( 2^{O(k)} \), this is the case neither for our data structure (factor \( 2^{O(k^2)} \)) nor for the data structure of Alman et al. [1] (factor \( k^{O(k)} \)). It would be interesting to investigate whether there is a data structure for the dynamic \( k \)-path problem that achieves (amortized) update time \( 2^{O(k)} \cdot \log^c n \), for some constant \( c \).

Our data structure for maintaining a recursively optimal elimination forest of a graph of treedepth \( d \) achieves update time \( 2^{O(d^2)} \). It is natural to ask whether this could be improved. The only bottleneck here is the use of the static algorithm of Reidl et al. [25] whose running time has parametric factor \( 2^{O(d^2)} \); all the other parts of the algorithm run in time \( O(d^{O(d)}) \). Note that a dynamic data structure for maintaining a
recursively optimal elimination forest of depth $d$ can be used for solving the static problem of determining whether the treedepth of a given graph is at most $d$: just insert the edges to the data structure one by one. Thus, our approach shows that improving the $2^{O(d^2)}$ parametric complexity in the static and in the dynamic setting is actually equivalent. However, note that for the purpose of applications, for instance to the $k$-path problem, maintaining an elimination forest of approximately optimum depth would be sufficient. Approximation algorithms for treedepth remain largely unexplored even in the static setting, which brings us to an old question [10]: is there a constant-factor approximation algorithm for treedepth with running time $2^{O(d)} \cdot n^{O(1)}$, where $d$ is the value of the treedepth?

Our data structure has an interesting application to dynamic connectivity problem on graphs of bounded treedepth. To be more precise, for any two vertices in the dynamic graph, the structure can answer if the vertices are currently connected (i.e., are in the same connected component). Indeed, if $F$ is a recursively optimal elimination forest of $G$, then to check whether $u$ and $v$ are connected in $G$ it suffices to check whether they are in the same connected component of $F$. This can be done by following the parent pointers from $u$ to $v$ to the roots of respective components of $F$, and checking that these roots are equal. Hence, the updates take $2^{O(d^2)}$ worst case time, while the queries take $O(d)$ worst case time. We find that this an interesting side remark, as there is a lower bound of $\Omega(\log n)$ for dynamic connectivity even if the dynamic graph remains a collection of paths at all times [24]; note here that a collection of paths may have arbitrarily large 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. [1] and by Dvořák et al. [12].

References

[1] 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.

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

[3] 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.

[4] A. Björklund. Determinant sums for undirected hamiltonicity. SIAM J. Comput., 43(1):280–299, 2014.

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

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

[7] H. L. Bodlaender, F. V. Fomin, D. Lokshtanov, E. Penninkx, S. Saurabh, and D. M. Thilikos. (Meta) Kernelization. J. ACM, 63(5):44:1–44:69, 2016.

[8] B. Courcelle and J. Engelfriet. Graph Structure and Monadic Second-Order Logic — A Language-Theoretic Approach, volume 138 of Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2012.
[9] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer, 2015.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984.

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

[18] M. Grohe, S. Kreutzer, R. Rabinovich, S. Siebertz, and K. Stavropoulos. Coloring and covering nowhere dense graphs. *SIAM Journal on Discrete Mathematics*, 32(4):2467–2481, 2018.

[19] 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.

[20] H. A. Kierstead and D. Yang. Orderings on graphs and game coloring number. *Order*, 20(3):255–264, 2003.

[21] 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.

[22] 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.

[23] J. Nešetřil and P. Ossona de Mendez. *Sparsity — Graphs, Structures, and Algorithms*, volume 28 of *Algorithms and combinatorics*. Springer, 2012.
[24] M. Patraşcu and E. D. Demaine. Lower bounds for dynamic connectivity. In Proceedings of the 36th Annual ACM Symposium on Theory of Computing, STOC 2004, pages 546–553. ACM, 2004.

[25] 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.

[26] 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.

[27] D. E. Willard. Log-logarithmic worst-case range queries are possible in space Θ(N). Information Processing Letters, 17(2):81–84, 1983.

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

[29] X. Zhu. Colouring graphs with bounded generalized colouring number. Discret. Math., 309(18):5562–5568, 2009.

A Pseudocodes

Algorithm 1: method core($L, q$)

Input: A subset of vertices $L$ and a positive integer $q$

Output: A $q$-core $K$ of $(G, F)$ such that $L \subseteq K$

1. $\hat{L} \leftarrow \text{anc}_F(L)$

2. return $\text{recCore}(\hat{L}, q, \bot)$
Algorithm 2: method recCore($\hat{L}, q, u$)

**Input**: A prefix $\hat{L}$ of $F$, a positive integer $q$, and a vertex $u \in V(G) \cup \{\bot\}$

**Output**: A $q$-core $K_u$ of $(G[\text{desc}_F(u)], F_u)$ such that $\hat{L} \cap \text{desc}_F(u) \subseteq K_u$

1. $R \leftarrow \text{new List}()$
2. \text{foreach } $w \in \hat{L}$ \text{ do}
3. \hspace{1em} \text{if } *(\text{toParent}(w)) = u \text{ then}
4. \hspace{2em} $R.\text{append}(w)$
5. \text{foreach } $X \subseteq \text{SReach}_F(u) \cup \{u\}$ such that $|X| \leq 2$ \text{ do}
6. \hspace{1em} $c \leftarrow q$
7. \hspace{2em} \text{for } $i \leftarrow d \text{ downto } 1$ \text{ do}
8. \hspace{3em} \text{foreach } $Y \subseteq \text{SReach}_F(u) \cup \{u\}$ such that $Y \supseteq X$ \text{ do}
9. \hspace{4em} $w \in B[u, Y, i]$ \text{ do}
10. \hspace{5em} \text{if } $w \notin R$ \text{ then}
11. \hspace{6em} $R.\text{append}(w)$
12. \hspace{5em} $c \leftarrow c - 1$
13. \hspace{6em} \text{if } $c = 0$ \text{ then}
14. \hspace{7em} \text{goto exit}
15. \hspace{3em} \text{exit:}
16. $K_u \leftarrow \text{new List}()$
17. \text{foreach } $w \in R$ \text{ do}
18. \hspace{1em} $K_u.\text{append}(\text{recCore}(\hat{L}, q, w))$
19. \text{if } $u \neq \bot$ \text{ then}
20. \hspace{1em} $K_u.\text{append}(u)$
21. \text{return } $K_u$
Algorithm 3: method \texttt{insert}(uv)

\begin{algorithm}
\textbf{Input} : A new edge uv \notin E(G)

\textbf{Result} : Structure $D[G,F]$ is updated to $D[G + uv, F']$, where $F'$ is a recursively optimal elimination forest of $G + uv$ of depth at most $d$

\begin{enumerate}
\item $K \leftarrow \text{core}(\{u,v\}, d + 1)$
\item construct $G[K] + uv$
\item $F^K \leftarrow$ recursively optimal elimination forest of $G[K] + uv$ obtained using Lemma 2
\item \textbf{if} height($F^K$) $> d$ \textbf{then}
\item \hspace{1em} raise exception: td($G + uv$) $> d$
\item \textbf{foreach} $v \in K$ \textbf{do}
\item \hspace{1em} remove $v$ from the bucket that it belongs to
\item \textbf{foreach} $u \in K \cup \{\bot\}$ and $X \subseteq \text{SReach}_{F^K}(u) \cup \{u\}$ and $i \in \{1, \ldots, d\}$ \textbf{do}
\item \hspace{1em} $B'[u,X,i] \leftarrow \text{new List}()$
\item \hspace{1em} $p'[u,X,i] \leftarrow \text{new memory cell containing } u$
\item \textbf{foreach} $u \in K \cup \{\bot\}$ and $X \subseteq \text{SReach}_{F^K}(u) \cup \{u\}$ and $i \in \{1, \ldots, d\}$ \textbf{do}
\item \hspace{1em} if $B[u,X,i]$ is empty then
\item \hspace{2em} continue
\item \hspace{1em} $u' \leftarrow$ deepest vertex of $X$ in $F^K$, or $\bot$ if $X = \emptyset$
\item \hspace{1em} $p[u,X,i] \leftarrow u'$
\item \hspace{1em} $B'[u',X,i] \leftarrow B[u,X,i]$
\item \hspace{1em} $p'[u',X,i] \leftarrow p[u,X,i]$
\item \textbf{foreach} $u \in K \cup \{\bot\}$ and $X \subseteq \text{SReach}_{F^K}(u) \cup \{u\}$ and $i \in \{1, \ldots, d\}$ \textbf{do}
\item \hspace{1em} $B[u,X,i] \leftarrow B'[u,X,i]$
\item \hspace{1em} $p[u,X,i] \leftarrow p'[u,X,i]$
\item \textbf{foreach} $u \in K$ \textbf{do}
\item \hspace{1em} $\text{SReach}(u) \leftarrow \text{SReach}_{F^K,G[K]+uv}(u)$
\item \hspace{1em} $\text{NeiUp}(u) \leftarrow \text{NeiUp}_{F^K,G[K]+uv}(u)$
\item \hspace{1em} $\text{height}(u) \leftarrow \text{height}(F^K_u)$
\item \hspace{1em} toParent($u$) \leftarrow pointer to $p[\text{parent}_{F^K}(u), \text{SReach}_{F^K}(u), \text{height}(F^K_u)]$
\item \hspace{1em} $B[\text{parent}_{F^K}(u), \text{SReach}_{F^K}(u), \text{height}(F^K_u)]$.append($u$)
\end{enumerate}
\end{algorithm}