On Short Fastest Paths in Temporal Graphs

Umesh Sandeep Danda
Zippr Private Limited, Hyderabad
India

G. Ramakrishna
Indian Institute of Technology Tirupati
India

Jens M. Schmidt
Hamburg University of Technology
Institute for Algorithms and Complexity, Germany

M. Srikanth
Indian Institute of Technology Tirupati
India

Abstract

[1] Temporal graphs equip their directed edges with a *departure time* and a *duration*, which allows to model a surprisingly high number of real-world problems. Recently, Wu et al. have shown that a *fastest* path in a temporal graph G from a given vertex s to a vertex z can be computed in near-linear time, where a *fastest* path is one that minimizes the arrival time at z minus the departure time at s.

Here, we consider the natural problem of computing a fastest path from s to z that is in addition *short*, i.e. minimizes the sum of durations of its edges; this maximizes the total amount of spare time at stops during the journey. Using a new dominance relation on paths in combination with lexicographic orders on the departure and arrival times of these paths, we derive a near-linear time algorithm for this problem with running time $O(n + m \log p(G))$, where $n := |V(G)|$, $m := |E(G)|$ and $p(G)$ is upper bounded by both the maximum in-degree and the maximum edge duration of G.

The dominance relation is interesting in its own right, and may be of use for several related problems like fastest paths with minimum

[1]The final authenticated version is available online at https://doi.org/10.1007/978-3-030-68211-8_4.

fare, fastest paths with minimum number of stops, and other pareto-optimal path problems in temporal graphs.

1 Introduction

Temporal graphs capture various problems such as message dissemination in online social networks, epidemics spreading in complex networks and routing in scheduled public transportation networks [10]. This generality comes with a price: many standard graph parameters (such as the number of strongly connected components) are not known to admit polynomial-time algorithms in temporal graphs, and not even standard results in combinatorics like Menger's theorem hold without adapting them adequately [6, 8].

On the other hand, a growing number of positive results has been developed in recent years for various problems in temporal graphs [1, 4, 6, 7, 9, 12].

In this paper, we focus on path problems in temporal graphs, for which, in contrast to static graphs, various notions of optimality exist [3, 5, 11]. For example, one may not only want to find the *fastest* paths mentioned above, but also *shortest* paths, which minimize the sum of durations of their edges (we give precise definitions in Section 1.2).

It was recently shown in [11] that, given a temporal graph G and two of its vertices s and z, both fastest and shortest paths from s to z can be computed efficiently in running times $O(n+m \log c_{min})$ and $O(n+m \log c_{in}(G))$, respectively, where $c_{in}(G)$ is the maximum number of ingoing edges over all vertices of G, S is the number of outgoing edges of s with distinct departure times, and $c_{min} = \min\{|S|, c_{in}(G)\}$.

A natural strengthening that we investigate here is to compute a fastest path from s to z that has minimal duration. To our surprise, no efficient algorithm seems to be known for this problem.

1.1 Temporal Graphs

A *temporal graph* G is a pair (V, E), where V is a finite set and $E := (e_1, e_2, \ldots, e_m)$ is a finite sequence such that $e_i := (v_i, w_i, t_i, d_i) \in V \times V \times \mathbb{N} \times \mathbb{N}^{>0}$ and $v_i \neq w_i$ for every $1 \leq i \leq m$. For every $1 \leq i \leq m$, we call e_i an *edge* of G, v_i and w_i the *source* and *target vertex* of e_i, t_i the *departure time* of e_i and d_i the *duration* of e_i. Hence, in the terminology of usual graphs, every edge e_i of G is directed (as e_i is ordered), not a self-loop (parallel edges may occur), and has positive duration. Every edge e_i is equipped with a departure time t_i and a duration d_i, where t_i is the point in time at which one may depart from v_i in order to arrive at w_i at time $t_i + d_i$; we call *arr*$(e_i) := t_i + d_i$ the *arrival time* of e_i.

This model generalizes the models of temporal graphs that were used in [3]. In the above definition, the edges $(e_i)_i$ are used in a stream representation: for temporal graphs, it is usually assumed that the edges in this stream $(e_i)_i$ are ordered with respect to some natural and easy-to-pick ordering such as their creation, collection or deletion [11, Section 4.1]. Here, we assume that the edges are ordered monotonically increasing according to their arrival times, so that we have $i < j$ if and only if $\mathrm{arr}(e_i) \leq \mathrm{arr}(e_j)$. If for some reason such an ordering cannot be expected in a particular use case, a sorting routine with additional running time $O(m \log m)$ has to be invoked in advance.

We inherit standard graph-theoretic notions like paths and cycles (both are always given as edge sequences) for temporal graphs G. A path from a vertex s to a vertex z ($s = z$ is possible) is called an *s-z-path*. For any $G = (V, E)$, we define $V(G) := V$, $E(G) := E$ and $n := |V(G)|$ (note that $m := |E(G)|$ by definition of E).

1.2 Our Result

A path $P := (e_{j_1}, \ldots, e_{j_k})$ of a temporal graph G is *temporal* if $t_{j_i} + d_{j_i} \leq t_{j_{i+1}}$ for every $1 \leq i < k$. We call *dep*$(P) := t_{j_1}$ the *departure time* of P and *arr*$(P) := t_{j_k} + d_{j_k}$ the *arrival time* of P if $k > 0$. The *journey time* of such a temporal path P is *journey(P)* $:= \mathrm{arr}(P) - \mathrm{dep}(P)$, and the *duration* of P is *dur(P)* $:= \sum_{i=1}^{k} d_{j_i}$ (see Figure 1).

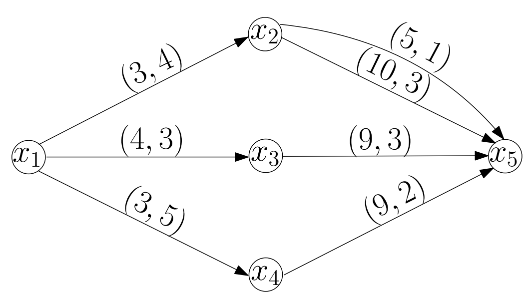


Figure 1: A temporal graph G in which the departure time t_i and the duration d_i of every edge e_i is shown. The path $P := (x_1x_3, x_3x_5)$ is a fastest path that has journey time $8 = 9 + 3 - 4$, and $Q := (x_1x_4, x_4x_5)$ is another fastest path from x_1 to x_5 that has journey time $8 = 9 + 2 - 3$. However, only P is a short fastest path, as $\mathrm{dur}(P) = 6 = 3 + 3 < \mathrm{dur}(Q) = 7 = 5 + 2$.

Definition 1. A temporal s-z-path P of a temporal graph G is called

(i) *fastest* if every temporal s-z-path Q of G satisfies $\text{journey}(P) \leq \text{journey}(Q)$,

(ii) *shortest* if every temporal s-z-path Q of G satisfies $\text{dur}(P) \leq \text{dur}(Q)$, and

(iii) *short fastest* if P is fastest and every fastest temporal s-z-path Q of G satisfies $\text{dur}(P) \leq \text{dur}(Q)$.

In other words, P is *short fastest* if P is fastest and has minimum duration among all fastest s-z-path of G. Note that all three notions fix the start- and end-vertex of the paths in question, while allowing an arbitrary departure time at vertex s. Short fastest paths arise naturally when we want to travel from s to z in the fastest journey time possible such that the total amount of time spent traveling is minimized (this maximizes the total amount of spare time at stops during the journey).

For an edge e_i, let $p(e_i) := |\{\text{arr}(e_j) : w_j = v_i \text{ and } t_i \leq \text{arr}(e_j) \leq \text{arr}(e_i)\}|$ be the number of integers in $[t_i, \text{arr}(e_i)]$ that are arrival times for at least one incoming edge to v_i. In particular, we have $p(e_i) \leq d_i$ and $p(e_i)$ is at most the in-degree of v_i. Let $p(G) := \max\{p(e_i) : 1 \leq i \leq m\}$ and let $\delta^-(G)$ be the maximum in-degree of G. Given two vertices s and z of a temporal graph G, the problem ShortFastestPath(s, z, G) asks for a short fastest temporal s-z-path of G. We solve this problem as follows.

Theorem 2. *Given a source vertex s of a temporal graph G on n vertices and m edges, short fastest paths from s to every vertex $z \neq s$ can be computed in total time $O(n + m \log p(G))$, where $p(G) \leq \min\{\delta^-(G), \max\{d_i : 1 \leq i \leq m\}\}$.*

As the duration in public-transport networks is often bounded by a constant, the factor $\log p(G)$ in our running time is typically insignificant for applications. The algorithm of Theorem 2 may easily be adapted to compute short fastest paths in given time intervals, and to allow rational departure and duration times (e.g. by multiplying with the greatest common divisor in advance). Further, the algorithm may also be customized to solve related problems such as computing a fastest path with minimum waiting time, computing a fastest path with minimum fare, and computing a fastest path with minimum number of transfers at intermediate stations.

While our algorithm is inspired by the algorithm in [11] for fastest paths, it deviates from this algorithm by using a new dominance relation on paths and lexicographic orderings on the departure and arrival times of these paths. These two ideas allow us to perform various operations on dominating paths such as searching, insertion, and deletion efficiently. Another difference is that our algorithm processes the edges of G by increasing arrival time.

2 Dominating Paths

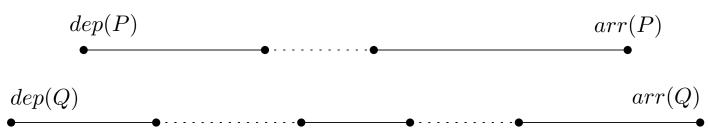
From now on, let a temporal graph G and a source vertex s of G for the problem ShortFastestPath be given. We first provide structural properties of temporal paths that are useful to reduce the search space.

Definition 3. For temporal x-y-paths P and Q of G, P *dominates* Q if either

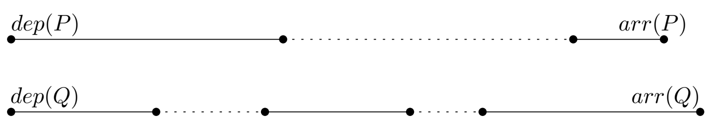
(i) $\mathrm{dep}(P) > \mathrm{dep}(Q)$ and $\mathrm{arr}(P) \leq \mathrm{arr}(Q)$,

(ii) $\mathrm{dep}(P) = \mathrm{dep}(Q)$, $\mathrm{arr}(P) < \mathrm{arr}(Q)$ and $\mathrm{dur}(P) \leq \mathrm{dur}(Q)$, or

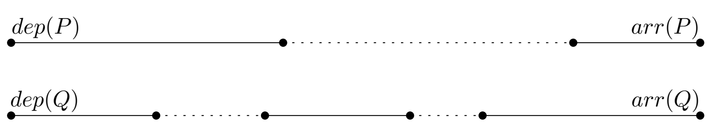
(iii) $\mathrm{dep}(P) = \mathrm{dep}(Q)$, $\mathrm{arr}(P) = \mathrm{arr}(Q)$ and $\mathrm{dur}(P) < \mathrm{dur}(Q)$.



(a) $\mathrm{dep}(P) > \mathrm{dep}(Q)$ and $\mathrm{arr}(P) \leq \mathrm{arr}(Q)$



(b) $\mathrm{dep}(P) = \mathrm{dep}(Q)$, $\mathrm{arr}(P) < \mathrm{arr}(Q)$ and $\mathrm{dur}(P) \leq \mathrm{dur}(Q)$



(c) $\mathrm{dep}(P) = \mathrm{dep}(Q)$, $\mathrm{arr}(P) = \mathrm{arr}(Q)$ and $\mathrm{dur}(P) < \mathrm{dur}(Q)$

Figure 2: Three instances of a path P that dominates Q. Solid and dotted lines depict the duration of edges and the waiting times for the next departure, respectively.

The three cases of Definition 3 are depicted in Figure 2. A temporal x-y-path is *dominating* if it is not dominated by any other temporal x-y-path, and *non-dominating* otherwise. In order to motivate these definitions, observe that every short fastest path is dominating (Definition 3(i) and (ii) strictly decrease the journey time, while Definition 3(iii) strictly decreases the duration). We therefore are interested in computing all dominating paths. Definition 3 and the resulting properties of the dominance relation on paths will be crucial for establishing the structural properties that allow the algorithm to be efficient in the remainder of the paper.

For a path, a *prefix subpath* of this path is a subpath of this path that starts at the same start vertex. The correctness of shortest path algorithms in traditional graphs such as Dijkstra's algorithm rely heavily on the fact that subpaths of shortest paths are again shortest. For temporal graphs however, such properties are bound to fail (in fact, they fail for fastest as well as for shortest paths). For example, any prefix subpath of a fastest path in which the departure time of the last edge is sufficiently far away from the arrival time of the second last edge may not be fastest (as the second last vertex may be reached by much faster paths). The next lemma shows that paths that are dominating (as defined in the last paragraph) obey this property.

Lemma 4. *Every prefix subpath of every dominating path is dominating.*

Proof. Assume to the contrary that Q' is a (temporal) non-dominating prefix x-y-path of a dominating path Q. Then $Q' \neq Q$, since Q is dominating, and there is a temporal x-y-path P' that dominates Q'. Let P be the path obtained from Q by replacing the subpath Q' with P'; in particular, $\mathrm{arr}(P) = \mathrm{arr}(Q)$.

Since P' dominates Q', Q' satisfies exactly one of the Conditions 3(i)–(iii). The first is not satisfied, as $\mathrm{dep}(P') > \mathrm{dep}(Q')$ implies $\mathrm{dep}(P) > \mathrm{dep}(Q)$, which contradicts that Q is dominating due to $\mathrm{arr}(P) = \mathrm{arr}(Q)$. Condition 3(ii) is not satisfied, as $\mathrm{arr}(P') < \mathrm{arr}(Q')$ and $\mathrm{dur}(P') \leq \mathrm{dur}(Q')$ imply $\mathrm{dur}(P) < \mathrm{dur}(Q)$ due to $\mathrm{dep}(P') = \mathrm{dep}(Q')$, which contradicts that Q is dominating. Condition 3(iii) is not satisfied, as $\mathrm{dur}(P') < \mathrm{dur}(Q')$ implies $\mathrm{dur}(P) < \mathrm{dur}(Q)$, which contradicts that Q is dominating. This gives the claim. $\square$

3 An Algorithm for Short Fastest Paths

Given any vertex s of a temporal graph G, we describe an efficient one-pass algorithm by dynamic programming that computes the journey time and duration of a short fastest path from s to any other vertex $z \neq s$ in G.

For every temporal x-y-path P, we call $(\mathrm{dep}(P), \mathrm{arr}(P), \mathrm{dur}(P))$ a *temporal triple from x to y*. This allows to inherit the dominance relation of temporal paths to temporal triples as follows: a triple (t, a, d) from x to y *dominates* a triple (t', a', d') from x to y if there is a temporal x-y-path P with temporal triple (t, a, d) that dominates a temporal x-y-path Q with temporal triple (t', a', d'). The *dominating triples* from x to $y \neq x$ are then defined analogously to paths that are dominating; in addition, for every $1 \leq i \leq m$ such that $v_i = s$, let $(t_i, t_i, 0)$ be an artificial *dominating triple* from s to s. These artificial dominating triples will later allow the algorithm to start at s using any outgoing edge e_j of s at its departure time t_j.

We will compute dominating triples of G by starting with an edge-less subgraph of G and updating these triples each time after the next edge of $E(G)$ in the given ordering of $E(G)$ is added. We therefore define a sequence of temporal graphs that adheres to this ordering (i.e. adds the edges of $E(G)$ one by one). For every $0 \leq i \leq m$, let G_i be the temporal graph $(V(G), \{e_1, \dots, e_i\})$. Hence, G_0 has no edges at all and, for every $1 \leq i \leq m$, $V(G_i) = V(G)$ and e_i is the only edge of G_i that is not in G_{i-1}.

In every graph, we aim to maintain for every $y \neq s$ the list L_y of all dominating triples from s to y. The lists L_y are initially empty. We will store the final journey time and duration of a short fastest s-y-path for every $y \neq s$ in *journey*(y) and *dur*(y), respectively.

Now we add the edges of $E(G)$ one by one, which effectively iterates through the sequence $G_0, \dots, G_m$ (see Algorithm 1). After the edge e_i has been processed, we ensure that L_y stores the set of all dominating triples from s to $y \neq s$ in G_i. For an edge $e_i \in E(G)$, we say that (t, a, d) is a *predecessor triple of e_i* if

- (t, a, d) is a dominating triple from s to v_i,
- $a \leq t_i$, and
- a is maximal among all such triples.

A predecessor triple (t, a, d) thus allows to traverse e_i after taking its corresponding dominating s-v_i-path. Note that not every edge has a predecessor triple, and that the artificial dominating triples correspond to temporal paths having no edge (and thus duration 0) that allow to traverse any outgoing edge of s).

Without loss of generality, we may ignore all edges e_i whose target vertex is s, as s is the start vertex. In order to update L_y during the processing

Algorithm 1: Short Fastest Path

Input: A vertex s of a temporal graph $G = (V, E)$, where the sequence E is ordered increasingly according to arrival time.

Output: For every vertex $z \neq s$ in G, the journey time and duration of a short fastest path from s to z.

1 Initialize $L_y := \emptyset$ and $\text{journey}(y) := \text{dur}(y) := \infty$ for every vertex $y \neq s$;
2 **for** $i = 1$ *to* m **do**
3   **if** $w_i = s$ **then continue**;
4   **if** $v_i = s$ **then** Append artificial dominating triple $(t_i, t_i, 0)$ to L_{v_i};
5   **if** L_{v_i} *contains a predecessor triple of* e_i **then**
6   Choose a predecessor triple (t, a, d) of e_i in L_{v_i};
7         $T := (t, \text{arr}(e_i), d + d_i)$;
8   **if** $T \notin L_{w_i}$ **then**
9   Append T to L_{w_i};
10   Delete all elements of L_{w_i} that are dominated by an element of L_{w_i};
11   **if** $\mathit{arr}(e_i) - t < \mathit{journey}(w_i)$ *or* $(\mathit{arr}(e_i) - t = \mathit{journey}(w_i)$ *and* $d + d_i < \mathit{dur}(w_i))$ **then**
12                 $\text{journey}(w_i) := \text{arr}(e_i) - t$;
13                 $\text{dur}(w_i) := d + d_i$;
14 **return** *journey(z) and dur(z) for every* $z \neq s$;

phase of e_i if $w_i \neq s$, we choose a predecessor triple of e_i in L_{v_i} (if exists) in Line 6 of Algorithm 1 and create from it a new triple T in Line 7. The newly created triple T is then appended to L_{w_i} in Line 9, followed by removing all triples of L_{w_i} that are dominated by an element of L_{w_i}. Finally, the journey time *journey*(w_i) and duration *dur*(w_i) that are attained by T are updated if they improve the solution.

4 Correctness

In order to show the correctness of Algorithm 1, we rely on the next three basic lemmas, which collect helpful properties of dominating paths with respect to the sequence $G_0, \ldots, G_m$.

Lemma 5. *For every $1 \le i \le m$, every temporal path of G_i that contains e_i has e_i as its last edge.*

Proof. Assume to the contrary that G_i has a temporal path P that contains e_i such that the last edge of P is $e_j \neq e_i$. Then $j < i$ by definition of G_i, which implies $\mathrm{arr}(e_j) \le \mathrm{arr}(e_i)$ by the ordering assumed for E. This contradicts that P is temporal. $\square$

The next two lemmas explore whether dominating and non-dominating paths are preserved when going from G_{i-1} to G_i.

Lemma 6. *For every $1 \le i \le m$, every non-dominating path P of G_{i-1} is non-dominating in G_i.*

Proof. Since P is non-dominating, G_{i-1} contains a temporal path Q that dominates P. Since neither Q nor P contains e_i, Q dominates P also in G_i. Hence, P is non-dominating in G_i. $\square$

In contrast to Lemma 6, a dominating path of G_{i-1} may in general become non-dominating in G_i, for example by Definition 3(iii) if e_i and the path have the same source and target vertex and the duration of e_i is very small. The next lemma states a condition under which dominating paths stay dominating paths.

Lemma 7. *For every $1 \le i \le m$, every dominating x-y-path P of G_{i-1} that satisfies $y \neq w_i$ is dominating in G_i.*

Proof. Assume to the contrary that P is non-dominating in G_i. Then an x-y-path Q dominates P in G_i; in particular, Q is temporal. Since $y \neq w_i$, e_i is not the last edge of Q. By Lemma 5, e_i is not contained in Q at all, so that Q is a path of G_{i-1}. Since Q dominates P in G_i, Q does so in G_{i-1}, which contradicts that P is dominating in G_{i-1}. $\square$

Let $L_y(i)$ be the list L_y in Algorithm 1 after the edge e_i has been processed. The correctness of Algorithm 1 is based on the invariant revealed in the next lemma.

Lemma 8. *For every $0 \le i \le m$ and every vertex $y \neq s$ of G_i, (t, a, d) is a dominating triple from s to y in G_i if and only if $(t, a, d) \in L_y(i)$.*

Proof. Due to space constraints, we defer this to the full version of this paper. $\square$

For $i = m$, we conclude the following corollary.

Corollary 9. *At the end of Algorithm 1, L_y contains exactly the dominating triples from s to y in G for every $y \neq s$.*

Theorem 10. *Algorithm 1 computes the journey-time and duration of a short fastest path from s to every vertex $z \neq s$ in G.*

Proof. Every short fastest path of G from s to z is dominating. By Corollary 9, L_z therefore contains every short fastest path of G from s to z at the end of Algorithm 1. By comparing the journey-time and duration of every path that is added to L_z (this may be a superset of the short fastest paths), Algorithm 1 computes the journey-time and duration of a short fastest path from s to z in G. □

Now the correctness of Theorem 2 follows from Theorem 10 by tracing back the path from z to s. This may be done by storing an additional pointer to the last edge of the current short fastest path found for every vertex z. Since following this pointer is only a constant-time operation, we may trace back the short fastest path from z to s in time proportional to its length.

5 Running Time

We investigate the running time of Algorithm 1. If the edge e_i satisfies $v_i = s$, the artificial predecessor triple $(t_i, t_i, 0)$ of Line 4 will reside at the end of the ordered list $L_v(i)$, and therefore can be appended to and retrieved from $L_v(i)$ in constant time. It suffices to clarify how we implement Lines 5–6 and Line 10, since every other step is computable in constant time. In particular, we have to maintain dominating triples in L_y and be able to compute a predecessor triple of e_i in L_y efficiently.

For every L_y, we enforce a lexicographic order $<_{\text{lex}}$ on the first two elements of all dominating triples stored. The first two elements suffice, as two distinct dominating triples (we do not store duplicates) differ always in their first two elements by Definition 3(iii). The following basic lemma will be useful.

Lemma 11. *Let $T = (t, a, d)$ and $T' = (t', a', d')$ be two distinct dominating triples from x to y such that $T <_{lex} T'$. Then $a < a'$ and either $t < t'$ or ($t = t'$ and $d > d'$).*

Proof. First, assume $t < t'$. Then $a < a'$, as otherwise T' dominates T by Definition 3(ii), which contradicts that T is dominating. In the remaining case, we have $t = t'$ and $a < a'$ by the lexicographic order on the first two elements. Then $d > d'$, as otherwise T dominates T' by Definition 3(ii). □

Let $T_1 <_{\text{lex}} T_2 <_{\text{lex}} \cdots <_{\text{lex}} T_r$ be the dominating triples of L_{w_i} in Line 10 and let $T_j = (t^j, a^j, d^j)$ for every $1 \leq j \leq r$. By Lemma 11, $a^1 < a^2 < \cdots < a^r$. Let $T = (t, \text{arr}(e_i), d + d_i)$ be the new dominating triple in G_i that is created in Line 7. Since e_i is the currently processed edge of Algorithm 1 and $E(G)$ is ordered by increasing arrival times, we have $a^r \leq \text{arr}(e_i)$. Thus, appending T to L_{w_i} in Line 9 preserves the lexicographic ordering of L_{w_i}. The next two lemmas determine which elements of L_{w_i} are dominated by an element of L_{w_i}.

Lemma 12. *(i) T does not dominate any element of $\{T_1, T_2, \ldots, T_{r-1}\}$.*

(ii) If an element of $\{T_1, T_2, \ldots, T_{r-1}\}$ dominates T, then T_r dominates T.

Proof. By Lemma 11 and since $E(G)$ is ordered by increasing arrival times, $a^1 < a^2 < \cdots < a^r \leq \text{arr}(e_i)$. Then $a^j < \text{arr}(e_i)$ for every $1 \leq j < r$, so that T does not dominate T_j by Definition 3. This gives the first claim.

For the second claim, let T_j dominate T for some $1 \leq j < r$. Then either Definition 3(i), (ii) or (iii) holds. In Case (i), we have $t^j > t$ and $a^j \leq \text{arr}(e_i)$, which implies $t < t^r$ by the lexicographic ordering. Since $a^r \leq \text{arr}(e_i)$, T_r dominates T. In both Cases (ii) and (iii), we have $t^j = t$, $a^j \leq \text{arr}(e_i)$ and $d^j \leq d + d_i$. By Lemma 11, either $t^j < t^r$ or $t^j = t^r$. If $t^j < t^r$, we have $t < t^r$ and $a^r \leq \text{arr}(e_i)$, so that T_r dominates T. If otherwise $t^j = t^r$, we have $d^j > d^r$ by Lemma 11. Then, since $d^j \leq d + d_i$, we have $t^r = t$, $a^r \leq \text{arr}(e_i)$ and $d + d_i > d^r$, so that T_r dominates T. □

Lemma 13. *There is no element of $\{T_1, \ldots, T_r\}$ that dominates another element of $\{T_1, \ldots, T_r\}$. After Line 10, L_{w_i} consists of*

(i) $(T_1, \ldots, T_{r-1}, T)$ if T dominates T_r (then Line 10 deletes only T_r from L_{w_i}),

(ii) $(T_1, \ldots, T_{r-1}, T_r, T)$ if no element of $\{T_r, T\}$ dominates the other element of $\{T_r, T\}$ (then Line 10 deletes nothing from L_{w_i}), and

(iii) $(T_1, \ldots, T_{r-1}, T_r)$ if T_r dominates T (then Line 10 deletes only T from L_{w_i}).

Proof. Since T is the only triple that was added to L_{w_i}, every element of $\{T_1, \ldots, T_r\}$ was dominating in G_{i-1}. Thus, no element of $\{T_1, \ldots, T_r\}$ dominates another element of this set in G_i. By Lemma 12(i), T does not dominate any element of $\{T_1, T_2, \ldots, T_{r-1}\}$.

Consider Claim (i). Since T dominates T_r, T_r does not dominate T. Then the contrapositive of Lemma 12(ii) implies that no element of $\{T_1, \ldots, T_r\}$ dominates T. Together with the first claim, this gives $L_{w_i} = (T_1, \ldots, T_{r-1}, T)$.

Consider Claim (ii). Since T_r does not dominate T, the same argument as before implies that no element of $\{T_1, \ldots, T_r\}$ dominates T. Since T does not dominate T_r, the first claim of this lemma implies $L_{w_i} = (T_1, \ldots, T_{r-1}, T_r, T)$. Claim (iii) follows directly from the first claim and the fact that T_r dominates T. □

Algorithm 2: Deleting Dominating Triples (Line 10 of Algorithm 1)

Input: A list L_{w_i} of dominating triples ordered by $<_{\text{lex}}$, and the new triple T of G_i from Line 7.

```
Retrieve the last element T_r of L_{w_i} if L_{w_i} ≠ ∅;
if L_{w_i} = ∅ or T ≠ T_r then append T to L_{w_i};
if |L_{w_i}| ≥ 2 then
    if T dominates T_r then delete T_r from L_{w_i};
    if T_r dominates T then delete T from L_{w_i};
```

Lemma 13 allows us to implement Line 10 of Algorithm 1 very efficiently by comparing just the last element T_r of L_{w_i} with T. This can be done in constant time by Algorithm 2.

It remains to show how a predecessor triple (t, a, d) of e_i in L_{v_i} in Lines 5+6 of Algorithm 1 can be computed efficiently. By Lemma 11 and its preceding remark, the arrival times of all triples in L_{v_i} are for every i distinct. Hence, for every i, the number of elements in L_{v_i} is at most the maximum in-degree $\delta^-(G)$ of G.

Since the triples in L_{v_i} are ordered by $<_{\text{lex}}$, the last triple in L_{v_i} (if exists) whose arrival time is at most t_i is a predecessor triple of e_i, so that we only have to compute this unique predecessor triple for every i. In order to do this, we might use binary search on the arrival times of triples of L_{v_i}. We achieve however the slightly better running time $O(\log p(G))$ when first using an exponential search [2] that starts with the triple having highest arrival time (which is at most $\mathrm{arr}(e_i)$ due to the edge-ordering) until some triple with arrival time at most t_i is found (see Figure 3 for an example).

If such a triple exists, there is also a predecessor triple of e_i, which we then compute by binary search in the resulting range; see Algorithm 3 for a detailed description. This takes only time $O(\log p(G))$, which is upper

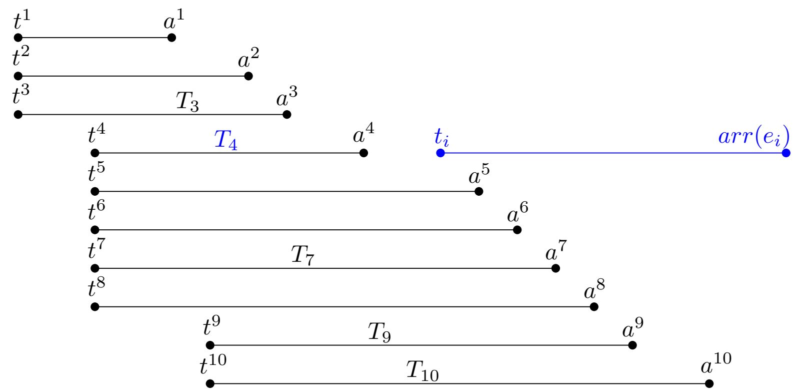


Figure 3: A list $L_{v_i} = (T_1, \ldots, T_{10})$ containing dominating triples from s to v_i, ordered by $<_{\text{lex}}$ from top to bottom, and the departure and arrival times of e_i. The only predecessor triple of e_i is $T_4 = (t^4, a^4, d^4)$. In order to compute T_4, Algorithm 3 tests the arrival times of T_{10}, T_9 and T_7 until it stops at T_3 (because $a^3 \leq t_i$) and computes $T_4 \in \{T_3, \ldots, T_6\}$ by binary search.

bounded by $O(\min\{\delta^-(G), \max\{d_i : 1 \leq i \leq m\}\})$. If no such triple exists, there is no predecessor triple of e_i, and this information is given as output.

Algorithm 3: Computing a Predecessor Triple (Lines 5+6 of Algorithm 1)

Input: A list $L_{v_i} = (T_1, T_2, \ldots, T_r)$ of dominating triples in G_{i-1} ordered by $<_{\text{lex}}$ such that $T_j = (t^j, a^j, d^j)$ for every $1 \leq j \leq r$, and an edge e_i of G.

Output: A predecessor triple of e_i if exists, and otherwise the output "not existent"

$j := 1$;
while $j \leq r$ *and* $a_{r+1-j} > t_i$ **do** $j := 2j$;
if $j \leq r$ **then**
    Compute the maximal $r + 1 - j \leq l < r + 1 - \lfloor j/2 \rfloor$ such that $a_l \leq t_i$ by performing binary search (then a_l is maximal by Lemma 11);
    return T_l;
else
    output "not existent";

Lemma 14. *The running time of Algorithm 1 for a temporal graph G on n vertices and m edges is $O(n + m \log p(G))$.*

Proof. Apart from Lines 5+6 and 10, every of the m edges can be processed in constant time. By Algorithms 2 and 3, the running times for Lines 5+6 and 10 amount to $O(\log p(G))$ and $O(1)$ time for every edge, which gives the claim. □

This concludes the proof of Theorem 2.

References

[1] Eric Aaron, Danny Krizanc, and Elliot Meyerson. DMVP: Foremost waypoint coverage of time-varying graphs. In *40th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'14)*, pages 29–41, 2014.

[2] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976.

[3] B.-M. Bui Xuan, A. Ferreira, and A. Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, 14(2):267–285, 2003.

[4] Guillermo de Bernardo, Nieves R. Brisaboa, Diego Caro, and M. Andrea Rodríguez. Compact data structures for temporal graphs. In *Data Compression Conference (DCC'13), IEEE*, page 477, 2013.

[5] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly simple and fast transit routing. In *International Symposium on Experimental Algorithms (SEA'13), volume 7933 of LNCS*, pages 43–54, 2013.

[6] David Kempe, Jon M. Kleinberg, and Amit Kumar. Connectivity and inference problems for temporal networks. *J. Comput. Syst. Sci.*, 64(4):820–842, 2002.

[7] Qingkai Liang and Eytan Modiano. Survivability in time-varying networks. *IEEE Transactions on Mobile Computing*, 16(9):2668–2681, 2017.

[8] George B. Mertzios, Othon Michail, and Paul G. Spirakis. Temporal network optimization subject to connectivity constraints. *Algorithmica*, 81(4):1416–1449, 2019.

[9] Othon Michail and Paul G. Spirakis. Traveling salesman problems in temporal graphs. *Theoretical Computer Science*, 634:1–23, 2016.

[10] John Kit Tang, Mirco Musolesi, Cecilia Mascolo, and Vito Latora. Characterising temporal distance and reachability in mobile and online social networks. *Computer Communication Review*, 40(1):118–124, 2010.

[11] Huanhuan Wu, James Cheng, Yiping Ke, Silu Huang, Yuzhen Huang, and Hejun Wu. Efficient algorithms for temporal path computation. *IEEE Trans. Knowl. Data Eng.*, 28(11):2927–2942, 2016.

[12] Philipp Zschoche, Till Fluschnik, Hendrik Molter, and Rolf Niedermeier. The complexity of finding small separators in temporal graphs. *Journal of Computer and System Sciences*, 107:72–92, 2020.