

Temporal Queries for Dynamic Temporal Forests

Davide Bilò  

Department of Information Engineering, Computer Science, and Mathematics,
University of L'Aquila, Italy

Luciano Gualà  

Department of Enterprise Engineering, University of Rome “Tor Vergata”, Italy

Stefano Leucci  

Department of Information Engineering, Computer Science, and Mathematics,
University of L'Aquila, Italy

Guido Proietti  

Department of Information Engineering, Computer Science, and Mathematics,
University of L'Aquila, Italy

Alessandro Straziota  

Department of Enterprise Engineering, University of Rome “Tor Vergata”, Italy

Abstract

In a temporal forest each edge has an associated set of time labels that specify the time instants in which the edges are available. A temporal path from vertex u to vertex v in the forest is a selection of a label for each edge in the unique path from u to v , assuming it exists, such that the labels selected for any two consecutive edges are non-decreasing.

We design linear-size data structures that maintain a temporal forest of rooted trees under addition and deletion of both edge labels and singleton vertices, insertion of root-to-node edges, and removal of edges with no labels. Such data structures can answer temporal reachability, earliest arrival, and latest departure queries. All queries and updates are handled in polylogarithmic worst-case time. Our results can be adapted to deal with latencies. More precisely, all the worst-case time bounds are asymptotically unaffected when latencies are uniform. For arbitrary latencies, the update time becomes amortized in the incremental case where only label additions and edge/singleton insertions are allowed as well as in the decremental case in which only label deletions and edge/singleton removals are allowed.

To the best of our knowledge, the only previously known data structure supporting temporal reachability queries is due to Brito, Albertini, Casteigts, and Travençolo [Social Network Analysis and Mining, 2021], which can handle general temporal graphs, answers queries in logarithmic time in the worst case, but requires an amortized update time that is quadratic in the number of vertices, up to polylogarithmic factors.

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

Keywords and phrases temporal graphs, temporal reachability, earliest arrival, latest departure, dynamic forests

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2024.11

Related Version *Full Version*: <https://arxiv.org/abs/2409.18750>

1 Introduction

Temporal graphs, a.k.a. time-varying graphs, are graphs in which each edge is only available in specific time instants. These graphs are crucial for modeling and analyzing a variety of complex systems such as social networks, communication networks, transportation systems, and biological networks, where relationships and interactions evolve.



© Davide Bilò, Luciano Gualà, Stefano Leucci, Guido Proietti, and Alessandro Straziota;
licensed under Creative Commons License CC-BY 4.0

35th International Symposium on Algorithms and Computation (ISAAC 2024).

Editors: Julián Mestre and Anthony Wirth; Article No. 11; pp. 11:1–11:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A simple and widely-adopted model of temporal graphs is the one of Kempe, Kleinberg, and Kumar [19] in which a temporal graph is modeled as a graph G where each edge has an assigned integral temporal label representing a time instant in which the edge is available. Temporal paths are time-respecting walks on G , i.e., walks in which any two consecutive traversed edges have non-decreasing labels. This model naturally generalizes to the case in which an edge may be traversed in multiple time instants, i.e., when each edge of G is associated with a set of integer labels, and many other generalizations have also been proposed (see, e.g., [17]).

Due to their generality, many algorithmic problems regarding temporal graphs have been considered in the literature, including the computation of good temporal paths [1, 10, 22] w.r.t. several quality measures, sparsification [7, 8, 11, 12, 19], exploration [13, 14], a temporal version of the classical vertex cover problem [2, 15], and network creation games [6], to mention a few. Perhaps surprisingly, the problem of designing dynamic data structures maintaining information on temporal connectivity has only been addressed quite recently, despite the vast amount of literature for the non-temporal case (see, e.g., [16, 18] and the references therein). In this regard, Brito, Albertini, Casteigts, and Travençolo [9] provide a data structure that can answer *temporal reachability* queries in the *incremental* setting, i.e., under the addition of new edges or of new labels to already existing edges. This data structure has size $O(n^2\tau)$, and an amortized update time of $O(n^2 \log \tau)$, where n is the number of vertices of the temporal graph and τ is its *lifetime*, i.e., the number of distinct labels. Given two vertices u, v and two time instants t, t' , it can report whether there exists a temporal path from u to v that departs from u no earlier than t and arrives in v no later than t' . The worst-case time required by such a query is $O(\log \tau)$ and the corresponding path can be retrieved in $O(\log \tau)$ worst-case time per edge.

In [4] the authors consider the problem of maintaining a directed graph in which edge (u, v) exists if there is a temporal path from u to v in G , and show how to efficiently update the *time-transitive closure* in the restricted *chronological incremental* case, where the inserted labels must be monotonically non decreasing.

Our results. We focus on the case in which the temporal graph of interest is acyclic, i.e., it is a temporal forest F , and we design data structures that support insertions and deletions of edge labels and can answer temporal reachability queries along with the more general *earliest arrival* (EA) and *latest departure* (LD) queries. Given two distinct vertices u, v and time instant t , an EA (resp. LD) query reports the smallest arrival time (reps. largest departure time) among those of all temporal paths from u to v that depart no earlier than t (resp. arrive no later than t).

Our main data structure is dynamic also w.r.t. a second aspect. In addition of being able to add and remove labels from the edges of F , it also supports the addition and deletion of singleton vertices along with *link* and *cut* operations in a temporal forest of rooted trees. The link operation adds a new edge (u, v) with label ℓ in F from the root u of any tree to any other vertex v in a different tree, thus merging the two trees into one. The cut operation deletes the last label of an edge e and removes e from F , thus splitting the tree T that contained e into two trees T_1, T_2 , where the root of T_1 coincides with that of T and T_2 is rooted in the unique endvertex of e in T_2 .

Our data structure has linear size, and can be updated in $O(\log M)$ worst-case time per operation, where M is the total number of (not necessarily distinct) labels in the forest at the time of the operation. EA and LD queries require $O(\log L \cdot \log M)$ worst-case time, where L is the maximum number of labels on a single edge at the time of the operation, and a temporal reachability query can be answered in $O(\log M)$ worst-case time (see Theorem 3).

In the special case of temporal paths with fixed topology (i.e., when only insertions and deletions of edge labels are allowed), we can improve the worst-case time complexity of EA and LD queries to $O(\log M)$, thus shaving a $O(\log L)$ factor (see Theorem 2).

Our data structure can also be adapted to the more general case of temporal forests with *latencies*, in which the labels on a generic edge are pairs (ℓ, d) , with $d \geq 0$, encoding the fact that the edge can be traversed at time ℓ from any endvertex to reach the other endvertex at time $\ell + d$. The data structure retains the same worst-case asymptotic guarantees when latencies are *uniform* (see Corollary 7). For arbitrary latencies, we consider both the *incremental* and the *decremental* scenarios. In the incremental scenario we support the addition of new labels to existing edges, the addition of singleton vertices, and the above link operation, while the decremental scenario only involves label/singleton deletions and cut operations. In both cases, the above $O(\log M)$ upper bound on the update time becomes amortized (see Theorem 6).

2 Preliminaries

A *temporal forest* is an undirected forest $F = (V(F), E(F))$ paired with a function $\lambda : E(F) \rightarrow \mathcal{P}(\mathbb{Z})$ that associates a set $\lambda(e) \subseteq \mathbb{Z}$ of labels to each edge $e \in E(F)$. A *temporal path* π from vertex $u \in V(F)$ to vertex $v \in V(F) \setminus \{u\}$ in F is a sequence of pairs $\langle (e_1, \ell_1), (e_2, \ell_2), \dots, (e_k, \ell_k) \rangle$ such that $\langle e_1, e_2, \dots, e_k \rangle$ is a path from u to v in F , $\ell_i \in \lambda(e_i)$ for all $i = 1, \dots, k$, and $\ell_i \leq \ell_{i+1}$ for all $i = 1, \dots, k-1$. The departure time $\text{departure}(\pi)$ and the arrival time $\text{arrival}(\pi)$ of π are ℓ_1 and ℓ_k , respectively.

Given two distinct vertices $u, v \in V(F)$, and $t_a, t_d \in \mathbb{Z} \cup \{-\infty, +\infty\}$, $\Pi_F(u, v, t_d, t_a)$ is the set of all temporal paths from u to v in F having a departure time of at least t_d and an arrival time of at most t_a . For a time $t \in \mathbb{Z} \cup \{-\infty, +\infty\}$, the *earliest arrival time* of a temporal path from u to v with departure time at least t is denoted by $\text{EA}(u, v, t)$ and is defined as $\min_{\pi \in \Pi(u, v, t, +\infty)} \text{arrival}(\pi)$. If no temporal path from u to v with departure time at least t exists in F , i.e., if $\Pi(u, v, t, +\infty) = \emptyset$, we define $\text{EA}(u, v, t) = +\infty$. A path $\pi \in \Pi(u, v, t, +\infty)$ such that $\text{arrival}(\pi) = \text{EA}(u, v, t)$ is called an *earliest arrival path*.

Similarly, we denote the *latest departure time* of a temporal path from u to v having an arrival time of at most $t \in \mathbb{Z} \cup \{-\infty, +\infty\}$ as $\text{LD}(u, v, t) = \max_{\pi \in \Pi(u, v, -\infty, t)} \text{departure}(\pi)$. If $\Pi(u, v, -\infty, t) = \emptyset$, then we let $\text{LD}(u, v, t) = -\infty$. A path $\pi \in \Pi(u, v, t, +\infty)$ such that $\text{arrival}(\pi) = \text{LD}(u, v, t)$ is called a *latest departure path*. As an edge case, we define $\text{EA}(u, u, t) = \text{LD}(u, u, t) = t$ for all $u \in V(F)$.

We design a data structure for maintaining information on a dynamic temporal forest of rooted trees. The data structure must efficiently answer to the following queries:

Earliest arrival time (EA query): Given two vertices $u, v \in V(F)$ with $u \neq v$, and a time $t \in \mathbb{Z} \cup \{-\infty, +\infty\}$, report $\text{EA}(u, v, t)$;

Latest departure time (LD query): Given two vertices $u, v \in V(F)$ with $u \neq v$, and a time $t \in \mathbb{Z} \cup \{-\infty, +\infty\}$, report $\text{LD}(u, v, t)$;

Temporal reachability: Given two vertices $u, v \in V(F)$ with $u \neq v$, and two times $t_a, t_d \in \mathbb{Z} \cup \{-\infty, +\infty\}$ with $t_a \leq t_d$, report whether there exists a temporal path π from u to v in F with $\text{departure}(\pi) \geq t_a$ and $\text{arrival}(\pi) \leq t_d$.

Clearly, a data structure that supports EA queries or LD queries also supports temporal reachability queries with the same worst-case query time. The update operations we consider are the following:

11:4 Temporal Queries for Dynamic Temporal Forests

Label addition: Given an edge $e \in E(F)$ and a value $\ell \in \mathbb{Z} \setminus \lambda(e)$, add ℓ to $\lambda(e)$;

Label deletion: Given an edge $e \in E(F)$ with $|\lambda(e)| \geq 2$ and some $\ell \in \lambda(e)$, delete ℓ from $\lambda(e)$;

Link: Given two vertices u, v where u is the root of some tree T in F and v is some vertex of a tree T' of F with $T' \neq T$, and a label $\ell \in \mathbb{Z}$, add the edge (u, v) to F and set $\lambda((u, v)) = \{\ell\}$. The new tree resulting from merging T with T' retains the root of T' ;

Cut: Given an edge $e \in E(F)$ with $|\lambda(e)| = 1$, remove the edge e from the tree T of F containing e , thus splitting T into two new trees T_1, T_2 where T_1 retains the root of T and T_2 is rooted in the unique endvertex of e in T_2 ;

Singleton addition/deletion: Add a new (resp. remove an existing) singleton vertex to (resp. from) F .

If u, v, w are vertices of F such that w lies on the unique path between u and v in F we have that $EA(u, v, t) = EA(u, w, EA(w, v, t))$ and $LD(u, v, t) = LD(u, w, LD(w, v, t))$.

Let $-F$ be the temporal forest such that $V(F) = V(-F)$, $E(F) = E(-F)$, and for every $e \in E(F)$, the set of labels of e in $-F$ is $\{-\ell \mid \ell \in \lambda(e)\}$. One can observe that:

► **Lemma 1.** *The value of $LD(u, v, t)$ in F coincides with $-EA(v, u, -t)$ in $-F$.*

Finally, given a subset X of a universe possessing a strict total order, and an element x of the universe, we define the *successor* $\text{succ}(x, X)$ (resp. *predecessor* $\text{pred}(x, X)$) of x w.r.t. X as the minimum element $y \in X$ s.t. $y \geq x$ (resp. the maximum element $y \in X$ s.t. $y \leq x$). If such an element y does not exist, $\text{succ}(x, X) = +\infty$. Similarly, the *strict successor* (resp. *strict predecessor*) is defined as $\text{succ}(x, X \setminus \{x\})$ (resp. $\text{pred}(x, X \setminus \{x\})$).

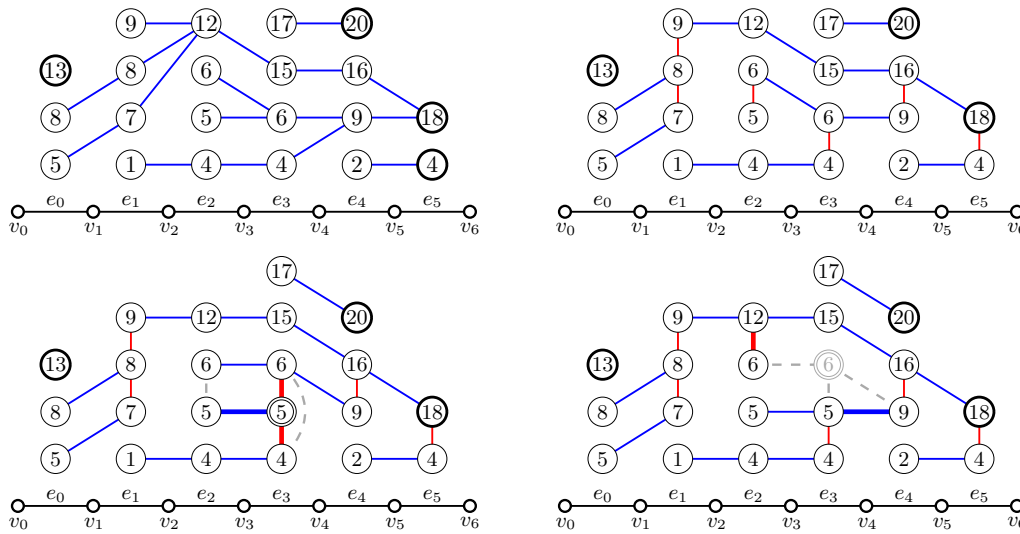
3 Warm up: a data structure for temporal paths

As a warm up, in this section we consider the simpler case in which the forest F is actually a fixed path, and the only supported operations are label additions and deletions to/from the edges of F . We use this section to introduce some of the ideas of our construction, which we generalize to the case of dynamic temporal forests in Section 4 that also contains the correctness proofs. More precisely, we establish the following result:

► **Theorem 2.** *Given a temporal path with n vertices, it is possible to build in $O(n + M \log L)$ time, a data structure of linear size supporting EA and LD queries under label insertions and deletions in $O(\log M)$ worst-case time per operation, where L is the maximum number of labels on the same temporal edge, and M is the total number of (not necessarily distinct) labels in the path at the time of the operation.*

Fix an arbitrary endvertex v_0 of the path F , and let v_i be the unique vertex at hop-distance i from v_0 in F . We think of F as a tree rooted in v_{n-1} . With this interpretation, the main technical difficulty lies in designing a data structure capable of answering *upward* EA queries, i.e., EA queries from some vertex v_i to some vertex v_j with $j > i$. The cases of *downward* EA queries can be managed by a similar data structure rooted in v_0 . LD queries and reachability queries can be recast as EA queries as already discussed in the preliminaries and in Lemma 1.

The naive solution to represent all possible upward earliest arrival paths in F is that of building a forest of rooted trees similar to the one shown on the top-left of Figure 1. This forest represents each label ℓ of each edge $e = (v_i, v_{i+1})$ in F as a node whose parent corresponds to the first label of (v_{i+1}, v_{i+2}) that is larger than or equal to ℓ (if any). Intuitively, moving upwards in F along an earliest arrival path corresponds to moving upwards in a tree of such



■ **Figure 1** Top left: a forest representing all possible upward earliest arrival paths. Top right: the corresponding weighted forest \mathcal{F} . Red edges have weight 0, while blue ones have weight 1. Here, a label $\ell \in \lambda(e_i)$ is shown above edge e_i and models the node (ℓ, i) . Bottom left and bottom right: the forests \mathcal{F} obtained after adding label 5 to $\lambda(e_3)$ and then deleting label 6 from $\lambda(e_3)$. New edges are shown in bold while removed ones are dashed.

a forest. Unfortunately, explicitly maintaining this forest would be costly since even a single label insertion/deletion could cause a large number of edges to be rewired. We circumvent this problem by maintaining a different edge-weighted forest $\mathcal{F}(\lambda)$ that still represents all possible upward earliest arrival paths while guaranteeing that each node has constant degree.

Let e_i be the edge (v_i, v_{i+1}) . Our forest $\mathcal{F}(\lambda)$ contains a *node* (ℓ, i) for each edge e_i in F , and for each $\ell \in \lambda(e_i)$. To aid readability we use the term *vertex* to refer to vertices in F , and the term *node* for those in $\mathcal{F}(\lambda)$. To describe the edges of $\mathcal{F}(\lambda)$, we first define a partial function $\sigma_i(\ell, \lambda)$ that maps the labels $\ell \in \lambda(e_i)$ to a “successor” node, which is either the one corresponding to the smallest label strictly larger than ℓ on the same edge e_i , or the one corresponding to the successor of ℓ among the labels of edge e_{i+1} . More precisely, given $i \in \{0, \dots, n-2\}$ and $\ell \in \lambda(e_i)$, we consider $\ell^+ = \text{succ}(\ell + 1, \lambda(e_i))$. For $i = n-2$, $\sigma_i(\ell, \lambda)$ is defined as (e_i, ℓ^+) if $\ell^+ < +\infty$, and is undefined if $\ell^+ = +\infty$. For $i \in \{0, \dots, n-3\}$, we compare ℓ^+ with $\ell' = \text{succ}(\ell, \lambda(e_{i+1}))$. If $\ell^+ = \ell' = +\infty$, then $\sigma_i(\ell, \lambda)$ is undefined. Otherwise $\sigma_i(\ell, \lambda) = \begin{cases} (\ell^+, i) & \text{if } \ell^+ \leq \ell'; \\ (\ell', i+1) & \text{if } \ell' < \ell^+. \end{cases}$

All the nodes (ℓ, i) for which $\sigma_i(\ell, \lambda)$ is undefined are the roots of their respective trees in $\mathcal{F}(\lambda)$. Whenever λ is clear from context we write \mathcal{F} in place of $\mathcal{F}(\lambda)$ and $\sigma_i(\ell)$ in place of $\sigma_i(\ell, \lambda)$. See Figure 1 (top-right) for an example.

The weights of all edges of the form $((\cdot, i), (\cdot, i+1))$ are set to 1, while those of the remaining edges, of the form $((\cdot, i), (\cdot, i))$, are set to 0. Our definition of σ_i ensures that different children of the same node v in \mathcal{F} must be linked to v using edges of different weights. As a consequence, each vertex has at most 2 children in \mathcal{F} .

Our data structure stores:

- A *top tree* [3] representing \mathcal{F} . A top-tree is a data structure capable of maintaining an edge-weighted forest under insertion of new nodes, deletion of singleton nodes, and under link and cut operations, i.e., addition and deletion of edges. Moreover, it supports

weighted level ancestor queries: given a vertex v and $w \in \mathbb{N}$, it reports the w -th weighted level ancestor $\text{LA}(v, w)$ of v in \mathcal{F} , i.e., the deepest ancestor of v at distance at least w from v (if any). Each of the above operations requires $O(\log \eta)$ worst-case time, where η is the number of nodes in the forest. A top tree with η nodes can be built in $O(\eta)$ time. Whenever a node/edge is added/removed from \mathcal{F} , we perform the corresponding update operation on the backing top tree.

- A dictionary D_i for each edge e_i that supports insertions, deletions, and predecessor/successor queries in $O(\log \eta)$ worst-case time per operation, where η is the number of keys in D_i . The dictionary can be built in $O(\eta \log \eta)$ time. The keys in D_i are the labels in $\lambda(e_i)$ and each label ℓ is associated with a pointer to vertex (ℓ, i) in \mathcal{F} . Such a dictionary can be implemented using any dynamic balanced binary search tree.

Answering EA queries. To report $\text{EA}(v_i, v_j, t)$ with $j > i$, we first find $\ell = \text{succ}(t, \lambda(e_i))$ using D_i , then we query \mathcal{F} for the $(j - i - 1)$ -th weighted level ancestor (e_{j-1}, ℓ^*) of (e_i, ℓ) . If such ancestor exist, we answer with ℓ^* . Otherwise we answer with $+\infty$. Finding ℓ requires time $O(\log L)$ in the worst case, while querying \mathcal{F} can be done in worst-case $O(\log M)$ time. Hence the overall worst-case time required to answer the query is $O(\log M)$.

An auxiliary procedure. A label insertion causes the addition of a new node into \mathcal{F} , while a label deletion causes the deletion of the corresponding node from \mathcal{F} . This may result in some nodes in \mathcal{F} that have an incorrect parent (or lack of thereof) from the one that would be expected according to the construction discussed above.

To address this problem, we define an auxiliary procedure $\text{FixParent}(\ell, i)$ that will be helpful in restoring the correct state of \mathcal{F} . Such procedure takes the index i of some edge e_i in \mathcal{F} and a label $\ell \in \lambda(e_i)$, and ensures that the edge from node (ℓ, i) to its parent in \mathcal{F} (if any) is properly set (or unset) by only considering D_i and D_{i+1} .

To implement $\text{FixParent}(\ell, i)$ we first *cut* the current edge from (ℓ, i) to its parent in \mathcal{F} (if any). Then, we compute $\sigma_i(\ell)$ in $O(\log L)$ time by searching for the needed successors in D_i and D_{i+1} , according to the definition of σ_i . Finally, we *link* vertex (ℓ, i) with $\sigma_i(\ell)$, if any, in \mathcal{F} . Since link and cut operations also require time $O(\log M)$, the overall worst-case time spent by FixParent is $O(\log M)$.

Label addition. To add label ℓ on edge e_i , we first insert node (ℓ, e_i) into \mathcal{F} , and ℓ into D_i . Then, if $i \geq 1$ and $\ell' = \text{pred}(\ell, \lambda(e_{i-1}))$ exists, we perform $\text{FixParent}(\ell, i - 1)$. Next, we perform $\text{FixParent}(\ell, i)$. Finally, if $\ell^- = \text{pred}(\ell - 1, \lambda(e_i))$ exists, we perform $\text{FixParent}(\ell^-, i)$. See Figure 1 for an example.

The overall worst-case time required is $O(\log M)$ since we only perform a constant number of predecessor/successor lookups, node insertions into \mathcal{F} , and calls to FixParent .

Label deletion. To remove label ℓ from edge e_i , we first delete ℓ from D_i and remove node (e_i, ℓ) , along with all its incident edges, from \mathcal{F} . Then, if $i \geq 1$ and $\ell' = \text{pred}(\ell, \lambda(e_{i-1}))$ exists, we perform $\text{FixParent}(\ell', i - 1)$. Finally, if $\ell^- = \text{pred}(\ell - 1, \lambda(e_i))$ exists, we perform $\text{FixParent}(\ell^-, i)$. The overall worst-case time required is $O(\log M)$ since we only perform a constant number of predecessor/successor lookups, edge/node deletions from \mathcal{F} , and calls to FixParent .

4 Our data structure for temporal forests

In this section we discuss how our data structure for temporal paths with static topology can be generalized to temporal forests of rooted trees while also supporting link and cut operations as well as insertions and deletions of singleton vertices. More precisely, we prove the following result.

► **Theorem 3.** *Given a temporal forest of rooted trees, it is possible to build, in $O(n + M \log L)$ time, a data structure of linear size, where n is the number of vertices, L is the maximum number of labels on the same temporal edge, and M is the total number of (not necessarily distinct) labels in the forest at the time of the operation. The data structure supports label additions, label deletions, link, and cut operations, and addition/deletion of singleton vertices in $O(\log M)$ worst-case time per operation, EA and LD queries in $O(\log L \cdot \log M)$ worst-case time per operation, and reachability queries in $O(\log M)$ worst-case time per operation.*

We observe that it is easy to extend Theorem 2 to *fixed* temporal forests where one only needs to support label additions and deletions and aims for a query time of $O(\log n \cdot \log M)$, where n is the number of vertices in F . Indeed, it suffices to construct the data structure for temporal paths on each path of a heavy-light decomposition [21] of (each tree of) F . Details are given in the full version of the paper.

This section is organized as follows: we first discuss a data structure that only supports updates are label additions and deletions, and then we show how to handle link and cut operations as well as insertions and deletions of singleton vertices.

4.1 A data structure supporting only label additions and deletions

While the natural generalization of our construction for temporal paths of Section 3 to the case of trees would already provide a data structure supporting fast upward EA queries, such a solution runs into a similar problem as the one discussed for the naive approach. Indeed, in a tree of degree Δ a node of \mathcal{F} may have $\Omega(\Delta)$ children, and its removal may cause the rewiring of $\Omega(\Delta)$ edges.

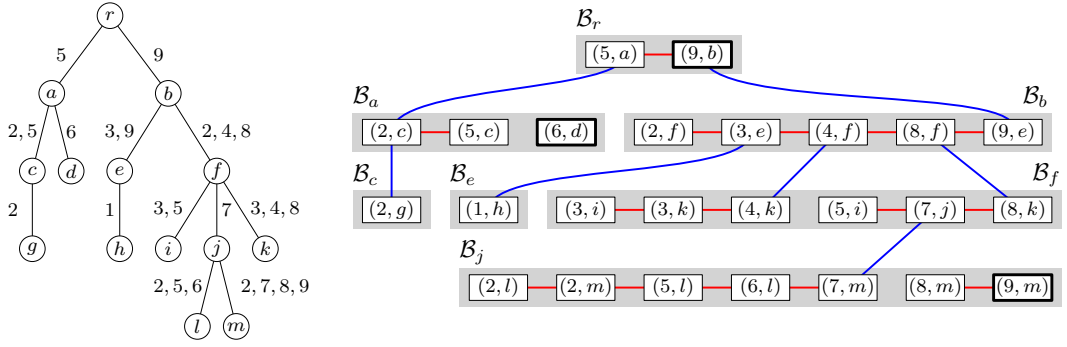
Our construction avoids this problem by further grouping the (nodes in \mathcal{F} corresponding to the) edges from sibling vertices to their common parent v in F into a *block* \mathcal{B}_v (see Figure 2).

For a non-root vertex $v \in V(F)$, we denote by $p(v)$ the parent of v in F , and by e_v the edge $(v, p(v))$. The forest $\mathcal{F}(\lambda)$ contains a *node* (ℓ, v) for each non-root vertex $v \in V(F)$ and for each $\ell \in \lambda(e_v)$. Moreover, for every non-leaf vertex $v \in V(F)$, we define the *block of v* as the set $\mathcal{B}_v(\lambda)$ containing all nodes (ℓ, u) where u is a child of v in F . We fix an arbitrary strict total ordering of the vertices of $V(F)$ and we think of the nodes of \mathcal{F} as being ordered w.r.t. the order relation that compares nodes lexicographically, i.e., (ℓ, u) precedes (ℓ', v) if either $\ell < \ell'$, or if $\ell = \ell'$ and u precedes v in the chosen ordering of the vertices.

Similarly to the case of temporal paths, the edges of $\mathcal{F}(\lambda)$ are defined by making use of a partial function $\sigma_u(\ell, \lambda)$ that associates the nodes of \mathcal{F} to their parents. More precisely, given a non-root vertex $u \in V(F)$ and $\ell \in \lambda(e_u)$, we let $v = p(u)$ so that $(\ell, u) \in \mathcal{B}_v$. If v is a root in F , $\sigma_u(\ell, \lambda)$ is defined as the strict successor of (ℓ, u) in \mathcal{B}_v , if it exists, otherwise $\sigma_u(\ell, \lambda)$ is undefined.

To define $\sigma_u(\ell, \lambda)$ when v is not a root of F , let $\ell' = \text{succ}(\ell, \lambda(e_v))$. We distinguish two cases depending on whether the strict successor (ℓ^+, u^+) of (ℓ, u) in \mathcal{B}_v exists. If (ℓ^+, u^+)

exists we define $\sigma_u(\ell, \lambda) = \begin{cases} (\ell^+, u^+) & \text{if } \ell^+ \leq \ell'; \\ (\ell', v) & \text{if } \ell^+ > \ell'. \end{cases}$



■ **Figure 2** Left: a sample rooted temporal tree. Right: a representation of the forest \mathcal{F} maintained by the data structure for temporal forests of Section 4.

If (ℓ^+, u^+) does not exist, then $\sigma_u(\ell, \lambda) = (\ell', v)$ when $\ell' < +\infty$ and $\sigma_u(\ell, \lambda)$ is undefined when $\ell' = +\infty$. The nodes (ℓ, u) such that $\sigma_u(\ell, \lambda)$ is undefined are the roots of the corresponding trees in $\mathcal{F}(\lambda)$. Whenever λ is clear from context, we write \mathcal{F} in place of $\mathcal{F}(\lambda)$, \mathcal{B}_v in place of $\mathcal{B}_v(\lambda)$, and $\sigma_u(\ell)$ in place of $\sigma_u(\ell, \lambda)$.

We say that an edge in \mathcal{F} between nodes in the same block is *red*, while inter-block edges are *blue*. We assign weight 0 to red edges, and weight 1 to blue edges. See Figure 2 for an example.

Our data structure stores:

- A top tree representing \mathcal{F} . In addition to the operations discussed in Section 3, a top tree also supports lowest common ancestor (LCA) queries, i.e., given two nodes u, v of \mathcal{F} , it either reports that u and v belong to different trees in \mathcal{F} , or it answers with the deepest vertex w in the unique tree T containing both u and v such that w is an ancestor of both u and v . An LCA query requires $O(\log \eta)$ worst-case time, where η is the number of nodes in \mathcal{F} .
- A dictionary D_v for each non-root vertex $v \in V(F)$ that stores a key for each label in $\lambda(e_v)$ and that supports insertions, deletions and predecessor/successor queries. Each key ℓ stores, as satellite data, a pointer to node (ℓ, v) in \mathcal{F} .
- A dictionary for each block \mathcal{B}_v , where v is a non-leaf vertex in F . Such a dictionary stores all elements in \mathcal{B}_v and supports insertions, deletions and predecessor/successor queries w.r.t. our order relation on the nodes.
- The depth $d(v)$ of each vertex v in the (unique) rooted tree T containing v in F , i.e., the hop distance between v and the root of T .

Adapting FixParent. $\text{FixParent}(\ell, v)$ takes a non-root vertex $v \in V(F)$ and a label $\ell \in \lambda(e_v)$, and ensures that the edge from node (ℓ, v) to its parent in \mathcal{F} , if any, is properly set.

To implement $\text{FixParent}(\ell, v)$ we first *cut* the current edge from (ℓ, v) to its parent in \mathcal{F} (if any). Then we compute $\sigma_v(\ell)$ in $O(\log L)$ worst-case time by searching for the needed successors in $\mathcal{B}_{p(v)}$ and $D_{p(v)}$ (if $p(v)$ exists), according to the definition of σ_v . Notice that $\sigma_v(\ell)$ might be undefined. Finally, if $\sigma_v(\ell)$ is defined, we *link* vertex (ℓ, v) with $\sigma_v(\ell)$. Since link and cut operations require time $O(\log M)$, the overall worst-case time spent by FixParent is $O(\log M)$.

Label addition. To add label ℓ on edge e_v , we first insert node (ℓ, v) into \mathcal{F} , ℓ into D_v , and (ℓ, v) into $\mathcal{B}_{p(v)}$. Then, if v is not a leaf in F and $(\ell', u) = \text{pred}((\ell, +\infty), \mathcal{B}_v)$ exists, we perform $\text{FixParent}(\ell', u)$.¹ Next, we perform $\text{FixParent}(\ell, v)$. Finally, if the strict predecessor (ℓ^-, v^-) of (ℓ, v) in $\mathcal{B}_{p(v)}$ exists, we perform $\text{FixParent}(\ell^-, v^-)$.²

Label deletion. To delete label ℓ from edge e_v , we remove the node (ℓ, v) from \mathcal{F} along with all its incident edges, we delete ℓ from D_v , and we delete (ℓ, v) from $\mathcal{B}_{p(v)}$. Then, if v is not a leaf in F and $(\ell', u) = \text{pred}((\ell, +\infty), \mathcal{B}_v)$ exists, we perform $\text{FixParent}(\ell', u)$. Finally, if the strict predecessor (ℓ^-, v^-) of (ℓ, v) in $\mathcal{B}_{p(v)}$ exists, we perform $\text{FixParent}(\ell^-, v^-)$.

To prove the correctness of our label addition deletion procedures we need the following lemma, whose proof is given in the full version of the paper, which captures the changes to \mathcal{F} following an update:

► **Lemma 4.** *Consider a forest F , two functions $\lambda', \lambda : E(F) \rightarrow \mathcal{P}(\mathbb{Z})$ and an edge $e_v \in E(F)$ such that, for each $e' \in E(F) \setminus \{e_v\}$, $\lambda(e') = \lambda'(e')$ and $\lambda(e_v) = \lambda'(e_v) \setminus \{\ell\}$ with $\ell \in \lambda'(e_v)$. Let $\mathcal{F}' = \mathcal{F}(\lambda')$ and $\mathcal{F} = \mathcal{F}(\lambda)$. Let U be the set containing (ℓ, v) , $(\ell', u) = \text{pred}((\ell, +\infty), \mathcal{B}_v(\lambda))$ (if it exists), and the strict predecessor (ℓ^-, v^-) of (ℓ, v) in $\mathcal{B}_{p(v)}(\lambda)$ (if it exists).*

We have that $V(\mathcal{F}) = V(\mathcal{F}') \setminus \{(\ell, v)\}$ and that all nodes in $V(\mathcal{F}) \setminus U$ have the same parent (or lack of thereof) in both F and \mathcal{F}' .

Then, the correctness of the label addition procedure follows from Lemma 4 with λ (resp. λ') chosen as the function that maps each edge to its labels before (resp. after) the addition, once one observe that FixParent is invoked on all vertices of the set U defined by the lemma. Symmetrically, the correctness of the label deletion procedure follows from Lemma 4 when the roles of λ and λ' are reversed.

The overall worst-case time required per update is $O(\log M)$ since we only perform a constant number of (strict) predecessor lookups, edge/node deletions from \mathcal{F} , and calls to FixParent . Observe that Lemma 4 implies that, when node (ℓ, v) is deleted, its degree in \mathcal{F} is constant (since each child of (ℓ, v) changes its parent following the deletion).

Answering upward EA queries and downward LD queries. We first discuss how to report $\text{EA}(u, v, t)$ when u is a proper descendent of v in F .³ To do so, we start by finding $\ell = \text{succ}(t, \lambda(e_u))$. If $\ell = +\infty$, we answer with $+\infty$, otherwise we query \mathcal{F} for the $(d_v - d_u - 1)$ -th weighted level ancestor of (ℓ, u) . If such an ancestor (ℓ^*, w) exists we answer with ℓ^* , otherwise we answer with $+\infty$. This requires $O(\log M)$ worst-case time since it only involves a successor lookup in D_u and a weighted level ancestor query on the top tree representing \mathcal{F} .

The correctness of our query procedure stems by a structural property of \mathcal{F} captured by the following lemma, whose proof is given in the full version of the paper.

► **Lemma 5.** *Let $u, v \in V(F)$ where v is a proper ancestor of u , let $t \in \mathbb{Z} \cup \{-\infty\}$, and define $\ell = \text{succ}(t, \lambda(e_u))$. If (i) $\ell = +\infty$ or (ii) $\ell < +\infty$ and $\text{LA}((\ell, u), d_v - d_u - 1)$ does not exist, then $\text{EA}(u, v, t) = +\infty$. Otherwise, $\text{LA}((\ell, u), d_v - d_u - 1) = (\ell^*, w)$ where w is the unique ancestor of u such that $p(w) = v$ and $\ell^* = \text{EA}(u, v, t)$.*

¹ Notice that the set of nodes in B_v does not change following the label addition.

² Notice that the insertion of (ℓ, v) into $B_{p(v)}$ does not affect the value (nor the existence) of the strict predecessor of (ℓ, v) in $B_{p(v)}$.

³ Since the topology of F does not change, checking whether u is a proper descendent of v can be done in constant time after $O(n)$ -time preprocessing, where n is the number of vertices in F .

11:10 Temporal Queries for Dynamic Temporal Forests

To answer downward LD queries, we maintain a mirrored data structure for $-F$, in which each original label ℓ is replaced with $-\ell$. Lemma 1 allows us to answer downward LD queries in $O(\log M)$ worst-case time by performing upward EA queries on the data structure for $-F$.

Answering upward LD queries and downward EA queries. To report $\text{LD}(u, v, t)$ when u is a proper descendent of v in F , we binary search for the largest label $\ell^* \in \lambda(e_u)$ such that $\text{EA}(u, v, \ell^*) \leq t$. Then, we report ℓ^* . The correctness of our query immediately follows from the fact that the values $\text{EA}(u, v, \ell)$ are monotonically non decreasing w.r.t. ℓ . This requires a worst-case time of $O(\log L \cdot \log M)$. To report $\text{EA}(w, v, t)$ when v is a proper descendant of w in F , we compute $\text{LD}(v, w, -t)$ on the data structure for $-F$ and answer with $-\text{LD}(v, w, -t)$. The worst-case time required is $O(\log L \cdot \log M)$.

Answering general EA and LD queries. To answer a general $\text{EA}(u, v, t)$ query we compute the lowest common ancestor w of u and v in F in constant time using the data structure in [5], and we return $\text{EA}(w, v, \text{EA}(u, v, t))$ where $\text{EA}(w, v, \cdot)$ is a downward query, and $\text{EA}(u, v, \cdot)$ is an upward query. General LD queries can similarly be answered by using the data structure for $-F$. The worst-case time required to answer such queries is $O(\log L \cdot \log M)$.

Answering temporal reachability queries. To report whether a vertex v is reachable from a vertex u using a temporal path in F that departs no earlier than t_d and arrives no later than t_a , we compute the lowest common ancestors w of u and v in F in constant time and we answer affirmatively iff $\text{EA}(u, w, t_d) \leq \text{LD}(w, v, t_a)$. The overall worst-case time required is $O(\log M)$ since we only need to perform an upward EA query and a downward LD query.

4.2 Supporting link and cut operations

To support general *link* and *cut* operations we additionally maintain F using a top tree.⁴ Each time that an edge is added/removed from F , we perform the corresponding operation on the backing top tree. Moreover, we no longer explicitly maintain the depths d_v , but rather we query the top tree of F every time any such depth is needed.⁵ Similarly, all LCA queries on F are now performed using the backing top tree.

Insertions and deletion of singleton vertices are straightforward, therefore we only discuss how to handle link and cut operations.

Link operations. To implement a link operation where u is the root of some tree T in F and v is some vertex of a tree T' of F with $T' \neq T$, and a label $\ell \in \mathbb{Z}$, we first *link* u and v in F by adding edge (u, v) , so that the parent of u becomes v , and we add label ℓ to (u, v) as explained in Section 4.1. The worst-case time required is $O(\log M)$.

Cut operations. To cut an edge (u, v) of F with a single label $\ell \in \lambda((u, v))$, where u is a child of v w.l.o.g., we first remove the only label ℓ (corresponding to the only key in D_u) from (u, v) as explained in Section 4.1. Then, we *cut* (u, v) from F , thus creating a new tree rooted in u . The worst-case time required is $O(\log M)$.

⁴ Some technical care is needed to obtain the stated bounds, which only depend on M , when $M = o(n)$. This can be achieved by not actually storing singleton vertices in the top tree, so that the number of vertices in the top tree is always $O(M)$. The operations of our data structure are easy to adapt to handle this edge case. E.g., whenever a link operation on F involves some singleton vertex v , we can add v to the top tree immediately before performing the link.

⁵ Indeed, the top tree can report the root of the tree in F that contains v and the hop-distance between any two nodes in F in logarithmic time.

5 Our data structure for temporal forests with latencies

Our model of temporal graphs can be generalized by introducing *latencies*. In *temporal graphs with latencies*, each edge e is associated with a collection of pairs (ℓ, d) encoding that edge e can be traversed at the *departure time* ℓ starting from one of its endvertices in order to reach the other endvertex at time $\ell + d$. The value d is called a *latency*.

Equivalently, each (activation time, latency) pair (ℓ, d) can be expressed as (ℓ, α) , where $\alpha = \ell + d$ is the *arrival time*. These two representations are clearly equivalent, but the latter one results in a lighter notation for our purposes. Therefore, in the rest of the section we will adopt the (departure time, arrival time) convention and we accordingly define $\lambda(e)$ as the set of all (departure time, arrival time) pairs (ℓ, α) associated with edge e .⁶

A *temporal path* π from vertex $u \in V(F)$ to vertex $v \in V(F) \setminus \{u\}$ in F is a sequence of triples $\langle (e_1, \ell_1, \alpha_1), (e_2, \ell_2, \alpha_2), \dots, (e_k, \ell_k, \alpha_k) \rangle$ such that $\langle e_1, e_2, \dots, e_k \rangle$ is a path from u to v in F , $(\ell_i, \alpha_i) \in \lambda(e_i)$ for all $i = 1, \dots, k$, and $\alpha_i \leq \ell_{i+1}$ for all $i = 1, \dots, k-1$. The departure time of π is ℓ_1 and its arrival time is α_k . The notions of earliest arrival paths, latest departure paths, reachability, and the corresponding queries extend naturally.

In this section we argue that our data structure for temporal forests can be adapted to additionally support latencies. This comes at the cost of turning our worst-case bounds on the time complexities of the link and cut operations into amortized bounds that hold for the incremental case, in which only link operations are allowed, and in the decremental case, in which only cut operations are allowed.⁷

► **Theorem 6.** *Given a temporal forest of rooted trees with latencies, it is possible to build a data structure of linear size that supports EA and LD queries in $O(\log L \cdot \log M)$ worst-case time per operation, and reachability queries in $O(\log M)$ worst-case time per operation, where L is the maximum number of labels on the same temporal edge, and M is the total number of (not necessarily distinct) labels in the forest at the time of the operation. In the incremental case, the data structure also supports insertions of singleton vertices, label insertions, and link operations in amortized time $O(\log M)$. In the decremental case, the data structure also supports deletions of singleton vertices, label deletions, and cut operations in amortized $O(\log M)$ time and amortized building time of $O(M \log L)$.⁸*

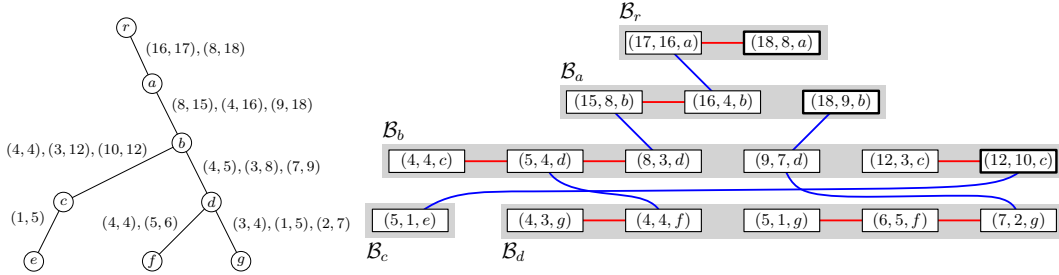
As before, we consider a temporal forest with latencies F containing rooted temporal trees and, for a non-root vertex v , we define e_v as the edge from v to its parent $p(v)$ in the unique tree of F containing v . We define $\mathcal{F}(\lambda)$ as the forest containing a *node* (α, ℓ, v) for each non-root vertex v and for each $(\ell, \alpha) \in \lambda(e_v)$. For each non-leaf vertex $v \in V(F)$, we also define the *block of v* as the set $\mathcal{B}_v(\lambda)$ containing all nodes (α, ℓ, u) such that $p(u) = v$. We fix an arbitrary order of the vertices of $V(F)$ and we think of the nodes of \mathcal{F} as being ordered w.r.t. the order relation that compares nodes lexicographically, i.e., (α, ℓ, u) precedes (α', ℓ', v) if (i) $\alpha < \alpha'$, or (ii) $\alpha = \alpha'$ and $\ell < \ell'$, or (iii) $\alpha = \alpha'$, $\ell = \ell'$, and u precedes v in the chosen ordering of the vertices. Informally, we first order the nodes in a block by arrival time, then by departure time, and finally by their corresponding vertex in F .

⁶ The special case in which all labels (ℓ, α) have $\alpha = \ell$ corresponds to the model used in the previous sections.

⁷ Our data structure can still handle arbitrary sequences of link and cut operations, but the amortized bounds do not hold for this case.

⁸ In the decremental case, we can naturally assume that the initial number of vertices n in F satisfies $n = \Omega(M)$.

11:12 Temporal Queries for Dynamic Temporal Forests



■ **Figure 3** Left: a sample rooted temporal tree with latencies. Right: a representation of the forest \mathcal{F} maintained by the data structure for temporal forests with latencies of Section 5.

We now define the analogue of the function σ for temporal forests with latencies. More precisely, given a non root vertex u in F with parent $v = p(u)$ and $(\ell, \alpha) \in \lambda(e_u)$, we let (α^+, ℓ^+, u^+) be the *strict successor* of (α, ℓ, u) in $\mathcal{B}_v(\lambda)$, if any. If v is a root of a tree in F , then $\sigma_u(\alpha, \ell, \lambda) = (t^+, \ell^+, u^+)$ if (t^+, ℓ^+, u^+) exists, otherwise $\sigma_u(\alpha, \ell, \lambda)$ is undefined. When v is not a root of any tree in F , we define $\text{NextHop}(\alpha, \ell, u)$ as the node (α', ℓ', v) such that $(\ell', \alpha') \in \lambda(e_v)$, $\ell' \geq \alpha$, and α' is minimized, breaking ties in favor of labels with the largest departure time (such a node might not exist).

We observe that if $\text{NextHop}(\alpha^+, \ell^+, u^+)$ exists, if it exists, either coincides with (α', ℓ', v) or it follows (α', ℓ', v) in $\mathcal{B}_{p(v)}$ w.r.t. our ordering. Thus, if both (α^+, ℓ^+, u^+) and (α', ℓ', v)

exist, we define:
$$\sigma_u(\alpha, \ell, \lambda) = \begin{cases} (\alpha^+, \ell^+, u^+) & \text{if } \alpha^+ \leq \ell'; \\ (\alpha', \ell', v) & \text{if } \ell' < \alpha^+. \end{cases}$$

Here the condition $\alpha^+ \leq \ell'$ is equivalent to the following: $\text{NextHop}(\alpha^+, \ell^+, u^+)$ exists and coincides with (α', ℓ', v) . If neither (α^+, ℓ^+, u^+) nor (α', ℓ', v) exist, then $\sigma_u(\alpha, \ell, \lambda)$ is undefined and thus (α, ℓ, λ) is a root. Otherwise, $\sigma_u(\alpha, \ell, \lambda)$ is defined as the only node that exists among (α^+, ℓ^+, u^+) and (α', ℓ', v) . As usual, we drop the parameter λ from $\mathcal{F}(\lambda)$, $\sigma_u(\alpha, \ell, \lambda)$, and $\mathcal{B}(\lambda)$ whenever λ is clear from context.

Our data structure is analogous to that of Section 4, with the following exceptions:

- each dictionary D_v storing the pairs in $\lambda(e_v)$ now supports queries of the following form: given a range of values of interest for ℓ (resp. α), return the minimum/maximum value of α (resp. ℓ) w.r.t. all labels $(\ell, \alpha) \in \lambda(e_v)$ such that ℓ (resp. α) is in the sought range (notice that value might not exist). This can be done in time $O(\log |\lambda(e_v)|)$ using, e.g., priority search trees [20], which require space $O(|\lambda(e_v)|)$.
- for each non-leaf node v we say that a node $(\alpha, \ell, u) \in \mathcal{B}_v$ is a *head* of \mathcal{B}_v if (α, ℓ, u) either has no parent in \mathcal{F} or it is linked to its parent with a blue edge. We store a dictionary \mathcal{H}_v that contains all heads of \mathcal{B}_v .

Since queries are analogous to the latency-free case, we only focus on label additions/deletions.

An auxiliary procedure. The auxiliary procedure $\text{FixParent}(\ell, \alpha, v)$ for the case with latencies is similar to FixParent in the case without latencies, since it only uses the definition of σ_v as before. The only difference is that that the execution of FixParent also needs to update $\mathcal{H}_{p(v)}$ taking into account the new parent of (ℓ, α, v) .

Label addition. To add label (ℓ, α) on edge e_v , we first insert node (α, ℓ, v) into \mathcal{F} , (ℓ, α) into D_v , and (α, ℓ, v) into $\mathcal{B}_{p(v)}$. We find the maximum value ℓ^- of ℓ'' among all pairs $(\ell'', \alpha'') \in \lambda(e_v)$ with $\alpha'' < \alpha$ using D_v .

Let $(\alpha_1, \ell_1, u_1), \dots, (\alpha_k, \ell_k, u_k)$ be all the nodes in \mathcal{B}_v such that $\ell^- < \alpha_i \leq \ell$ sorted w.r.t. our order (see Figure 4). We observe that all these nodes are consecutive in \mathcal{B}_v . The parent of (α_k, ℓ_k, u_k) becomes (α, ℓ, v) , while the parent of all (α_i, ℓ_i, u_i) becomes $(\alpha_{i+1}, \ell_{i+1}, u_{i+1})$. This requires updating all heads in \mathcal{H}_v between (α_1, ℓ_1, u_1) and $(\alpha_{k-1}, \ell_{k-1}, u_{k-1})$, and can be in time $O(h \log M)$, where h is the number of such heads. This causes all the updated heads to be removed from \mathcal{H}_v , and (α_k, ℓ_k, u_k) to become a new head (if that was not already the case). Next, we perform $\text{FixParent}(\alpha, \ell, v)$. Finally, if the strict predecessor (α^*, ℓ^*, v^*) of (α, ℓ, v) in $\mathcal{B}_{p(v)}$ exists, we perform $\text{FixParent}(\alpha^*, \ell^*, v^*)$.

We now argue that the amortized time complexity of each label insertion is $O(\log M)$ in the incremental case using the accounting method. Let $c > 0$ be a sufficiently large constant. We keep a coin of value at least $c \cdot H_M$, where $H_i = \sum_{j=1}^i \frac{1}{j}$ denotes the i -th harmonic number, on each node of \mathcal{F} that is either a root or is linked to its parent with a blue edge. When a new label is inserted, we pay up to $cM \cdot \frac{1}{M+1}$ for the increase in value of the existing coins so that each coin has value cH_{M+1} , and cH_{M+1} for the coin on the new node (α, ℓ, v) in \mathcal{F} . We also add a coin of value cH_{M+1} on each of (α, ℓ, v) and (α_k, ℓ_k, u_k) . Notice that each node of $(\alpha_1, \ell_1, u_1), \dots, (\alpha_{k-1}, \ell_{k-1}, u_{k-1})$ that changes the edge towards its parent either had no parent, or it was linked to its previous parent with a blue edge. Moreover, the new edges towards its parent must be red. This means that such a node had a coin of value cH_{M+1} that we can use to pay for the cost of the rewiring. All the other operations performed during a label addition cost $O(\log M)$ worst-case time.

Label deletion. To delete the label (ℓ, α) from edge e_v , we remove the node (α, ℓ, v) from \mathcal{F} along with all its incident edges, we delete (ℓ, α) from D_v , and we delete (α, ℓ, v) from $\mathcal{B}_{p(v)}$, and possibly from $\mathcal{H}_{p(v)}$.

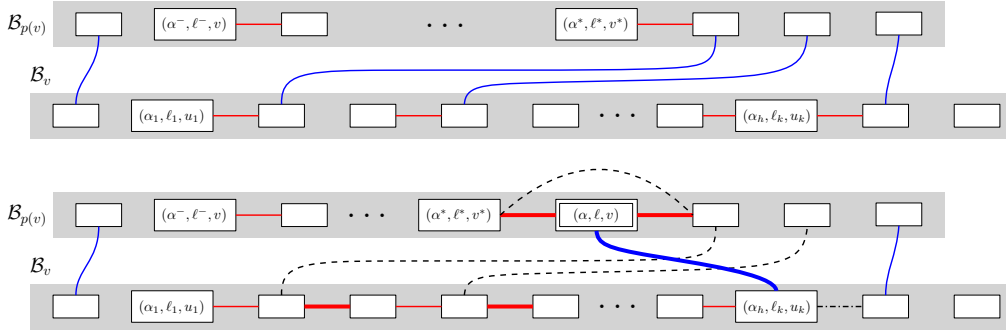
If v is not a leaf in F , let $(\alpha_1, \ell_1, u_1), \dots, (\alpha_k, \ell_k, u_k)$ be the nodes, in order, that had the NextHop equal to (α, ℓ, v) . Notice that such nodes induce a red path in \mathcal{F} , and that (α_k, ℓ_k, u_k) was the unique blue child of (α, ℓ, v) before the deletion.

As a consequence of the deletion, some of the nodes in $(\alpha_1, \ell_1, u_1), \dots, (\alpha_k, \ell_k, u_k)$ will become heads of \mathcal{B}_v , and hence will have a blue edge towards their new parent in $\mathcal{B}_{p(v)}$. We find (α_k, ℓ_k, u_k) in $O(\log M)$ worst-case time via binary search in \mathcal{B}_v and we use it to discover such heads as follows: Initially $(\alpha^*, \ell^*, u^*) = (\alpha_k, \ell_k, u_k)$, and $z = \text{NextHop}(\alpha_k, \ell_k, u_k)$. We binary search \mathcal{B}_v for the rightmost node (α', ℓ', u') (w.r.t. our ordering) that precedes (α_k, ℓ_k, u_k) and is such that $\text{NextHop}(\alpha', \ell', u') \neq z$. We mark node (α', ℓ', u') as a head, and repeat the above procedure using $(\alpha^*, \ell^*, u^*) = (\alpha', \ell', u')$, and $z = \text{NextHop}(\alpha', \ell', u')$. We stop the above procedure as soon as z precedes (α, ℓ, v) in $\mathcal{B}_{p(v)}$. Then this requires $O(\log M)$ time, plus an additional $O(\log M)$ time per discovered head since we can check whether $\text{NextHop}(\alpha', \ell', u') \neq z$ in constant time. Indeed, calling $z = (\alpha_z, \ell_z, v)$, we can define ℓ_z^- as the largest value of ℓ'' among the pairs $(\ell'', \alpha'') \in \lambda(e_v)$ with $\alpha'' < \alpha_z$ (such ℓ_z^- can be found by querying D_v). Then $\text{NextHop}(\alpha', \ell', u') \neq z$ iff $\ell' < \ell_z^-$.

We run FixParent on (α_k, ℓ_k, u_k) and on all the marked heads, which also updates \mathcal{H}_v .

Finally, if the strict predecessor (α^-, ℓ^-, v^-) of (α, ℓ, v) in $\mathcal{B}_{p(v)}$ exists, we perform $\text{FixParent}(\alpha^-, \ell^-, v^-)$.

We now argue that the amortized time complexity of each label deletion is $O(\log M)$ in the decremental case using the accounting method. Let c be sufficiently large constant. We keep a coin of value at least $c \cdot H_M$ on each node of \mathcal{F} that is either a root or is linked to its parent with a red edge. Hence, we pay an amortized cost of $O(M \log M)$ at construction time. When a label is deleted, we pay $2cH_M$ to add a coin on each of (α^-, ℓ^-, v^-) , and



■ **Figure 4** A qualitative representation of the changes resulting from the addition of label (ℓ, α) on edge e_v . On the top: the blocks $\mathcal{B}_{p(v)}$ and \mathcal{B}_v before inserting node (α, ℓ, v) . On the bottom: the new state of $\mathcal{B}_{p(v)}$ and \mathcal{B}_v . Bold lines represent new edges, while dashed lines represent removed ones.

(α_k, ℓ_k, u_k) . Moreover, since each marked head (α', ℓ', u') had a red edge towards its parent before the deletion, we spend such a coin to pay for the execution of FixParent on (α', ℓ', u') . All the other operations performed during a label deletion cost $O(\log M)$ worst-case time.

Uniform latencies. A special case of temporal graphs with latencies is the one with *uniform latencies*, where all time labels have the same latency. In this case, the ordering defined on the nodes of \mathcal{F} is exactly the same as the one used in Section 4, for the case without latencies. For this reason, the data structure we presented in this section guarantees a worst-case update time of $O(\log M)$ because only a constant number of nodes in \mathcal{F} can change their parents.

► **Corollary 7.** *Given a temporal forest of rooted trees with uniform latencies, it is possible to build in $O(n + M \log L)$ time, a data structure of linear size, n is the number of vertices, L is the maximum number of labels on the same temporal edge, and M is the total number of (not necessarily distinct) labels in the forest at the time of the operation. The data structure supports label additions/deletions, link/cut operations and addition/deletion of singleton vertices in $O(\log M)$ worst-case time per operation, EA and LD queries in $O(\log L \cdot \log M)$ worst-case time per operation, and reachability queries in $O(\log M)$ worst-case time per operation.*

References

- 1 Eleni C. Akrida, George B. Mertzios, Sotiris E. Nikolettseas, Christoforos L. Raptopoulos, Paul G. Spirakis, and Viktor Zamaraev. How fast can we reach a target vertex in stochastic temporal graphs? *J. Comput. Syst. Sci.*, 114:65–83, 2020. doi:10.1016/J.JCSS.2020.05.005.
- 2 Eleni C. Akrida, George B. Mertzios, Paul G. Spirakis, and Viktor Zamaraev. Temporal vertex cover with a sliding time window. *J. Comput. Syst. Sci.*, 107:108–123, 2020. doi:10.1016/J.JCSS.2019.08.002.
- 3 Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, 2005. doi:10.1145/1103963.1103966.
- 4 Matthieu Barjon, Arnaud Casteigts, Serge Chaumette, Colette Johnen, and Yessin M. Neggaz. Testing temporal connectivity in sparse dynamic graphs. *CoRR*, abs/1404.7634, 2014. arXiv:1404.7634.

- 5 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.
- 6 Davide Bilò, Sarel Cohen, Tobias Friedrich, Hans Gawendowicz, Nicolas Klodt, Pascal Lenzner, and George Skretas. Temporal network creation games. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*, pages 2511–2519. ijcai.org, 2023. doi:10.24963/IJCAI.2023/279.
- 7 Davide Bilò, Gianlorenzo D’Angelo, Luciano Gualà, Stefano Leucci, and Mirko Rossi. Sparse temporal spanners with low stretch. In Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman, editors, *30th Annual European Symposium on Algorithms, ESA 2022, September 5-9, 2022, Berlin/Potsdam, Germany*, volume 244 of *LIPICs*, pages 19:1–19:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.ESA.2022.19.
- 8 Davide Bilò, Gianlorenzo D’Angelo, Luciano Gualà, Stefano Leucci, and Mirko Rossi. Blackout-tolerant temporal spanners. *J. Comput. Syst. Sci.*, 141:103495, 2024. doi:10.1016/J.JCSS.2023.103495.
- 9 Luiz F. Afra Brito, Marcelo Keese Albertini, Arnaud Casteigts, and Bruno Augusto Nassif Travençolo. A dynamic data structure for temporal reachability with unsorted contact insertions. *Social Network Analysis and Mining*, 12, 2021. URL: <https://api.semanticscholar.org/CorpusID:231847148>.
- 10 Arnaud Casteigts, Anne-Sophie Himmel, Hendrik Molter, and Philipp Zschoche. Finding temporal paths under waiting time constraints. *Algorithmica*, 83(9):2754–2802, 2021. doi:10.1007/S00453-021-00831-W.
- 11 Arnaud Casteigts, Joseph G. Peters, and Jason Schoeters. Temporal cliques admit sparse spanners. *J. Comput. Syst. Sci.*, 121:1–17, 2021. doi:10.1016/J.JCSS.2021.04.004.
- 12 Arnaud Casteigts, Michael Raskin, Malte Renken, and Viktor Zamaraev. Sharp thresholds in random simple temporal graphs. *SIAM J. Comput.*, 53(2):346–388, 2024. doi:10.1137/22M1511916.
- 13 Thomas Erlebach, Frank Kammer, Kelin Luo, Andrej Sajenko, and Jakob T. Spooner. Two moves per time step make a difference. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPICs*, pages 141:1–141:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ICALP.2019.141.
- 14 Thomas Erlebach and Jakob T. Spooner. Parameterised temporal exploration problems. *J. Comput. Syst. Sci.*, 135:73–88, 2023. doi:10.1016/J.JCSS.2023.01.003.
- 15 Thekla Hamm, Nina Klobas, George B. Mertzios, and Paul G. Spirakis. The complexity of temporal vertex cover in small-degree graphs. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*, pages 10193–10201. AAAI Press, 2022. doi:10.1609/AAAI.V36I9.21259.
- 16 Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Recent advances in fully dynamic graph algorithms - A quick reference guide. *ACM J. Exp. Algorithmics*, 27:1.11:1–1.11:45, 2022. doi:10.1145/3555806.
- 17 Petter Holme. Temporal networks. In *Encyclopedia of Social Network Analysis and Mining*, pages 2119–2129. Springer, 2014. doi:10.1007/978-1-4614-6170-8_42.
- 18 Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. Fully dynamic connectivity in $o(\log n(\log \log n)^2)$ amortized expected time. *TheoretCS*, 2, 2023. doi:10.46298/THEORETICS.23.6.

11:16 Temporal Queries for Dynamic Temporal Forests

- 19 David Kempe, Jon Kleinberg, and Amit Kumar. Connectivity and inference problems for temporal networks. *Journal of Computer and System Sciences*, 64(4):820–842, 2002. doi:10.1006/jcss.2002.1829.
- 20 Edward M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985. doi:10.1137/0214021.
- 21 Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.
- 22 Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path problems in temporal graphs. *Proc. VLDB Endow.*, 7(9):721–732, 2014. doi:10.14778/2732939.2732945.