Dynamic Detours

Daniel Dadush
Centrum Wiskunde & Informatica, Amsterdam, The Netherlands

Michał Pilipczuk
Institute of Informatics, University of Warsaw, Poland

Amadeus Reinald
Institute of Informatics, University of Warsaw, Poland

Marek Sokołowski
Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany

Michał Włodarczyk
Institute of Informatics, University of Warsaw, Poland

Abstract

Fix a parameter $k \in \mathbb{N}$. We give dynamic data structures that for a fully dynamic undirected graph G, updated over time by edge insertions and edge deletions, can answer the following queries:

- Long (u, v)-path: Given $u, v \in V(G)$, is there a path from u to v of length at least k?
- Long (u, v)-detour: Given $u, v \in V(G)$, is there a path from u to v of length at least $\mathsf{dist}_G(u, v)+k$?
- Even/odd (u, v)-path: Given $u, v \in V(G)$, is there a path from u to v of even/odd length?

The amortized time of executing an update or answering a query is $2^{\mathcal{O}(k^3)} \log n + \mathcal{O}(\log^2 n \log^2 \log n)$ in the first two cases, and $\mathcal{O}(\log^2 n \log^2 \log n)$ in the last, where n is the number of vertices of G.

The first result is in sharp contrast with known conditional lower bounds for reporting paths of length at most k. Specifically, there is no data structure supporting queries about (u, v)-paths of length at most two in time $n^{o(1)}$ unless the Triangle Conjecture fails.

Our main technical contribution is a mechanism of "delayed edge insertion" that works locally on the level of biconnected components.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms; Theory of computation → Fixed parameter tractability; Theory of computation → Data structures design and analysis

Keywords and phrases Dynamic algorithms, Fixed-parameter tractability

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.35

Related Version *Preprint*: `https://arxiv.org/abs/2605.03225` [13]

Funding The work of MiP on this manuscript is a part of project BOBR that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 948057). AR was supported by Polish National Science Centre SONATA BIS-12 grant number 2022/46/E/ST6/00143. MW was supported by Polish National Science Centre SONATA-19 grant number 2023/51/D/ST6/00155.

Acknowledgements This research has been initiated/conducted at the AlgUW workshop (Bedlewo 09.2025), supported by the Excellence Initiative – Research University (IDUB) funds of the University of Warsaw.

1 Introduction

The area of *parameterized dynamic data structures* aims to apply the principles of parameterized complexity to the setting of data structures for dynamically changing inputs. That is, instead of designing a static algorithm for a parameterized problem that just reads the input and computes the answer, the goal is to propose a data structure that efficiently

maintains the answer to the problem under updates to the maintained instance. Following the parameterized paradigm, the complexity guarantees for updates and queries to the data structure can now be measured both in terms of the considered parameters – where we typically allow superpolynomial dependence – and in terms of the total instance size – where we wish to keep the dependence sublinear, or even (poly)logarithmic or constant.

It is natural to ask which classic results and techniques of parameterized complexity can be lifted to the setting of dynamic data structures. So far, a number of positive results have been found, including data structures for detection and counting of small subgraph patterns [3, 11, 16], parameterized graph modification problems [26, 36], kernelization [5, 7, 26], problems in topologically-constrained and geometric graphs [4, 32], parameterized string problems [37], or even problems related to automata [19]. Of particular importance is the recent direction of dynamic maintenance of graph decompositions related to classic width parameters, such as treedepth [11, 15], feedback vertex number [35], treewidth [29, 30], or cliquewidth [33]. Indeed, such data structures typically also allow maintaining suitable dynamic programming tables for any problem that can be solved using the relevant decomposition by means of a bottom-up dynamic programming. Thus, dynamic data structures for maintaining graph decompositions typically lead to results that apply not to single problems, but to whole classes of problems. This is probably best exemplified by the dynamic counterparts of Courcelle's Theorem for treewidth [29, 30] and for cliquewidth [33].

In this work, we add another point to this growing list of results: we give parameterized dynamic data structures for finding long paths with prescribed endpoints. Our first result is the following.

▶ **Theorem 1.** *For every $k \in \mathbb{N}$ there exists a data structure that for a fully dynamic n-vertex graph G, supports the following operations in amortized $2^{\mathcal{O}(k^3)} \log n + \mathcal{O}(\log^2 n \log^2 \log n)$ time.*

- *Insert(u, v): insert an edge uv, provided it does not exist in G;*
- *Delete(u, v): delete an edge uv, provided it exists in G;*
- *LongPath(u, v): decide whether there is a path of length at least k between u and v.*

The data structure can be initialized for an edgeless G in time $2^{\mathcal{O}(k^3)} \cdot n$.

This should be compared to the task of reporting (u, v)-paths of length *at most* k. An easy reduction shows that even a static data structure reporting if $\mathsf{dist}_G(u, v) \leqslant k$ in time $g(k, n)$ can be used to detect a $(k+1)$-cycle in time $m \cdot g(k, n) \cdot 2^{\mathcal{O}(k \log k)} \log n$, where m denotes the number of edges [3, 20]. Therefore, such a data structure with query time $f(k) \cdot n^{o(1)}$ would break the Triangle Conjecture, which asserts that triangle detection requires time $m^{1+\Omega(1)}$. What is more, it would also break the 3SUM Conjecture [3]. Next, the k-Cycle Hypothesis states that for every $\varepsilon > 0$ there exists k so that detecting a cycle of length k requires time $\Omega(m^{2-\varepsilon})$ [1, 20]. Under this assumption, reporting if $\mathsf{dist}_G(u, v) \leqslant k$ cannot be even done in time $\mathcal{O}(m^{1-\varepsilon})$ per query, for any universal constant $\varepsilon > 0$.

We remark that the $\log n$ dependence on the number of vertices in the runtime is necessary: for $k = 1$, the data structure solves precisely the problem of dynamic connectivity, for which a classic $\Omega(\log n)$ lower bound was given by Pătraşcu and Demaine [38].

Finding long paths was among the first parameterized problems considered from the point of view of dynamic data structures. Alman, Mnich, and Vassilevska-Williams [3] used a dynamic variant of the color-coding technique to give a fully dynamic data structure with update time $2^{\mathcal{O}(k \log k)} \cdot \log n$ that can report whether the maintained graph contains a path of length k. This data structure can be easily adjusted to report a (u, v)-path of length *at least* k or conclude there is no (u, v)-path of length *exactly* k. However, if the shortest (u, v)-path has length $k - 1$ and the second shortest one has length $\Omega(n)$, color-coding alone seems not sufficient to obtain the guarantees of Theorem 1.

Later, Chen et al. [11] gave a dynamic data structure for detecting a path of length k with amortized update time $2^{\mathcal{O}(k^2)}$; thus, independent of n. Their approach is based on a dynamic data structure for maintaining an optimum treedepth decomposition of the graph, and crucially relies on the fact that any graph of large treedepth must contain a long path. Therefore, the approach inherently cannot handle queries about paths with prescribed roots.

Our technique allows us to not only detect (u, v)-paths of length at least k, but also (u, v)-paths that are exceed the distance between u and v by at least k. Such paths are often called *detours* [2, 6, 8, 18, 27]. Let us note that these objects are independent: while a long detour implies a long path, the converse direction fails.

▶ **Theorem 2.** *For every $k \in \mathbb{N}$ there exists a data structure that for a fully dynamic n-vertex graph G, supports the following operations in amortized $2^{\mathcal{O}(k^3)} \log n + \mathcal{O}(\log^2 n \log^2 \log n)$ time.*

- *Insert(u, v): insert an edge uv, provided it does not exist in G;*
- *Delete(u, v): delete an edge uv, provided it exists in G;*
- *LongDetour(u, v): decide whether there is a path of length at least $\mathsf{dist}_G(u, v) + k$ between u and v.*

The data structure can be initialized for an edgeless G in time $2^{\mathcal{O}(k^3)} \cdot n$.

The static problem underlying Theorem 2 is called Long Detour: given an undirected graph G, vertices u, v, and parameter k, decide whether there is a simple path from u to v of length at least $\mathsf{dist}_G(u, v) + k$ in G. An FPT algorithm for this problem was first given by Bezáková, Curticapean, Dell, and Fomin [8]. In fact, as we will later discuss, the approach proposed by Bezáková et al. in [8] is the main point of inspiration for our proofs of Theorems 1 and 2, along with the results of Korhonen et al. [30] and of Korhonen [29] for the *dynamic treewidth problem* – maintaining an approximate tree decomposition of a fully dynamic graph of bounded treewidth.

Finally, we show the robustness of our approach by applying it to a non-parameterized problem: detection of a path of given parity between a prescribed pair of vertices. Specifically, we prove the following.

▶ **Theorem 3.** *There exists a data structure that for a fully dynamic n-vertex graph G, supports the following operations in amortized $\mathcal{O}(\log^2 n \log^2 \log n)$ time.*

- *Insert(u, v): insert an edge uv, provided it does not exist in G;*
- *Delete(u, v): delete an edge uv, provided it exists in G;*
- *EvenPath(u, v): decide whether there is a path of even length between u and v;*
- *OddPath(u, v): decide whether there is a path of odd length between u and v.*

The data structure can be initialized for an edgeless G in time $\mathcal{O}(n)$.

Techniques. The main engine behind all our theorems is a stronger variant of the *delayed edge insertion* technique. Eppstein et al. [17] observed that dynamic planarity testing can be reduced to maintaining a dynamic planar graph and checking if a specified edge insertion violates planarity. From the perspective of amortized operation time, one can simply keep a pile of rejected edges and try inserting them again after some edge is removed. Naturally, this trick works for other graph properties as well [11, 30]. It can also be implemented on the level of connected components by keeping a separate pile for each component [25].

For the sake of detecting a (u, v)-path of length at least k, we want to apply the win-win approach developed for the static version [8, 11]: either a relevant part of the graph has sufficiently large treewidth (or treedepth), and the answer is yes, or treewidth is small and we can employ dynamic programming. The need to look at a "relevant" part comes from the

fact that large treewidth does not ensure the existence of a long (u, v)-path for an arbitrary pair (u, v). For such a path, a stronger condition is necessary: there needs to be a (relevant) biconnected component of large treewidth, located *between* u and v.

Our strategy globally maintains a subgraph of bounded treewidth [29] and, for each biconnected component, implicitly store a pile of (remaining) rejected edges – those that make the treewidth grow beyond some threshold. This is significantly harder than maintaining such piles on the level of connected components because single edge insertion can merge $\Omega(n)$ biconnected components into one while single edge deletion can revert this transformation. To mitigate this, we keep the rejected edges in a separate data structure: we adapt the dynamic biconnectivity data structure by Holm, Nadara, Rotenberg and Sokołowski [23] to support the following operations: (1) mark edge e, and (2) return some marked edge (if there is any) in the biconnected component containing vertices u, v. By combining these two data structures, we can effectively say that either there is a biconnected component of large treewidth between u and v (and then the answer is yes) or that this part of the graph has small treewidth, which allows us to read the answer from a DP state in the dynamic tree decomposition.

2 Preliminaries

In this work, all graphs are simple, finite and undirected. If $S \subseteq V(G)$, then $G[S]$ denotes the subgraph of G induced by S. For $u, v \in V(G)$, we write $G + uv$ (respectively, $G - uv$) to describe the supergraph (subgraph) of G formed by adding (removing) the edge uv. Note that $G + uv = G$ if $uv \in E(G)$, and similarly $G - uv = G$ if $uv \notin E(G)$. More generally, if H is a graph with $V(H) \subseteq V(G)$, then $G + H$ (resp., $G - H$) is formed from G by adding all edges of H that are absent from G (resp., removing all edges of H present in G).

2.1 Combinatorial properties of graphs

Biconnectivity and relevant subgraphs. We present two definitions of *biconnectivity* in graphs: one describing it as a relation on the vertices of the graph, and the other phrasing it as an equivalence relation on the edges of the graph. Namely, we say that two vertices u, v of a graph G are biconnected if either $uv \in E(G)$, or there exist two paths, vertex-disjoint except from their endpoints, connecting u and v. Equivalently, u and v are in the same biconnected component of G if u and v cannot be separated by removing a single vertex (a *cut-vertex*) different than u and v. Next, we will say that two edges e, f are biconnected if there exists a simple cycle in G including both e and f in its edge set. It is standard that the biconnectivity relation is an equivalence relation on the edges of G (but not the vertices of G) [21, 41].

We borrow the following definition of a *relevant subgraph* from [8]. Let G be a graph, and let $s, t \in V(G)$. The (s, t)*-relevant part* of G is the graph $G_{s,t}$ induced by all vertices that are contained in at least one (s, t)-path. The following lemma shows that we can recover $G_{s,t}$ dynamically, provided we can maintain biconnected components, motivating our use of dynamic biconnectivity data structures.

▶ **Observation 4.** *For any graph G, and any two vertices $s, t \in V(G)$, s and t are in the same biconnected component B of $G + st$, and $B = G_{s,t} + st$.*

Proof. Indeed, if s, t are not joined by a path of length at least 2 in G (in particular if they are not connected), B consists only of st, and satisfies the observation. Otherwise, any (s, t)-path P along with st forms a cycle in $G + st$, meaning $V(G_{s,t}) \subseteq V(B)$. Conversely,

for any $b \in V(B)$, Menger's theorem provides us with a path from b to s and one from b to t that are disjoint, and concatenating them yields $b \in V(G_{s,t})$. That is, $V(B) = V(G_{s,t})$, and since st is the only edge of B that may not be present in $G_{s,t}$, $B = G_{s,t} + st$. ◀

Treewidth. We use the classical notion of treewidth [40]. A *tree decomposition* of a graph G is a pair $\mathcal{T} = (T, \mathsf{bag})$, where T is a tree and $\mathsf{bag} : V(T) \to 2^{V(G)}$ (a mapping of nodes of T to sets of vertices, called *bags*), with the properties that: (i) each vertex of G belongs to a non-empty set of bags inducing a connected subtree of T, and (ii) for each edge of G, both of its endpoints belong together to some bag of $\mathcal{T}$. The *width* of the tree decomposition is the maximum cardinality of any bag, minus 1. The *treewidth* of G, denoted $\mathrm{tw}(G)$, is the minimum possible width of a tree decomposition of G.

In this work we implicitly use several combinatorial properties of treewidth: for $u, v \in V(G)$, we have $\mathrm{tw}(G) \leqslant \mathrm{tw}(G + uv) \leqslant \mathrm{tw}(G) + 1$; and $\mathrm{tw}(G)$ is equal to the maximum treewidth of the biconnected components of G (this follows e.g. from [14, Lemma 1]).

2.2 Dynamic data structures

Dynamic biconnectivity. Our work heavily relies on data structures that efficiently maintain the structure of biconnected components in a fully dynamic graph. Here, we use the data structure by Holm, Nadara, Rotenberg and Sokołowski [23]:

▶ **Theorem 5** ([23, Theorem 1]). *There exists a data structure maintaining a fully dynamic n-vertex undirected graph G in the word RAM model with $\Omega(\log n)$ word size. The data structure can handle the following updates and queries for vertices $u, v \in V(G)$:*

- *Insert(u, v) and Delete(u, v): insert or remove the edge uv;*
- *IsBiconnected(u, v): determine whether two vertices u and v are biconnected.*

Each operation can be performed in amortized time $\mathcal{O}(\log^2 n \log^2 \log n)$.

For the purposes of our work, we need to extend the data structure above to support *marking* edges. Namely, some edges of the dynamic graph can be designated as *marked*, and we want to additionally support searching marked edges in the biconnected components of the graph. Formally, we show the following:

▶ **Theorem 6.** *The data structure from Theorem 5 can be extended to support* marking *some of the edges of the graph. In this regime, the data structure can also support the following operations for an edge $e = uv$ of the current graph G:*

- *Mark(uv) and Unmark(uv): mark or unmark uv;*
- *FindMarkedEdge(uv): return a marked edge f biconnected with uv, or correctly output* Null *if no such edge exists.*

The amortized time complexity remains $\mathcal{O}(\log^2 n \log^2 \log n)$.

The proof of Theorem 6 follows by inspecting the implementation of the original biconnectivity data structure: there, biconnectivity in a dynamic graph G is maintained using an intricate dynamic *tree* data structure supporting an extensive list of updates and queries, including marking *vertices* of the tree and finding marked *vertices* in parts of the tree corresponding to the biconnected components of G. By suitably extending this tree data structure, we show that it can also be exploited to locate marked *edges* in the biconnected components of G. We move the formal exposition to Appendix A of the appendix.

Dynamic treewidth. We also rely on the dynamic treewidth data structure of Korhonen [29]. Specifically, we will need to maintain a tree-decomposition of some width t, as well as the ability to query for shortest and longest paths in G. The following lemma formalizes this. Its proof is a standard construction of a tree decomposition automaton, which we defer to Appendix B of the appendix.

▶ **Lemma 7.** *Let $t \in \mathbb{N}$. There exists a data structure maintaining an n-vertex undirected graph G and supporting the following operations for two vertices u, v:*

- *Insert(u, v): insert an edge with endpoints u, v as long as $\mathrm{tw}(G+uv) \leqslant t$; if $\mathrm{tw}(G+uv) > t$, the insertion is aborted,*
- *Delete(u, v): remove edge with endpoints u, v,*
- *ShortestPath(u, v) and LongestPath(u, v): output the lengths of the shortest and longest (u, v)-paths in G; or $+\infty$ and $-\infty$, respectively, if there is no (u, v)-path.*

Each operation can be performed in amortized time $2^{\mathcal{O}(t^3)} \cdot \log n$.

Dynamic bipartiteness. In the proof of Theorem 3 we will need a data structure that maintains a bipartite graph and reports when inserting some edge would create an odd cycle. It is well-known that dynamic bipartiteness can be implemented in the same way as dynamic pairwise connectivity [39]. We give a simple black-box reduction and employ the deterministic dynamic connectivity data structure [22]. Note that similar data structures have already been proposed (see e.g., [28, Theorem 8]), but for completeness we give an interface and an implementation of such a data structure below.

▶ **Lemma 8.** *There exists a data structure maintaining an n-vertex undirected bipartite graph G and supporting the following operations:*

- *Insert(u, v): insert edge uv as long as the insertion does not create an odd cycle; if $G + uv$ has an odd cycle, the insertion is aborted,*
- *Delete(u, v): remove edge uv from the graph,*
- *EvenPath(u, v) and OddPath(u, v): decide whether there is a (u, v)-path of even (odd) length.*

Each operation can be performed in amortized time $\mathcal{O}(\log^2 n)$.

Proof. Consider graph $\widehat{G}$ obtained from G by replacing each vertex v with two copies v_0, v_1 and replacing each edge uv with two edges v_0u_1, v_1u_0. We rely on the following easy observation.

▶ **Fact 9.** *Graph G contains an odd cycle passing through a vertex v if and only if graph $\widehat{G}$ contains a path from v_0 to v_1.*

Our data structure maintains the graph $\widehat{G}$ using the connectivity data structure from [22], that is, Insert(u, v) is implemented by insertion of v_0u_1, v_1u_0, and likewise for Delete(u, v). When an edge uv is being added, we need to report if $G+uv$ is still bipartite. Since insertions violating bipartiteness are rejected, we work under the assumption that G is bipartite. Observe that a hypothetical odd cycle in $G + uv$ must use the edge uv, hence it must pass through v. By Fact 9, this can be detected by testing if v_0, v_1 are connected in $\widehat{G}+v_0u_1+v_1u_0$. If they are, we revert the two edge insertions in $\widehat{G}$ and report that $G + uv$ has an odd cycle. Also, observe that EvenPath(u, v) (respectively, OddPath(u, v)) can be implemented by verifying the connectivity in $\widehat{G}$ between u_0 and v_0 (resp., u_0 and v_1).

Each operation in our data structure invokes $\mathcal{O}(1)$ operations in the connectivity data structure for $\widehat{G}$. Since the data structure from [22] handles each operation in amortized time $\mathcal{O}(\log^2 n)$, the same applies to our case. ◀

3 Dynamic (s,t)-Path and (s,t)-Detour

In this section we prove Theorems 1 and 2 by proposing appropriate dynamic data structures testing for long (s,t)-paths and (s,t)-detours. We begin by setting up the combinatorial foundations of our data structures in Section 3.1, and then we implement the data structures themselves in Sections 3.2 and 3.3.

3.1 Extracting long paths and detours from high-treewidth graphs

Before defining the data structures, we show here a few structural results ensuring that, for a given pair s,t of vertices of G, if the (s,t)-relevant part of G has sufficiently large treewidth, then G admits both long (s,t)-paths and long (s,t)-detours.

For (s,t)-LongPath, we will need the following folklore result to ensure the existence of a k-path.

▶ **Lemma 10.** *If B is a biconnected graph such that $\mathrm{tw}(B) \geqslant 2k-1$, then for any $s,t \in V(B)$, there exists an (s,t)-path of length at least k. Moreover, for any graph G and $s,t \in V(G)$ such that $\mathrm{tw}(G_{s,t}) \geqslant 2k-1$, the same holds.*

Proof. Since B has treewidth at least $2k-1$, a result of Birmelé [9] ensures that B contains a cycle C of length at least $2k$. If $k=1$, the result holds trivially. If $k \geqslant 2$, B cannot be a single edge, and for any $s,t \in V(B)$, Menger's theorem applied to $\{s,t\}$ and C yields a path from s to $c_s \in C$ and a vertex-disjoint path from t to $c_t \in C$. Then, taking these two paths along with longer arc of C between c_s and c_t yields a path of length at least $|C|/2 \geqslant k$ between them. Finally, note that for any G and $s,t \in V(G)$, if $\mathrm{tw}(G_{s,t}) \geqslant 2k-1$, then Observation 4 ensures the biconnected component of $G+st$ containing s,t is exactly $G_{s,t}+st$, so $\mathrm{tw}(G_{s,t}+st) \geqslant 2k-1$ and applying the first part of the lemma to $G_{s,t}+st$ concludes. ◀

The analogue of Lemma 10 for (s,t)-Detour was shown in [8]: imposing higher treewidth (in the relevant part of G) guarantees also a long detour.

▶ **Theorem 11** (Theorem 3.6 in [8]). *For any graph G and $s,t \in V(G)$ such that $\mathrm{tw}(G_{s,t}) > 32k+46$, there exists an (s,t)-path of length at least $\mathsf{dist}_G(s,t)+k$.*

3.2 Dynamic (s,t)-Path

Dynamic data structure. For any fixed k, our data structure for $(\geqslant k)$-paths instantiates the following auxiliary data structures for a fully dynamic graph G:

- A biconnectivity data structure BC for G given by Theorem 6,
- A dynamic treewidth data structure TW of a *subgraph* $H \subseteq G$ such that $\mathrm{tw}(H) \leqslant 2k$ given by Lemma 7.

Then, our calls to the methods of these data structures are prefixed with BC and TW, respectively. The subgraph H contains all the edges of G whose *last* insertion was accepted by TW. Then, the marked edges of (biconnected components of) G are exactly $E(G) \setminus E(H)$. We stress that if an edge $e \in E(G)$ is not in H, we only know that every past attempt to insert e into TW has failed (even though the insertion of e into H may be currently possible). This phenomenon also naturally appears in Eppstein's technique of delaying invariant-breaking insertions [17].

The implementation of our data structure is given in Algorithms 1–3. Insertion of an edge uv (Algorithm 1) resolves simply to attempting the insertion of uv to BC (which always succeeds) and TW (which may fail); if TW rejects the insertion, we mark uv in BC so as to

preserve the invariant. Removing the edge uv (Algorithm 2) is also simple: we simply drop uv from the auxiliary data structures. Finally, in order to test if G contains an (s,t)-path of length at least k (Algorithm 3), we begin by temporarily adding a helper edge st to BC, if not already present in G. We then iterate the marked edges in the biconnected component containing st, successively adding them to TW until either all such marked edges are exhausted, or some insertion is aborted due to the treewidth of the graph becoming too large. In the former case, all edges of $G_{s,t}$ are present in TW and so the length of the longest (s,t)-path can be simply queried in TW; in the latter, we claim that a sufficiently long (s,t)-path exists due to Lemma 10.

Algorithm 1 Insert.

```
function Insert(u, v)
    bc.Insert(u, v)
    if tw.Insert(u, v) aborts then
        bc.Mark(uv)
```

Algorithm 2 Delete.

```
function Delete(uv)
    bc.Delete(uv)
    tw.Delete(uv)
```

Algorithm 3 LongPath.

```
function LongPath(s, t)
    hasHelperEdge ← False
    if st ∉ E(G) then
        bc.Insert(s, t)
        hasHelperEdge ← True
    while True do
        e ← bc.FindMarkedEdge(st)
        if e = Null then break
        if tw.Insert(e) aborts then                    ▷ tw(H + e) > 2k
            if hasHelperEdge then bc.Delete(st)
            return Yes
        else
            bc.Unmark(e)
    if hasHelperEdge then bc.Delete(st)
    return tw.LongestPath(s, t) ⩾ k
```

The following lemma proves the correctness of LongPath.

▶ **Lemma 12.** *For any $s,t \in V(G)$, LongPath(s,t) correctly outputs whether s and t are joined by a path of length at least k.*

Proof. We fix G to be the initial graph before the query, and $H \subseteq G$ the graph held by the treewidth data structure. Then, let G', H' be the final states of the graphs G, H, respectively, at the end of the run of LongPath(s,t), just before the helper edge st is (potentially) removed from BC.

Then, the first steps of the algorithm ensure $st \in E(G')$ in any case. In particular, s and t belong to the same biconnected component of $G' = G + st$, which by Observation 4 is exactly $G_{s,t} + st$ (note that possibly $G_{s,t} = \{st\}$). Let us note that st can only be added to H' if it was already present in G (otherwise, if $st \notin E(G)$, then st added temporarily to BC as an unmarked edge and cannot be inserted into TW by LongPath). Hence $H' \subseteq G$. Assume first that the while loop is broken when $e = \mathsf{Null}$. Then, all edges of $(G \setminus H) \cap G_{s,t}$ have been added to H', meaning $H'[V(G_{s,t})] = G_{s,t}$. That is, all paths between s and t in G are contained in the graph H' held by TW, and so TW.LongestPath(s,t) correctly outputs the length of the longest (s,t)-path in G. Otherwise, the while loop returns **Yes** when TW rejects adding some edge of $(G \setminus H) \cap G_{s,t}$, say uv, to H'. That is, we have $\mathrm{tw}(H') \leqslant 2k$ and $\mathrm{tw}(H' + uv) > 2k$ by Lemma 7. Consider then the graph $H'' = H + G_{s,t} + st$, note that $H' \subseteq H'' \subseteq G'$, and therefore $V(G_{s,t})$ induces the biconnected component $G_{s,t} + st$ in H'' (because it did in G'). Since we had $\mathrm{tw}(H') \leqslant 2k$, all biconnected components of H'' other than $G_{s,t} + st$ still have treewidth at most $2k$, but $\mathrm{tw}(H'') \geqslant \mathrm{tw}(H' + uv) > 2k$. Therefore, since the treewidth of H'' is the maximum treewidth of its biconnected components, we must have $\mathrm{tw}(G_{s,t} + st) > 2k$. Then, $\mathrm{tw}(G_{s,t}) > 2k - 1$ and Lemma 10 yields a k-path between s and t. ◀

▶ **Theorem 1.** *For every $k \in \mathbb{N}$ there exists a data structure that for a fully dynamic n-vertex graph G, supports the following operations in amortized $2^{\mathcal{O}(k^3)} \log n + \mathcal{O}(\log^2 n \log^2 \log n)$ time.*

- *Insert(u,v): insert an edge uv, provided it does not exist in G;*
- *Delete(u,v): delete an edge uv, provided it exists in G;*
- *LongPath(u,v): decide whether there is a path of length at least k between u and v.*

The data structure can be initialized for an edgeless G in time $2^{\mathcal{O}(k^3)} \cdot n$.

Proof. The fully dynamic graph G our data structure maintains is exactly the G maintained by the biconnectivity data structure, so clearly Insert and Delete are sound. Then Lemma 12 shows the soundness of LongPath. It remains to justify the amortized runtime of all three operations. Updates Insert, Delete clearly run in amortized time $2^{O(k^3)} \log n + O(\log^2 n \log^2 \log n)$ by Theorem 6 and Lemma 7; and Insert adds at most one marked edge in BC. Moreover, LongPath runs in constant time, plus a constant number of calls to TW and BC, plus an additional constant number of calls to TW and BC for each edge unmarked in the while loop; moreover, no new edges become marked in LongPath. Hence the running time of LongPath is amortized by the decrease in the number of marked edges in BC, and so by Theorem 6 and Lemma 7, all operations run in amortized time $2^{O(k^3)} \log n + O(\log^2 n \log^2 \log n)$. ◀

3.3 Dynamic (s,t)-Detour

Dynamic data structure. For any fixed k, our data structure for $(\geqslant k)$-detours instantiates the following auxiliary data structures for a fully dynamic graph G:

- A biconnectivity data structure BC for G given by Theorem 6,
- A dynamic treewidth data structure of a *subgraph* $H \subseteq G$ such that $\mathrm{tw}(H) \leqslant 32k + 47$ guaranteed by Lemma 7.

Again, for any biconnected component B of G, the marked edges of B are exactly $E(G) \setminus E(H)$.

Queries Insert and Delete are identical to the previous Algorithms 1 and 2. Then, LongDetour (Algorithm 4) is almost identical to Algorithm 3, given that TW is initialized with a higher treewidth bound. The only difference is essentially the condition on the length of the longest (s,t)-path tested in the low-treewidth case.

Algorithm 4 LongDetour.

```
function LongDetour(s, t)
    hasHelperEdge ← False
    if st ∉ E(G) then
        bc.Insert(s, t)
        hasHelperEdge ← True
    while True do
        e ← bc.FindMarkedEdge(st)
        if e = Null then break
        if tw.Insert(e) aborts then                    ▷ tw(H + e) > 32k + 47
            if hasHelperEdge then bc.Delete(st)
            return Yes
        else
            bc.Unmark(e)
    if hasHelperEdge then bc.Delete(st)
    return tw.LongestPath(s, t) − tw.ShortestPath(s, t) ⩾ k
```

▶ **Lemma 13.** *For any $s, t \in V(G)$, LongDetour(s, t) correctly outputs whether s and t are joined by a path of length at least $\mathsf{dist}_G(s, t) + k$.*

Proof. The proof follows that of Lemma 12 verbatim except for the following. Let G', H' be the graphs held by bc and tw at the end of the run of LongDetour, before (potentially) removing st from bc. If the while loop is broken, tw.LongestPath(s, t) − tw.ShortestPath(s, t) correctly outputs the order of the longest (s, t)-detour in G. Otherwise, the while loop returns **Yes** when we now have $\mathrm{tw}(H' + uv) > 32k + 47$ for some edge $uv \in E(G_{s,t})$, by Lemma 7. We argue in the same way that if $\mathrm{tw}(G_{s,t}) > 32k + 46$, then Theorem 11 yields a k-detour between s and t in G. ◀

The time complexity analysis is exactly the same as in the final proof of Theorem 1, which allows us to conclude the proof of Theorem 2.

4 Dynamic Even/Odd Path

We move on to the description of the data structure allowing querying for the (s, t)-paths of a given parity. The implementation of this data structure will follow the blueprint laid out in Section 3, only that we will use the dynamic bipartiteness data structure in place of dynamic treewidth. The reason is the well-known observation correlating the parities of (s, t)-paths in G to the bipartiteness of the (s, t)-relevant part of G:

▶ **Lemma 14** ([34, Lemma 3])**.** *G contains (s, t)-paths of both parities if and only if $G_{s,t}$ is not bipartite.*

Dynamic data structure. We instantiate the following auxiliary data structures:

- A biconnectivity data structure bc for G given by Theorem 6,
- A dynamic bipartiteness data structure bp of a bipartite subgraph $H \subseteq G$ given by Lemma 8.

Analogously to the long path and long detour data structures, inserting (resp., removing) an edge from the data structure resolves to inserting (resp., removing) the edge – if possible – within both bc and bp (Algorithms 5 and 6); and if the insertion of the edge to bp fails,

the edge is marked in BC. We only implement the even path query here; the odd path query is completely analogous. In order to test whether there exists an even (s,t)-path (Algorithm 7), we similarly temporarily insert an unmarked helper edge st to BC; and iterate the marked edges in the biconnected component containing st, progressively adding them to BP. Again, if all such edges are successfully inserted, then $G_{s,t}$ is bipartite and the parity of the length of the (s,t)-path can be verified directly in BP; otherwise, an even (s,t)-path exists by Lemma 14.

Algorithm 5 Insert for Even/Odd Path.

```
function Insert(u, v)
    BC.Insert(u, v)
    if BP.Insert(u, v) aborts then
        BC.Mark(uv)
```

Algorithm 6 Delete for Even/Odd Path.

```
function Delete(uv)
    BC.Delete(uv)
    BP.Delete(uv)
```

Algorithm 7 EvenPath.

```
function EvenPath(s, t)
    hasHelperEdge ← False
    if st ∉ E(G) then
        BC.Insert(s, t)
        hasHelperEdge ← True
    while True do
        e ← BC.FindMarkedEdge(st)
        if e = Null then break
        if BP.Insert(e) aborts then                ▷ H + e is not bipartite
            if hasHelperEdge then BC.Delete(st)
            return Yes
        else
            BC.Unmark(e)
    if hasHelperEdge then BC.Delete(st)
    return BP.EvenPath(s, t)
```

Note that the operation OddPath is implemented exactly as EvenPath, only that in the case where $G_{s,t}$ is bipartite, we conclude by returning BP.OddPath(s,t) instead of BP.EvenPath(s,t).

The amortized time complexity analysis is analogous to those in Section 3, only that each operation in BP runs in amortized $\mathcal{O}(\log^2 n)$ time. Therefore, each operation of our data structure takes amortized $\mathcal{O}(\log^2 n \log^2 \log n)$ time.

5 Conclusions

We presented efficient data structures for three dynamic path length problems: long (s,t)-path, long (s,t)-detour, and even/odd (s,t)-path. We now briefly discuss possible improvements and generalizations.

- Our strategy to obtain long paths and long detours works more generally for any problem to which the following win/win approach applies: when querying vertices s,t, low treewidth of the graph allows us to use a tree decomposition automaton answering the query efficiently, and the high treewidth of the relevant subgraph $G_{s,t}$ immediately yields a **Yes** or **No** answer to the query over s,t.
- The dependence of $\mathcal{O}(\log^2 n \log^2 \log n)$ on the number n of vertices only stems from the time complexity of the dynamic biconnectivity data structure; therefore, any improvement to the time complexity of this data structure automatically implies a better running time for each of the data structures presented in this work.
- We believe that the $2^{\mathcal{O}(k^3)}$ factor in the running time of the data structures for long path and long detour is not optimal, and that it could be improved to $2^{\mathcal{O}(k \log k)}$ or even $2^{\mathcal{O}(k)}$. The first improvement would require us to avoid the use of the Bodlaender–Kloks *tree decomposition automaton* tracking the *exact* value of the treewidth of the graph (Lemma 18). This seems plausible as we basically need the dynamic treewidth data structure to accept edge insertions as long as they do not make the treewidth of the corresponding biconnected component grow above the threshold. Then, to achieve the $2^{\mathcal{O}(k)}$ dependence on k, one also has to improve the evaluation time of the tree decomposition automaton for the longest (u,v)-path (Lemma 17); this seems to be attainable via Cut & Count or Gaussian elimination methods (see [12, Section 11.2]). Any improvement past $2^{\mathcal{O}(k)}$ is highly unlikely: for $k = n-1$, the long path problem is equivalent to the Hamiltonian Path problem (with prescribed endpoints), for which no $2^{o(n)}$-time algorithm exists under Exponential Time Hypothesis [12, Theorem 14.6].

References

1 Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014*, pages 434–443. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.53.

2 Shyan Akmal, Virginia Vassilevska Williams, Ryan Williams, and Zixuan Xu. Faster Detours in Undirected Graphs. In *31st Annual European Symposium on Algorithms, ESA 2023*, volume 274 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:17, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2023.7.

3 Josh Alman, Matthias Mnich, and Virginia Vassilevska Williams. Dynamic parameterized problems and algorithms. *ACM Transactions on Algorithms*, 16(4):45:1–45:46, 2020. doi:10.1145/3395037.

4 Shinwoo An, Kyungjin Cho, Leo Jang, Byeonghyeon Jung, Yudam Lee, Eunjin Oh, Donghun Shin, Hyeonjun Shin, and Chanho Song. Dynamic parameterized problems on unit disk graphs. In *35th International Symposium on Algorithms and Computation, ISAAC 2024*, LIPIcs, pages 6:1–6:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.ISAAC.2024.6.

5 Max Bannach, Zacharias Heinrich, Rüdiger Reischuk, and Till Tantau. Dynamic kernels for hitting sets and set packing. *Algorithmica*, 84(11):3459–3488, 2022. doi:10.1007/S00453-022-00986-0.

6 Eli Berger, Paul D. Seymour, and Sophie Spirkl. Finding an induced path that is not a shortest path. *Discrete Mathematics*, 344(7):112398, 2021. doi:10.1016/J.DISC.2021.112398.

7 Christian Bertram, Deborah Haun, Mads Vestergaard Jensen, and Tuukka Korhonen. Dynamic meta-kernelization. *CoRR*, abs/2511.03461, 2025. Accepted to STOC 2026. doi:10.48550/arXiv.2511.03461.

8 Ivona Bezáková, Radu Curticapean, Holger Dell, and Fedor V Fomin. Finding detours is fixed-parameter tractable. *SIAM Journal on Discrete Mathematics*, 33(4):2326–2345, 2019. doi:10.1137/17M1148566.

9 Etienne Birmele. Tree-width and circumference of graphs. *Journal of Graph Theory*, 43(1):24–25, 2003. doi:10.1002/jgt.10099.

10 Hans L. Bodlaender and Ton Kloks. Better algorithms for the pathwidth and treewidth of graphs. In *18th International Colloquium on Automata, Languages and Programming, ICALP 1991*, Lecture Notes in Computer Science, pages 544–555. Springer, 1991. doi:10.1007/3-540-54233-7_162.

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

12 Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015. doi:10.1007/978-3-319-21275-3.

13 Daniel Dadush, Michał Pilipczuk, Amadeus Reinald, Marek Sokołowski, and Michał Włodarczyk. Dynamic detours, 2026. doi:10.48550/arXiv.2605.03225.

14 Erik D. Demaine, Mohammad Taghi Hajiaghayi, and Dimitrios M. Thilikos. 1.5-approximation for treewidth of graphs excluding a graph with one crossing as a minor. In *5th International Workshop on Approximation Algorithms for Combinatorial Optimization, APPROX 2002*, Lecture Notes in Computer Science, pages 67–80. Springer, 2002. doi:10.1007/3-540-45753-4_8.

15 Zdenek Dvořák, Martin Kupec, and Vojtech Tůma. A dynamic data structure for MSO properties in graphs with bounded tree-depth. In *22nd Annual European Symposium on Algorithms, ESA 2014*, volume 8737 of *Lecture Notes in Computer Science*, pages 334–345. Springer, 2014. doi:10.1007/978-3-662-44777-2_28.

16 Zdenek Dvořák and Vojtech Tůma. A dynamic data structure for counting subgraphs in sparse graphs. In *13th International Symposium on Algorithms and Data Structures, WADS 2013*, volume 8037 of *Lecture Notes in Computer Science*, pages 304–315. Springer, 2013. doi:10.1007/978-3-642-40104-6_27.

17 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. Separator based sparsification. I. Planarity testing and minimum spanning trees. *Journal of Computer and System Sciences*, 52(1):3–27, 1996. doi:10.1006/jcss.1996.0002.

18 Fedor V. Fomin, Petr A. Golovach, William Lochet, Danil Sagunov, Saket Saurabh, and Kirill Simonov. Detours in directed graphs. *Journal of Computer and System Sciences*, 137:66–86, 2023. doi:10.1016/j.jcss.2023.05.001.

19 Alejandro Grez, Filip Mazowiecki, Michał Pilipczuk, Gabriele Puppis, and Cristian Riveros. Dynamic data structures for timed automata acceptance. *Algorithmica*, 84(11):3223–3245, 2022. doi:10.1007/s00453-022-01025-8.

20 Maximilian Probst Gutenberg, Virginia Vassilevska Williams, and Nicole Wein. New algorithms and hardness for incremental single-source shortest paths in directed graphs. In *52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*, pages 153–166. ACM, 2020. doi:10.1145/3357713.3384236.

21 Frank Harary. *Graph theory*. Addison-Wesley Publishing Co., Reading, Mass.-Menlo Park, Calif.-London, 1969.

22 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001. doi:10.1145/502090.502095.

23 Jacob Holm, Wojciech Nadara, Eva Rotenberg, and Marek Sokołowski. Fully dynamic biconnectivity in $\widetilde{O}(\log^2 n)$ time. In *57th Annual ACM Symposium on Theory of Computing, STOC 2025*, pages 156–165. ACM, 2025. doi:10.1145/3717823.3718270.

24 Jacob Holm, Wojciech Nadara, Eva Rotenberg, and Marek Sokołowski. Fully dynamic biconnectivity in $\widetilde{O}(\log^2 n)$ time. *CoRR*, abs/2503.21733, 2025. Full version of [23]. doi:10.48550/arXiv.2503.21733.

25 Jacob Holm and Eva Rotenberg. Fully-dynamic planarity testing in polylogarithmic time. In *52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*, pages 167–180, New York, NY, USA, 2020. ACM. doi:10.1145/3357713.3384249.

26 Yoichi Iwata and Keigo Oka. Fast dynamic graph algorithms for parameterized problems. In *14th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2014*, volume 8503 of *Lecture Notes in Computer Science*, pages 241–252. Springer, 2014. doi:10.1007/978-3-319-08404-6_21.

27 Ashwin Jacob, Michał Włodarczyk, and Meirav Zehavi. Long directed detours: Reduction to 2-disjoint paths. *Information Processing Letters*, 186:106491, 2024. doi:10.1016/J.IPL.2024.106491.

28 Manas Jyoti Kashyop, N. S. Narayanaswamy, Meghana Nasre, and Sai Mohith Potluri. Trade-offs in dynamic coloring for bipartite and general graphs. *Algorithmica*, 85(4):854–878, 2023. doi:10.1007/S00453-022-01050-7.

29 Tuukka Korhonen. Dynamic treewidth in logarithmic time. In *66th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2025*, pages 787–796. IEEE, 2025. doi:10.1109/FOCS63196.2025.00042.

30 Tuukka Korhonen, Konrad Majewski, Wojciech Nadara, Michał Pilipczuk, and Marek Sokołowski. Dynamic treewidth. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023*, pages 1734–1744, 2023. doi:10.1109/FOCS57990.2023.00105.

31 Tuukka Korhonen, Konrad Majewski, Wojciech Nadara, Michał Pilipczuk, and Marek Sokołowski. Dynamic treewidth. *CoRR*, abs/2304.01744, 2023. Full version of [30]. doi:10.48550/arXiv.2304.01744.

32 Tuukka Korhonen, Wojciech Nadara, Michał Pilipczuk, and Marek Sokołowski. Fully dynamic approximation schemes on planar and apex-minor-free graphs. In David P. Woodruff, editor, *2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024*, pages 296–313. SIAM, 2024. doi:10.1137/1.9781611977912.12.

33 Tuukka Korhonen and Marek Sokołowski. Almost-linear time parameterized algorithm for rankwidth via dynamic rankwidth. In *56th Annual ACM Symposium on Theory of Computing, STOC 2024*, pages 1538–1549. ACM, 2024. doi:10.1145/3618260.3649732.

34 Andrea S. LaPaugh and Christos H. Papadimitriou. The even-path problem for graphs and digraphs. *Networks*, 14(4):507–513, 1984. doi:10.1002/NET.3230140403.

35 Konrad Majewski, Michał Pilipczuk, and Marek Sokołowski. Maintaining CMSO_2 properties on dynamic structures with bounded feedback vertex number. In *40th International Symposium on Theoretical Aspects of Computer Science, STACS 2023*, volume 254 of *LIPIcs*, pages 46:1–46:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.STACS.2023.46.

36 Konrad Majewski, Michał Pilipczuk, and Anna Zych-Pawlewicz. Parameterized dynamic data structure for split completion. In *32nd Annual European Symposium on Algorithms, ESA 2024*, LIPIcs, pages 87:1–87:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.ESA.2024.87.

37 Jędrzej Olkowski, Michał Pilipczuk, Mateusz Rychlicki, Karol Węgrzycki, and Anna Zych-Pawlewicz. Dynamic data structures for parameterized string problems. In *40th International Symposium on Theoretical Aspects of Computer Science, STACS 2023*, volume 254 of *LIPIcs*, pages 50:1–50:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.STACS.2023.50.

38 Mihai Pătraşcu and Erik D. Demaine. Lower bounds for dynamic connectivity. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 546–553. ACM, 2004. doi:10.1145/1007352.1007435.

39 Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999. doi:10.1145/320211.320215.

40 Neil Robertson and Paul D. Seymour. Graph Minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984. doi:10.1016/0095-8956(84)90013-3.

41 Robert Endre Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985. doi:10.1137/0214061.

A Dynamic biconnectivity with marked edges

We will now sketch how to implement the marking extension in the dynamic biconnectivity data structure, that is, we show Theorem 6.

▶ **Theorem 6.** *The data structure from Theorem 5 can be extended to support* marking *some of the edges of the graph. In this regime, the data structure can also support the following operations for an edge $e = uv$ of the current graph G:*

- *Mark(uv) and Unmark(uv): mark or unmark uv;*
- *FindMarkedEdge(uv): return a marked edge f biconnected with uv, or correctly output* Null *if no such edge exists.*

The amortized time complexity remains $\mathcal{O}(\log^2 n \log^2 \log n)$.

In their implementation of the data structure, Holm et al. reduce the problem of dynamic biconnectivity to the *restricted dynamic tree cover level data structure*. We omit most details in the exposition of this data structure below, opting to present only the elements relevant to our proof.

Suppose G is a fully dynamic graph and F is its (dynamic) spanning forest (so $V(F) = V(G)$ and $E(F) \subseteq E(G)$). An *ℓ-level* data structure maintains for every pair of adjacent edges $e_1, e_2 \in E(F)$ the *cover level* of (e_1, e_2), denoted $c(e_1, e_2)$, which is an integer from $[-1, \ell]$. For any pair $e, f \in E(F)$ of non-adjacent edges in the same connected component of G, this induces the cover level $c(e, f)$ as the minimum cover level of any pair of adjacent edges on the unique simple path in F connecting e and f. If e and f are in distinct connected components of G, we declare that $c(e, f) = -1$. For the purposes of this section, it is enough to know that additional requirements placed on the definition of cover levels guarantee that two edges $e, f \in E(F)$ are biconnected in G if and only if $c(e, f) \geqslant 0$. The forest F, along with its cover levels, is manipulated and queried through an extensive list of operations, split into two types, *light* and *heavy*; roughly speaking, heavy operations modify the shape of F, while the light operations only maintain and query the cover levels of F.

Holm et al. implement the restricted dynamic tree cover level data structure along with several extensions; one of them is the *marking extension*, which we now describe. At any point of time, each vertex of F may be arbitrarily *marked* by the user of the data structure. The user can mark a vertex v via a light operation $\mathsf{Mark}(v)$, and similarly remove a mark from v via a light operation $\mathsf{Unmark}(v)$; neither operation modifies the cover levels of the pairs of edges of the underlying forest.[1] Marked vertices can be located in F using a light

[1] Formally, the *marking extension* of the cover level data structure in [23] allows placing *level-i marks* on vertices for each $i \in \{0, 1, \ldots, \ell\}$. However, in our own data structure we will not utilize the marking extension in its full generality. The notion of a marked vertex in our data structure directly corresponds

query FindFirstReach(p, q) with the following semantics: assume that $p, q \in V(F)$ are in the same connected component of F, and let P be the unique simple path in F with endpoints p and q. The query returns a marked vertex v that is 0-*reachable from* P: a vertex for which there exists an incident edge e and an edge $f \in E(P)$ such that the cover level of the pair e, f is at least 0. If no such vertex exists, the query returns $\perp$.[2] The query does not modify the marks or the underlying forest F.

In [24], it is shown that the tree cover level data structure with the marking extension can be implemented efficiently:

▶ **Lemma 15** ([24, Lemma 16]). *Let $\ell \in O(\log n)$. There exists an ℓ-level restricted dynamic tree cover level data structure with the marking extension that processes each heavy operation in amortized $O(\log^2 n \log^2 \log n)$ time, and each light operation in amortized $O(\log n \log^2 \log n)$ time.*

We are now ready to (informally) state the main reduction proved by Holm et al.:

▶ **Lemma 16** ([23, Lemma 2], informal). *Suppose that a restricted dynamic tree cover level data structure for n-vertex forests with $\ell \in O(\log n)$ levels and marking extension can be implemented to support each light operation in time $T_{\text{light}}(n)$ and each heavy operation in time $T_{\text{heavy}}(n)$. Then there exists dynamic biconnectivity data structure that is initialized with an edgeless n-vertex graph and processes any sequence of m updates in time $O(m \log^2 n + m \log n \cdot T_{\text{light}}(n) + m \cdot T_{\text{heavy}}(n))$.*

Note that Lemmas 15 and 16 together imply Theorem 5 from [23].

In order to show Theorem 6, we will extend the reduction described above by essentially deploying a *man-in-the-middle* attack. To do so, we initialize a dynamic biconnectivity data structure, itself spawning an instance of restricted dynamic tree cover level DT, and we create an additional private instance DP maintaining the same restricted dynamic tree cover level as DT (except for marks). Then, intercepting all calls to DT (while still performing them in DT), and suitably forwarding them to DP, allows us to emulate marked edges of G as marked vertices in DP. The details follow below.

Proof of Theorem 6. Let G be a dynamic graph, updated by edge insertions and removals, along with edge marking and unmarking. Our aim is to construct the data structure BC implementing biconnectivity in G with edge marks as required by the statement of Theorem 6.

Let $G^{(1)}$ be the (dynamic) 1-subdivision of G, i.e., the graph with vertex set $V(G) \cup \{t_e \mid e \in E(G)\}$ formed from G by replacing each edge $e = uv$ with a path ut_ev. We initialize the following data structures:

- BC$^{(1)}$: the basic dynamic biconnectivity data structure for $G^{(1)}$, provided by Theorem 5. As promised by Lemma 16, the data structure internally keeps, for some $\ell \in \mathcal{O}(\log n)$, a restricted cover level data structure DT with ℓ levels and marking extension, maintaining a dynamic spanning forest $F^{(1)}$ of $G^{(1)}$ and some set of user marks on the vertices of $F^{(1)}$, updated internally by BC$^{(1)}$.
- DP: a restricted cover level data structure DP with ℓ levels and marking extension, which will maintain the same dynamic spanning forest $F^{(1)}$ of $G^{(1)}$, but a different set of user marks (selected by us).

to 0-*marked vertices* in [23].

[2] The query also provides an extensive tie-breaking scheme, which is of no relevance here.

We will maintain the following invariant on vertex marks in DP: the original vertices $V(G)$ are kept unmarked in DP, and we declare that the vertex $t_e \in V(F^{(1)})$ is marked in DP whenever $e \in E(G)$ is marked in G.

Observe that two vertices $u, v \in V(G)$ are biconnected in G if and only if they are adjacent in G or biconnected in $G^{(1)}$; hence biconnectivity in G can be tracked by BC$^{(1)}$ and a dynamic set data structure maintaining $E(G)$. All updates performed by BC$^{(1)}$ to the instance DT, *except* DT.Mark and DT.Unmark, are forwarded verbatim by us to DP. Note that BC$^{(1)}$ does *not* query DP directly (i.e., it still only queries DT to support the biconnectivity queries). Instead, we will use DP to search for marked edges in G.

Note that each edge insertion or removal in BC (i.e., each edge insertion or removal in G) can be translated to a constant number of vertex and edge updates in $G^{(1)}$, maintained by BC$^{(1)}$. It remains to implement the updates and queries related to the edge marks in G. Naturally, marking (respectively, unmarking) an edge $e \in E(G)$ with BC.MARK (resp. BC.UNMARK) is implemented by calling DP.Mark(t_e) (resp. DP.Unmark(t_e)). The query BC.FINDMARKEDEDGE(e) (given $e \in E(G)$, return marked $f \in E(G)$ biconnected with e) is implemented as follows. If e is already marked in BC, we can return e. Otherwise, let u be an arbitrary neighbor of t_e in the spanning forest $F^{(1)}$ of $G^{(1)}$ maintained by DP, and let us consider the result of calling DP.FindFirstReach(t_e, u).

- If the call returns a vertex of $G^{(1)}$, then it is a vertex t_f of the subdivision distinct from t_e (since only subdivision vertices are marked in $G^{(1)}$ and e is unmarked in G). Moreover, by the definition of FindFirstReach and the definition of cover levels in $G^{(1)}$, there exists an edge $t_f w \in E(F^{(1)})$ incident to t_f such that the edges $t_e u$ and $t_f w$ are biconnected in $G^{(1)}$. Thus there exists a simple cycle in $G^{(1)}$ containing these two edges. This cycle corresponds to a simple cycle in G containing both e and f, and so f can be safely returned as a correct answer to the query.
- Now suppose the call returns $\perp$. We claim that there is no marked edge in G biconnected with e. Suppose otherwise that such an edge f exists (so, equivalently, t_f is marked in $G^{(1)}$). Let w be a neighbor of t_f in $F^{(1)}$. Since e and f are biconnected in G, there exists a simple cycle in G containing both e and f, which translates to a simple cycle in $G^{(1)}$ containing $t_e u$ and $t_f w$ as edges. Hence $c(t_e u, t_f w) \geqslant 0$ in $F^{(1)}$ and thus t_f is 0-reachable from $t_e u$. Since t_f is marked in DP, this yields a contradiction.

By Lemmas 15 and 16, all updates relayed to DP performed by the original biconnectivity data structure BC$^{(1)}$ take amortized $O(\log^2 n \log^2 \log n)$ time. Moreover, each operation related to the edge marks in G (marking or unmarking of an edge of G, or searching for a marked biconnected edge) translates in constant time to a constant number of light operations in BC$^{(1)}$; therefore, all such operations are supported in additional amortized $O(\log n \log^2 \log n)$ time by Lemma 15. ◀

B Dynamic treewidth with shortest and longest paths

In this section we prove Lemma 7. We assume the reader's familiarity with the terminology of boundaried graphs and tree decomposition automata, introduced in [31, Appendix A]. The main part of the proof is the following construction of an automaton, which essentially boils down to writing a standard dynamic programming algorithm for the LONGEST PATH problem working on a tree decomposition.

▶ **Lemma 17.** *For every $t \in \mathbb{N}$, there is a tree decomposition automaton $\mathcal{A}_t^{\max}$ such that for any boundaried tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of width t of a boundaried graph G, if ρ is the run of $\mathcal{A}_t^{\max}$ on $\mathcal{T}$ and x is the root of T, then from the state $\rho(x)$ one can compute, in time $2^{\mathcal{O}(t \log t)}$, the following information: for any $u, v \in \partial G$, what is the maximum length of a simple (u, v)-path in G? The evaluation time of $\mathcal{A}_t^{\max}$ is $2^{\mathcal{O}(t \log t)}$.*

Proof. Recall that in the setting of boundaried graphs, we assume that all the vertices come from a common reservoir of vertices Ω. For a given set $B \subseteq \Omega$ with $|B| \leqslant t+1$, let $S[B]$ be the set of all pairs (δ, M) such that

- δ is a function from B to $\{0, 1, 2\}$, and
- M is a partition of $\delta^{-1}(\{1\})$ into sets of size 2.

Note that for any B as above, we have $|S[B]| \leqslant 2^{\mathcal{O}(t \log t)}$. Then we define the state space Q of $\mathcal{A}_t^{\max}$ to consist of all pairs (B, f), where $B \subseteq \Omega$ is such that $|B| \leqslant t+1$ and f is a function from $S[B]$ to $\mathbb{N} \cup \{-\infty\}$. Note that writing down one state of Q boils down to specifying B and $|S[B]| \leqslant 2^{\mathcal{O}(t \log t)}$ integers.

The idea behind this definition is that we want the following assertion (★) to hold:

> For any boundaried tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of a boundaried graph G, if $q = (B, f)$ is the state assigned by the run of $\mathcal{A}_t^{\max}$ to the root of T, then
> - $B = \partial G$; and
> - for any $(\delta, M) \in S[B]$, $f(\delta, M)$ is the maximum total length of a family $\mathcal{L}$ of vertex-disjoint paths in G such that:
>   1. $M = \{\text{endpoints of } P \colon P \in \mathcal{L}\}$, and
>   2. every vertex $b \in B$ is incident to exactly $\delta(b)$ edges of $\bigcup_{P \in \mathcal{L}} E(P)$; or $-\infty$ if no such $\mathcal{L}$ exists.

Note that if we achieve (★), then for any pair of vertices $u, v \in \partial G$, the length of the longest (u, v)-path is equal to the maximum of $f(\delta, \{u, v\})$, where δ ranges over all functions from B to $\{0, 1, 2\}$ such that $\delta^{-1}(\{1\}) = \{u, v\}$. We output $-\infty$ when there is no (u, v)-path.

It remains to specify the initial mapping and the transition function of $\mathcal{A}_t^{\max}$. For the initial mapping, we simply define it so that (★) is true for one-bag tree decompositions; and it can be computed in time $2^{\mathcal{O}(t \log t)}$ by brute force, for we work with a graph on at most $t+1$ vertices. For the transition function, writing it so that (★) can be proved by a bottom-up induction on the tree decomposition is a simple modification of [11, Lemma 8.2], which we leave the reader. In particular, the evaluation time is $2^{\mathcal{O}(t \log t)}$. ◀

The same reasoning as in the proof of Lemma 17 yields also a tree decomposition automaton $\mathcal{A}_t^{\min}$ that has the same properties, except it provides access to the lengths of the shortest paths between the vertices of ∂G, instead of the longest. The proof can be repeated verbatim except for replacing the word "maximum" with "minimum", and "$-\infty$" with "$+\infty$".

The dynamic treewidth data structures [30, 29] for parameter t maintain a tree decomposition of width $\mathcal{O}(t)$ under the assumption that the treewidth of the graph remains at most t. It may happen though that treewidth exceeds t after edge insertion but the data structure does not report failure. To determine for sure if treewidth is at most t, one can maintain an automaton that tests this property at the cost of increasing the dependence on t of the evaluation time of the automaton (see [30, Section II.A]).

▶ **Lemma 18** ([10])**.** *For every $k \leqslant \ell \in \mathbb{N}$, there is a tree decomposition automaton $\mathcal{B}_{k,\ell}$ such that for any boundaried tree decomposition $\mathcal{T} = (T, \mathsf{bag}, \mathsf{edges})$ of width ℓ of a graph G, if ρ is the run of $\mathcal{B}_{k,\ell}$ on $\mathcal{T}$ and x is the root of T, then from the state $\rho(x)$ one can determine, in time $2^{\mathcal{O}(k\ell^2)}$, whether $\mathrm{tw}(G) \leqslant k$. The evaluation time of $\mathcal{B}_{k,\ell}$ is $2^{\mathcal{O}(k\ell^2)}$.*

With the automata $\mathcal{A}_t^{\max}$, $\mathcal{A}_t^{\min}$ and $\mathcal{B}_{k,\ell}$ in place, we may proceed to the proof of Lemma 7.

Proof of Lemma 7. We may assume that every query for the length of a longest/shortest path in G concerns the same fixed pair of vertices $(u^\star, v^\star)$. To achieve this, we add two fresh vertices $u^\star, v^\star$ to the graph, which normally remain isolated during processing the updates. Upon receiving a query about the length of a longest/shortest (u, v)-path in G, we add edges $u^\star u$ and $vv^\star$, ask for the pair $(u^\star, v^\star)$ instead, and then remove again the edges $u^\star u$ and $vv^\star$. The answer to the query about (u, v) is then the answer to the query about $(u^\star, v^\star)$ decremented by 2.

Now, we set up the dynamic treewidth data structure of Korhonen [29], for the width parameter t. It processes each update in amortized time $2^{\mathcal{O}(t)} \log n$, with an additional factor of $\tau(\mathcal{O}(t))$ for maintaining a run of an automaton $\mathcal{Q}$ with evaluation time $\tau(\ell)$ on width-ℓ tree decomposition. The data structure maintains a rooted tree decomposition $\mathcal{T}$ of G of width $\mathcal{O}(t)$ together with the run of $\mathcal{Q}$ on $\mathcal{T}$. First, we enrich the data structure with the automaton $\mathcal{B}_{k,\ell}$ for $k = t$ and $\ell = \mathcal{O}(t)$ being the bound on the width of the maintained tree decomposition. This incurs a factor of $2^{\mathcal{O}(t^3)} \log n$ per update. Whenever edge insertion is accepted, we check using Lemma 18 if it makes the treewidth grow beyond t. If yes, we revert the insertion.

Next, we enrich the data structure with the automata $\mathcal{A}_t^{\max}$ and $\mathcal{A}_t^{\min}$. Since the evaluation time of $\mathcal{A}_t^{\max}$ and $\mathcal{A}_t^{\min}$ is $2^{\mathcal{O}(t \log t)}$, this data structure handles edge insertions and deletions (and reports if an edge insertion would make the treewidth larger than t) in time $2^{\mathcal{O}(t^3)} \log n$ and maintains the runs of $\mathcal{A}_t^{\max}$ and $\mathcal{A}_t^{\min}$ on $\mathcal{T}$. At the cost of increasing the width by 2, we may assume that $u^\star$ and $v^\star$ are at all times contained in every bag of $\mathcal{T}$, and thus $\mathcal{T}$ is a boundaried tree decomposition of G with boundary $\{u^\star, v^\star\}$. Therefore, by the properties of $\mathcal{A}_t^{\max}$ and $\mathcal{A}_t^{\min}$ provided by Lemma 17, from the states associated with the root of $\mathcal{T}$ we may compute the maximum and the minimum length of an $(u^\star, v^\star)$-path in G. ◀