Path Problems in Temporal Graphs

Huanhuan Wu[*], James Cheng[*], Silu Huang[*], Yiping Ke[#], Yi Lu[*], Yanyan Xu[*]

[*]*Department of Computer Science and Engineering, The Chinese University of Hong Kong*
{hhwu,jcheng,slhuang,ylu,yyxu}@cse.cuhk.edu.hk
[#]*Institute of High Performance Computing, Singapore*
keyp@ihpc.a-star.edu.sg

ABSTRACT

Shortest path is a fundamental graph problem with numerous applications. However, the concept of classic shortest path is insufficient or even flawed in a temporal graph, as the temporal information determines the order of activities along any path. In this paper, we show the shortcomings of classic shortest path in a temporal graph, and study various concepts of "shortest" path for temporal graphs. Computing these temporal paths is challenging as subpaths of a "shortest" path may not be "shortest" in a temporal graph. We investigate properties of the temporal paths and propose efficient algorithms to compute them. We tested our algorithms on real world temporal graphs to verify their efficiency, and also show that temporal paths are essential for studying temporal graphs by comparing shortest paths in normal static graphs.

1. INTRODUCTION

There have been a lot of interests in research on graph data management and graph mining in recent years, mainly thanks to the increasing popularity of many online social networks and communication networks. Existing research has mainly focused on the study of static graphs, while some have also considered dynamic graphs as a sequence of updates to static graphs. However, many real world graphs are actually **temporal graphs**, in which a vertex communicates with another vertex at specific time instances. For example, assume that Figure 1(a) shows an air-transport network, then the 2 edges from a to b indicate that there is a flight from a to b on Day 1 and Day 2, i.e., the numbers 1 and 2 on the edges represent flight departure time.

There are numerous real world applications for which data can be modeled as a temporal graph. For example, A calls B at time t in phone call networks, A sends message to B at time t in Short Message Service or emails networks, A follows B at time t in social networks, A cites B at time t in citation networks, A works with B at time t in collaboration networks, information spreads from A to B at time t in information dissemination networks, to name but a few. In a survey of temporal networks [8], Holme and Saramki also describe in details various temporal networks in cell biology, neural and brain connections, ecological systems, infra-structural networks, physical proximity, distributed computing, and so on.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 40th International Conference on Very Large Data Bases, September 1st - 5th 2014, Hangzhou, China.
Proceedings of the VLDB Endowment, Vol. 7, No. 9
Copyright 2014 VLDB Endowment 2150-8097/14/05.

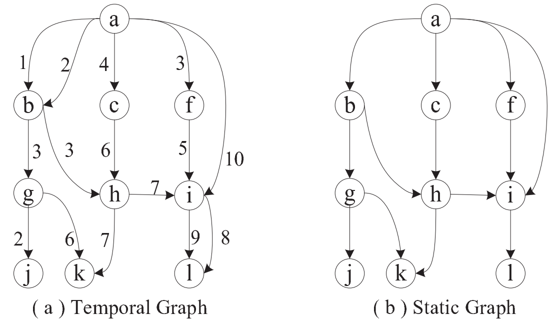

Figure 1: Temporal graph G and its condensed static graph G_s

The temporal graphs listed above also have a static version. In fact, temporal graphs are commonly condensed into static graphs because their static version is much easier to handle. For example, computing the strongly connected components (SCCs) of a static graph has a linear-time algorithm, but computing the SCCs of a temporal graph has no known polynomial-time algorithm [13].

Condensing a temporal graph into a static graph loses all the temporal information which is critical to the understanding of the relationship between objects in the graph. Not only so, the main concern is in fact that the resultant static graph often presents erroneous information that leads to serious incorrect understanding of the graph or relationship between objects. We illustrate the above-mentioned problems by the following example.

EXAMPLE 1. *Figure 1(a) shows a temporal graph G. Assume that G is an air-transport network, then each vertex represents an airport and the number on each edge is a flight's departure day. For simplicity, we assume that the duration of each flight is 1 day. Figure 1(b) shows the condensed static graph G_s of G.*

We can see some paths in the static graph may not be a meaningful path in the temporal graph. For example, $\langle a, b, g, j \rangle$ is a path in Figure 1(b), but $\langle a, b, g, j \rangle$ in Figure 1(a) is problematic because g has only one flight to j on Day 2 but we cannot reach g before Day 4 (leaving b on Day 3 and taking 1 day to fly from b to g).

Now consider a shortest path from a to l in the two graphs. In Figure 1(b), the shortest path is $\langle a, i, l \rangle$ with distance 2. But in the temporal graph in Figure 1(a), if we take the edge (a, i), then we cannot take either of the flights from i to l since the flight from a arrives at i on Day 11. Instead, a valid temporal path is $\langle a, f, i, l \rangle$ with distance 3, by going from a to f on Day 3, from f to i on Day 5, and from i to l on Day 8. ■

The above example shows that a condensed static graph can present misleading information about the original temporal graph, and hence it is essential to keep the temporal information in the graphs. However, efficient and effective tools for studying temporal graphs are severely lacking. In this paper, we focus on the study of "shortest" paths in a temporal graph, as shortest paths are fundamental to the study of a graph and algorithms for computing shortest paths are essential building blocks of many advanced graph analysis algorithms (e.g., centrality computation, graph clustering, etc.).

Due to the presence of temporal information, different forms of "shortest" paths exist and each has its own meaning and significance. We define four types of paths in a temporal graph, collectively we call them **minimum temporal paths**, as they give the minimum value for different measures: (1) **earliest-arrival path** (i.e., a path that gives earliest arrival time starting from a source x to a target y); (2) **latest-departure path** (i.e., a path that gives latest departure time starting from x in order to reach y by a given time); (3) **fastest path** (i.e., the path by which one goes from x to y with the minimum elapsed time); and (4) **shortest path** (i.e., the path that is shortest from x to y in terms of overall traversal time needed on the edges).

Note that a shortest path may not necessarily be a fastest path (e.g., in a traffic network the shortest path from x to y may have a lot of traffic lights, while a highway is longer but is the fastest way to go from x to y). Also, a fastest path may not be an earliest-arrival path (e.g., traveling from x to y may only take 1 hour at noon due to less traffic, but one may leave at 9 a.m., take 2 hours to travel and arrive before noon).

Due to the additional temporal information, computing temporal paths and their "time-distance" poses new challenges. For example, the greedy strategy used to compute shortest paths in a static graph (e.g., by Dijkstra's algorithm) is based on the property that a subpath of a shortest path is also shortest, which is not necessarily true when computing any of the four minimum temporal paths.

We formally define various concepts of temporal graphs, and the four types of minimum temporal paths. We investigate useful properties of temporal paths to address the challenges of computing minimum temporal paths discussed above, and propose efficient one-pass algorithms, as well as a graph transformation approach, to compute minimum temporal paths. We evaluate the performance of our algorithms over a wide spectrum of real world temporal graphs, showing that they significantly outperform existing algorithms. In addition, we also discuss the applications of minimum temporal paths, and demonstrate that analytic results obtained directly from temporal graphs can be dramatically different from those obtained from static graphs, and hence carry additional important information or even the real accurate information about the temporal data.

Paper organization. Section 2 defines notions and notations of temporal graphs. Section 3 formally defines the minimum temporal paths. Section 4 presents the one-pass algorithms and Section 5 offers an alternative approach. Section 6 discusses some applications. Section 7 reports experimental results. Section 8 discusses related work and Section 9 gives our concluding remarks.

2. NOTATIONS OF TEMPORAL GRAPHS

Let $G = (V, E)$ be a temporal graph, where V is the set of vertices of G and E is the set of edges of G. An edge $e \in E$ is a quadruple (u, v, t, λ), where $u, v \in V$, t is the **starting time**, λ is the **traversal time** to go from u to v starting at time t, and $t + \lambda$ is the **ending time**. We denote the starting time of e by $t(e)$ and the traversal time of e by $\lambda(e)$. For simplicity of discussion, we assume that $\lambda(e) \neq 0$ for all $e \in E$, but note that our algorithms can be extended to handle the case where $\exists e \in E$ that $\lambda(e) = 0$.

If edges are undirected, then the starting time and traversal time of an edge are the same from u to v as from v to u. We focus on directed temporal graphs in this paper since an undirected edge can be modeled by two bi-directed edges.

In Section 1, we give a list of temporal graphs from a wide spectrum of applications, we select a few of them to illustrate what temporal information is modeled as follows:

- Phone call or Short Message Service networks: each vertex represents a person (or simply a mobile device), and an edge (u, v, t, λ) indicates that vertex u calls or sends a message to vertex v at time t, and the connection time is λ.
- Social networks (e.g., Facebook, Twitter): each vertex models a person (or an organization, etc.), and an edge (u, v, t, λ) can be an interaction between u and v (e.g., u follows v) at time t which takes time λ.
- Flight graphs: each vertex represents a location, and an edge (u, v, t, λ) is a flight from u to v departing at time t and the flight duration is λ.

Note that in all the above examples, vertex u may communicate with vertex v at multiple time instances and in fact, the number of temporal edges from u to v can be large for all of the above graphs. We denote the set of temporal edges from u to v in G by $\Pi(u, v)$, and the number of temporal edges from u to v in G by $\pi(u, v)$, i.e., $\pi(u, v) = |\Pi(u, v)|$. We also define the maximum number of temporal edges from u to v, for any u and v in G, by $\pi = \max\{\pi(u, v) : (u, v) \in (V \times V)\}$. The value of π can be large for some real world temporal graphs (e.g., in one of the temporal graphs used in our experiments, $\pi = 1074$).

In a temporal graph $G = (V, E)$, given two temporal edges $e_1 = (u_1, v_1, t_1, \lambda_1) \in E$ and $e_2 = (u_2, v_2, t_2, \lambda_2) \in E$, we have $e_1 = e_2$ iff $(u_1 = u_2 \wedge v_1 = v_2 \wedge t_1 = t_2 \wedge \lambda_1 = \lambda_2)$. If we condense temporal edges into static edges, we obtain the corresponding **static graph** $G_s = (V_s, E_s)$ of G, where $V_s = V$ and $E_s = \{(u, v) : (u, v, t, \lambda) \in E\}$, that is, the condensation removes all temporal information from the edges in E and combines all edges with the same start and end vertices into a single edge.

We define the number of vertices in G and G_s as $n = |V| = |V_s|$, and the number of edges in G as $M = |E|$ and in G_s as $m = |E_s|$. We define the set of **out-neighbors** of a vertex u in G or G_s as $\Gamma_{out}(u, G)$=$\Gamma_{out}(u, G_s)$=$\{v : (u, v, t, \lambda) \in E\} = \{v : (u, v) \in E_s\}$. We define the **out-degree** of u in G as $d_{out}(u, G) = \sum_{v \in \Gamma_{out}(u,G)} \pi(u, v)$, and in G_s as $d_{out}(u, G_s) = |\Gamma_{out}(u, G_s)|$. The **in-neighbors** and **in-degree** of a vertex u in G or G_s are defined symmetrically, i.e., $\Gamma_{in}(u, G) = \Gamma_{in}(u, G_s) = \{v : (v, u, t, \lambda) \in E\} = \{v : (v, u) \in E_s\}$, $d_{in}(u, G) = \sum_{v \in \Gamma_{in}(u,G)} \pi(v, u)$, and $d_{in}(u, G_s) = |\Gamma_{in}(u, G_s)|$.

Figure 1(a) shows a temporal graph G and its corresponding static graph G_s is shown in Figure 1(b). For simplicity, we set $\lambda = 1$ for all edges. We have $\Gamma_{out}(a, G) = \Gamma_{out}(a, G_s) = \{b, c, f, i\}$, and $\Gamma_{in}(b, G) = \Gamma_{in}(b, G_s) = \{a\}$. Since $\Pi(a, b) = \{(a, b, 1, 1), (a, b, 2, 1)\}$, we have $\pi(a, b) = 2$, $d_{in}(b, G) = 2$ and $d_{in}(b, G_s) = 1$. Similarly, we have $d_{out}(a, G) = 5$ and $d_{out}(a, G_s) = 4$.

3. DEFINITIONS OF TEMPORAL PATHS

A **temporal path** P in a temporal graph G is a sequence of vertices $P = \langle v_1, v_2, \ldots, v_k, v_{k+1} \rangle$, where $(v_i, v_{i+1}, t_i, \lambda_i) \in E$ is the i-th temporal edge on P for $1 \leq i \leq k$, and $(t_i + \lambda_i) \leq t_{i+1}$ for $1 \leq i < k$. Note that for the last edge $(v_k, v_{k+1}, t_k, \lambda_k)$ on P, we do not put a constraint on $(t_k + \lambda_k)$ since t_{k+1} is not defined for

the path P. In fact, $(t_k + \lambda_k)$ is the **ending time** of P, denoted by $end(P)$. We also define the **starting time** of P as $start(P) = t_1$. We define the **duration** of P as $dura(P) = end(P) - start(P)$, and the **distance** of P as $dist(P) = \sum_{i=1}^{k} \lambda_i$.

The following example illustrates the concepts of temporal path.

EXAMPLE 2. *An example of a temporal path is* $P = \langle(a, f, 3, 1), (f, i, 5, 1), (i, l, 8, 1)\rangle$ *in the temporal graph G in Figure 1(a). We have* $start(P) = 3$, $end(P) = 8 + 1 = 9$, $dura(P) = 9 - 3 = 6$ *and* $dist(P) = 1 + 1 + 1 = 3$.

The stating time of the temporal edges on P follows a chronological order, which is important for real world applications such as itinerary planning. For example, if we choose the edge $(a, i, 10, 1)$ *instead to go from a to i, though the duration and distance are shorter, we cannot reach the final destination l as explained in Example 1. Thus, the route* $(a, i, 10, 1)$ *cannot be used as a valid travel itinerary.* ■

In the following, we formally define a set of minimum temporal paths.

DEFINITION 1 (MINIMUM TEMPORAL PATHS). *Given a temporal graph G, a source vertex x and a target vertex y in G, and a time interval* $[t_\alpha, t_\omega]$, *let* $\mathbf{P}(x, y, [t_\alpha, t_\omega]) = \{P : P$ *is a temporal path from x to y such that* $start(P) \geq t_\alpha$, $end(P) \leq t_\omega\}$, *we define the following four types of temporal paths from x to y within* $[t_\alpha, t_\omega]$ *that have the minimum value for different measures, thus collectively called* ***minimum temporal paths****:*

Earliest-arrival path*:* $P \in \mathbf{P}(x, y, [t_\alpha, t_\omega])$ *is an earliest-arrival path if* $end(P) = \min\{end(P') : P' \in \mathbf{P}(x, y, [t_\alpha, t_\omega])\}$.

Latest-departure path*:* $P \in \mathbf{P}(x, y, [t_\alpha, t_\omega])$ *is a latest-departure path if* $start(P) = \max\{start(P') : P' \in \mathbf{P}(x, y, [t_\alpha, t_\omega])\}$.

Fastest path*:* $P \in \mathbf{P}(x, y, [t_\alpha, t_\omega])$ *is a fastest path if* $dura(P) = \min\{dura(P') : P' \in \mathbf{P}(x, y, [t_\alpha, t_\omega])\}$.

Shortest path*:* $P \in \mathbf{P}(x, y, [t_\alpha, t_\omega])$ *is a shortest path if* $dist(P) = \min\{dist(P') : P' \in \mathbf{P}(x, y, [t_\alpha, t_\omega])\}$.

Note that if a time interval $[t_\alpha, t_\omega]$ is not explicitly specified for the minimum temporal paths, then it is simply taken as $[t_\alpha = 0, t_\omega = \infty]$. However, we may not be always interested in the entire temporal history of the graph and hence allowing users to specify $[t_\alpha, t_\omega]$ gives higher flexibility and applicability.

The concept of temporal path was introduced in [10]. Later a number of different types of paths were proposed [1, 11, 12, 14, 15, 16, 17, 21] based on the concept of temporal path. However, many of the existing path definitions are rather ad hoc, model incomplete information, and/or use over-complicated notations. There is no single work that studies all aspects or defines all types of minimum temporal paths that we study in this paper. The most complete existing definitions on minimum temporal paths were proposed in [21], which include three out of four types of paths we study here (i.e., earliest-arrival, fastest, and shortest paths). Compared with [21], our temporal path problems are more general: first, in [21] the traversal time λ is the same for any edge (u, v), while in our definition λ can be different when an edge has a different starting time (which is common such as for flight duration, phone call duration, etc.); second, their definition and algorithm for shortest paths can only count the number of hops, while our definition and algorithm allow edges to have either a traversal time or a weight.

Problem definition: single-source minimum temporal paths (SSMTP). Given a temporal graph $G = (V, E)$, a vertex x in V, and a time interval $[t_\alpha, t_\omega]$, the problem of **SSMTP** is to find: (1) the earliest-arrival path from x to every $v \in V$, or (2) the latest-departure path from every $v \in V$ to x, or (3) the fastest path from x to every $v \in V$, or (4) the shortest path from x to every $v \in V$, respectively, within the time interval $[t_\alpha, t_\omega]$.

Let P be a minimum temporal path to be computed. For simplicity of discussion, in the presentation of our algorithms for computing SSMTP, we only report: (1) earliest-arrival time $end(P)$, or (2) latest-departure time $start(P)$, or (3) duration of the fastest path $dura(P)$, or (4) distance of the shortest path $dist(P)$, respectively. We note that the algorithms can be straightforwardly extended to report the corresponding path P.

The following lemmas give some properties of minimum temporal paths (some examples for each case are given in [19]).

LEMMA 1. *A prefix-subpath of an earliest-arrival path may not be an earliest-arrival path.*

LEMMA 2. *A postfix-subpath of a latest-departure path may not be a latest-departure path.*

LEMMA 3. *A subpath of a fastest path may not be a fastest path.*

LEMMA 4. *A subpath of a shortest path may not be a shortest path.*

Lemmas 1-4 highlight the challenges of computing minimum temporal paths, as Dijkstra's greedy strategy cannot be directly applied to compute minimum temporal paths.

4. ONE-PASS ALGORITHMS FOR COMPUTING MINIMUM TEMPORAL PATHS

In this section, we present efficient one-pass algorithms for computing single-source minimum temporal paths.

4.1 Stream Representation of a Temporal Graph

Before we present the one-pass algorithms, we first describe the data stream representation of a temporal graph.

The **edge stream** representation of a temporal graph G is simply a sequence of all edges in G that come in the order of the time each edge is created/collected (i.e., the edges are ordered according to their starting time). If two temporal edges are created/collected at the same time, their ordering can be arbitrary. For example, if G has the following edges, $\{(v_1, v_2, 2, 5), (v_2, v_4, 4, 1), (v_3, v_2, 1, 1)\}$, then the edge stream of G appears as follows: $(v_3, v_2, 1, 1)$, $(v_1, v_2, 2, 5)$, $(v_2, v_4, 4, 1)$. The edge stream is a natural format with which a temporal graph is generated and collected, e.g., the communication logs captured by telecom operators over time, or the temporal user behavior captured by social networking sites over time.

The following lemma shows a property of a temporal path in connection with the edge stream representation.

LEMMA 5. *Let* $P = \langle v_1, v_2, \ldots, v_k, v_{k+1}\rangle$ *be a temporal path in G, where* $e_i = (v_i, v_{i+1}, t_i, \lambda_i) \in E$ *is the i-th temporal edge on P for* $1 \leq i \leq k$. *For any* e_i *and* e_j *on P, if* $i < j$, *then* e_i *comes before* e_j *(i.e.,* e_i *is ordered before* e_j*) in the edge stream of G.*

PROOF. By the definition of temporal path, we have $(t_i + \lambda_i) \leq t_{i+1}$ for $1 \leq i < k$, and hence $t_{i+1} > t_i$ as $\lambda_i > 0$. Thus, the starting times of e_1, e_2, ..., e_k are in strictly ascending order, and hence e_i comes before e_j in the edge stream of G. □

4.2 Earliest-Arrival Paths

In this subsection, we present our algorithm for computing the earliest-arrival time from a source vertex x to every vertex in a temporal graph G within the time interval $[t_\alpha, t_\omega]$.

Algorithm 1: Computing earliest-arrival time

Input : A temporal graph $G = (V, E)$ in its edge stream representation, source vertex x, time interval $[t_\alpha, t_\omega]$
Output : The earliest-arrival time from x to every vertex $v \in V$ within $[t_\alpha, t_\omega]$

```
1 Initialize t[x] = t_α, and t[v] = ∞ for all v ∈ V \ {x};
2 foreach incoming edge e = (u, v, t, λ) in the edge stream do
3     if t + λ ≤ t_ω and t ≥ t[u] then
4         if t + λ < t[v] then
5             t[v] ← t + λ;
6     else if t ≥ t_ω then
7         Break the for-loop and go to Line 8;
8 return t[v] for each v ∈ V;
```

The classic Dijkstra's algorithm for computing single-source shortest paths is based on the fact that the prefix-subpath of a shortest path is also a shortest path. However, according to Lemma 1, the prefix-subpath of an earliest-arrival path may not be an earliest-arrival path. This seems to imply that the greedy strategy to grow the shortest paths that is applied in Dijkstra's algorithm cannot be applied to compute earliest-arrival paths, though the following observation shows otherwise.

LEMMA 6. *Let $\mathbf{P}$ be the set of earliest-arrival paths from x to a vertex v_k within the time interval $[t_\alpha, t_\omega]$. If $\mathbf{P} \neq \emptyset$, then there exists $P = \langle x, v_1, v_2, \dots, v_k \rangle \in \mathbf{P}$ such that every prefix-subpath, $P_i = \langle x, v_1, v_2, \dots, v_i \rangle$, is an earliest-arrival path from x to v_i within $[t_\alpha, t_\omega]$, for $1 \leq i \leq k$.*

PROOF. Given any earliest-arrival path $P \in \mathbf{P}$, if not every prefix-subpath in it is an earliest-arrival path, we can always construct a path $\hat{P}$ as follows. We traverse P in reverse order and find the first vertex v_i such that the corresponding prefix-subpath P_i is not an earliest-arrival path from x to v_i. Thus, there exists another path $\hat{P}_i$ that is an earliest-arrival path from x to v_i. We replace P_i in P by $\hat{P}_i$. The new path $\hat{P}$ is still a valid temporal path because $end(\hat{P}_i) < end(P_i)$. In addition, $\hat{P}$ is an earliest-arrival path from x to v_k (i.e., $\hat{P} \in \mathbf{P}$) because $end(\hat{P}) = end(P)$. This process continues until every prefix-subpath is an earliest-arrival path and the resulting $\hat{P}$ is in $\mathbf{P}$, which proves the lemma. □

Based on Lemma 6, we can apply the greedy strategy to grow the earliest-arrival paths in a similar way to Dijkstra's algorithm. However, this approach needs to use a minimum priority queue, resulting in an algorithm with $O(m \log \pi + m \log n)$ time and $O(M + n)$ space complexity [19], which is too expensive for processing temporal graphs with a large number of temporal edges.

Dijkstra's greedy strategy requires the entire graph to be present as random access to vertices and edges are needed. However, for temporal graphs, Lemma 5 implies that the input graph can be in the natural edge stream representation, and it is possible to compute the earliest-arrival paths with only one scan of the graph. We present our one-pass algorithm in Algorithm 1 and elaborate as follows.

We use an array $t[v]$ to keep the current earliest-arrival time from x to every vertex $v \in V$ that has been seen in the stream. According to Lemma 5, if there is a temporal path P from x to v so that all edges on P have been seen in the stream, then $t[v] = end(P) = t + \lambda$ as updated in Line 5. The condition "$t + \lambda < t[v]$" in Line 4 ensures that $t[v]$ will be updated with the smallest $end(P)$ for any P from x to v within the time interval $[t_\alpha, t_\omega]$.

We linearly scan G and for each incoming edge $e = (u, v, t, \lambda)$ in the stream, we check whether e meets the time constraint of a temporal path within $[t_\alpha, t_\omega]$, i.e., whether $t + \lambda \leq t_\omega$ and $t \geq t[u]$. If yes, we grow the temporal path by extending to v via the edge e. During the process, we update $t[v]$ when necessary as discussed earlier. The process terminates when we meet the first edge in the stream that has starting time greater than or equal to t_ω (Lines 6-7).

EXAMPLE 3. *Consider the temporal graph G in Figure 1(a), where we assume that the traversal time λ is 1 for all edges. Let a be the source vertex. We compute the earliest-arrival time from a to every vertex in G within the time interval $[1, 4]$.*

Initially, $t[a] = 1$, and $t[v] = \infty$ for all $v \in V \setminus \{a\}$. The first incoming edge is $(a, b, 1, 1)$, since it satisfies the conditions in Lines 3-4, we update $t[b] = 1 + 1 = 2$ in Line 5. The second edge is $(a, b, 2, 1)$, the condition in Line 4 is not satisfied. The next edge is $(g, j, 2, 1)$, since $t[g] = \infty$, the condition "$t \geq t[u] = t[g]$" in Line 3 is not met. Then, the edges $(b, g, 3, 1)$, $(b, h, 3, 1)$, and $(a, f, 3, 1)$ are followed, for which we update $t[g] = 4$, $t[h] = 4$, and $t[f] = 4$. After that the edge $(a, c, 4, 1)$ comes, which satisfies the condition in Line 6 and the process is terminated. It can be easily verified that we have obtained the correct earliest-arrival time from a to every vertex in G within the time interval $[1, 4]$. ■

The following lemma shows that when Algorithm 1 terminates, $t[v]$ correctly reports the earliest-arrival time from x to v.

LEMMA 7. *For any vertex $v \in V$, if the earliest-arrival path from x to v within the time interval $[t_\alpha, t_\omega]$ exists, then $t[v]$ returned by Algorithm 1 is the corresponding earliest-arrival time; otherwise, $t[v] = \infty$.*

PROOF. Suppose that the earliest-arrival path from x to v within $[t_\alpha, t_\omega]$ exists. Then, according to Lemma 6, there exists an earliest-arrival path from x to v, $P = \langle x = v_1, v_2, \dots, v_k, v_{k+1} = v \rangle$, such that every prefix-subpath of P is an earliest-arrival path from x to some vertex v_i on P. Let $t_e[v_i]$ be the earliest-arrival time from x to v_i, for $1 \leq i \leq k + 1$. Let $e_1, e_2, \dots, e_k$ be the edges on P, where $e_i = (v_i, v_{i+1}, t_i, \lambda_i)$ for $1 \leq i \leq k$. Then, we have $t_i \geq t_e[v_i]$ and $t_i + \lambda_i = t_e[v_{i+1}]$ for $1 \leq i \leq k$.

We prove that Algorithm 1 computes $t[v_i] = t_e[v_i]$, for $1 \leq i \leq k + 1$, by induction on i. When $i = 1$, $x = v_1$, $t[x] = t_e[x] = t_\alpha$ is initialized in Line 1 of Algorithm 1, and $t[x]$ will not be updated any more. When $i = 2$, obviously we have $t[v_2] = t_e[v_2] = t_1 + \lambda_1$ when we process e_1, and $t[v_2]$ will not be updated again due to the condition in Line 4. Now assume that for $i = j$, where $j < k + 1$, $t[v_j] = t_e[v_j] = t_{j-1} + \lambda_{j-1}$ when we process e_{j-1}. Consider $i = j + 1$ and we want to prove $t[v_{j+1}] = t_e[v_{j+1}]$. According to Lemma 5, e_j comes after e_{j-1} in the stream. Thus, when the algorithm scans e_j, we have the following two cases regarding the value of $t[v_{j+1}]$. (1) $t[v_{j+1}] = t_e[v_{j+1}]$. In this case, Line 5 will not be processed due to the condition in Line 4 and $t[v_{j+1}]$ gives the correct earliest-arrival time from x to v_{j+1}. (2) $t[v_{j+1}] > t_e[v_{j+1}]$. In this case, $t[v_{j+1}]$ is updated to $t_e[v_{j+1}] = t_j + \lambda_j$ in Line 5, and it will not be updated again due to the condition in Line 4. In both cases, we have $t[v_{j+1}] = t_e[v_{j+1}]$ and by induction, $t[v_i] = t_e[v_i]$ for $1 \leq i \leq k + 1$.

Finally, if the earliest-arrival path from x to v does not exist, then there is no temporal path from x to v and $t[v]$ remains to be ∞. □

The following theorem states our main result for earliest-arrival path computation.

THEOREM 1. *Algorithm 1 correctly computes the earliest-arrival time from a source vertex x to every vertex $v \in V$ within the time interval $[t_\alpha, t_\omega]$ using only one linear scan of the graph, $O(n+M)$ time and $O(n)$ space.*

PROOF. The correctness is proved in Lemma 7. The initialization in Line 1 takes $O(n)$ time. Every temporal edge in G is

Algorithm 2: Computing latest-departure time

Input : A temporal graph $G = (V, E)$ in reverse edge stream representation, target vertex x, time interval $[t_\alpha, t_\omega]$
Output : The latest-departure time from every vertex $v \in V$ to x within $[t_\alpha, t_\omega]$

```
1 Initialize t[x] = t_ω, and t[v] = −∞ for all v ∈ V \ {x};
2 foreach edge e = (u, v, t, λ) in the reverse edge stream do
3     if t ≥ t_α then
4         if t + λ ≤ t[v] then
5             if t > t[u] then
6                 t[u] ← t;
7     else
8         Break the for-loop and go to Line 9;
9 return t[v] for each v ∈ V;
```

Algorithm 3: Computing fastest-path duration (multi-passes)

Input : A temporal graph $G = (V, E)$ in its edge stream representation, source vertex x, time interval $[t_\alpha, t_\omega]$
Output : The duration of the fastest path from x to every vertex $v \in V$ within $[t_\alpha, t_\omega]$

```
1 Initialize f[x] = 0, and f[v] = ∞ for all v ∈ V \ {x};
2 Let S be the set of distinct starting time of the out-edges of x within
  [t_α, t_ω], i.e.,
  S = {t(e) : e is an out-edge of x, t(e) ≥ t_α, t(e) + λ(e) ≤ t_ω};
3 foreach t ∈ S do
4     Call Algorithm 1 with input G, x, and time interval [t, t_ω];
      let t[v] be the earliest-arrival time from x to v returned by
      Algorithm 1, then update f[v] ← min{f[v], t[v] − t};
5 return f[v] for each v ∈ V;
```

scanned at most once and it takes $O(1)$ time to process every edge. Thus, the overall time complexity of Algorithm 1 is $O(n + M)$. We do not keep the edges, but use $O(n)$ space to keep $t[v]$ for each $v \in V$. And clearly, the algorithm takes at most one linear scan of the edge stream. □

4.3 Latest-Departure Paths

Next we present a one-pass algorithm for computing the latest-departure time from every vertex to a target vertex x in G.

We present the algorithm in Algorithm 2, which is essentially symmetric to Algorithm 1 as we now scan the edge stream in reverse order. The other differences can be easily observed from the definition of the latest-departure paths, and hence we omit the detailed algorithm description here.

Similar to the computation of earliest-arrival time, the following lemma shows that we can correctly compute latest-departure time (the proof is similar to that of Lemma 7 and hence omitted).

LEMMA 8. *For any vertex $v \in V$, if the latest-departure path from v to x within $[t_\alpha, t_\omega]$ exists, then $t[v]$ returned by Algorithm 2 is the corresponding latest-departure time; otherwise, $t[v] = -\infty$.*

The following theorem states our main result for latest-departure path computation (the proof is similar to that of Theorem 1).

THEOREM 2. *Algorithm 2 correctly computes the latest-departure time from every vertex $v \in V$ to a target vertex x within the time interval $[t_\alpha, t_\omega]$ using only one linear scan of the graph, $O(n+M)$ time and $O(n)$ space.*

4.4 Fastest Paths

We now present our algorithm for computing the duration of the fastest path from a source vertex x to every vertex in G.

A naive way to find the fastest path from x to a vertex v in G is to find all temporal paths from x to v, and then pick the one with the minimum duration. However, there may exist exponentially many temporal paths from x to v. Thus, effective pruning of search space is needed, and the following lemma is useful for this purpose.

LEMMA 9. *Let $\mathbf{P}$ be the set of temporal paths from x to v with the same starting time t. Then, $P \in \mathbf{P}$ is a fastest path from x to v starting at t if P is an earliest-arrival path from x to v starting at t.*

PROOF. The proof follows directly from the definitions of earliest-arrival path and fastest path. □

Lemma 9 implies that we can compute the fastest path from x by finding the earliest-arrival path starting at every distinct time instance from x in the time interval $[t_\alpha, t_\omega]$. Based on this observation, we design our algorithm as shown in Algorithm 3.

For each distinct starting time $t \in S$, where S is defined in Line 2, the algorithm calls Algorithm 1 to compute the earliest-arrival time from x to each $v \in V \setminus \{x\}$, within the time interval $[t, t_\omega]$. Then, the minimum duration of the earliest-arrival paths starting at different starting time is returned as the duration of the fastest path.

We give the correctness and complexity of Algorithm 3 below.

THEOREM 3. *Algorithm 3 correctly computes the duration of the fastest path from a source vertex x to every vertex $v \in V$ within $[t_\alpha, t_\omega]$ in $O(|S|(n + M))$ time and $O(n)$ space, using $|S|$ linear scans of the graph, where $S = \{t(e) : e$ is an out-edge of x, $t(e) \geq t_\alpha, t(e) + \lambda(e) \leq t_\omega\}$.*

PROOF. The correctness follows from Lemma 9 as Algorithm 3 calls Algorithm 1 to compute all earliest-arrival time from x to v starting at every distinct starting time from x within $[t_\alpha, t_\omega]$.

The time complexity follows from the number of times Algorithm 1 is called, while we need $O(n)$ space to keep $t[v]$ and $f[v]$ for each $v \in V$. The algorithm scans G once for each of the $|S|$ calls of Algorithm 1. □

4.4.1 One-Pass Algorithm with Better Time Bound

In Algorithm 3, there can be potentially much redundant processing due to multiple invocations of Algorithm 1. Every time when Algorithm 1 is invoked, we need to scan the graph once. Thus, we want to examine whether we can avoid scanning the graph multiple times and eliminate the redundant processing. To this end, we design a one-pass algorithm as given in Algorithm 4.

The algorithm uses a sorted list for each vertex v, denoted by L_v, to keep the earliest-arrival time from the source vertex x to v at different starting time that may potentially give the duration of the fastest path from x to v. For every element $(s[v], a[v])$ in L_v, defined in Line 2, if there exists another element $(s'[v], a'[v])$ in L_v, where $s'[v] > s[v]$ and $a'[v] \leq a[v]$, or $s'[v] = s[v]$ and $a'[v] < a[v]$, we say that $(s'[v], a'[v])$ **dominates** $(s[v], a[v])$, and call $(s[v], a[v])$ a **dominated element** in L_v.

The following lemma shows that a dominated element can be safely pruned from L_v.

LEMMA 10. *Given two elements $(s[v], a[v])$ and $(s'[v], a'[v])$ in L_v for any vertex $v \in V$, if $(s'[v], a'[v])$ dominates $(s[v], a[v])$ in L_v, then $(s[v], a[v])$ can be removed from L_v without affecting the computation of the duration of the fastest path from x to any vertex in V.*

PROOF. Since both $(s[v], a[v])$ and $(s'[v], a'[v])$ are in L_v, this implies that there is one temporal path P starting from x at time

$s[v]$ and arriving at v at time $a[v]$, and another temporal path P' starting from x at time $s'[v]$ and arriving at v at time $a'[v]$. Let P_w be a fastest path from x to any vertex $w \in V$ such that P is a prefix-subpath of P_w. Let P'_w be the path obtained by replacing P with P' in P_w. Since $a'[v] \le a[v]$, P'_w is still a valid temporal path. If $s'[v] > s[v]$, then P'_w is a temporal path with a smaller duration than P_w, which contradicts to the fact that P_w is a fastest path. If $s'[v] = s[v]$, then P'_w also is a fastest path from x to w. In both cases, if we have $(s'[v], a'[v])$, then we do not need $(s[v], a[v])$ in the computation of the duration of the fastest path from x to any vertex $w \in V$. □

In Algorithm 4, every time after removing dominated elements in L_v, we have the following property regarding L_v.

LEMMA 11. *Each time after Line 16 of Algorithm 4 is executed, for any two elements $(s[v], a[v])$ and $(s'[v], a'[v])$ in L_v, either (1) $s'[v] > s[v]$ and $a'[v] > a[v]$, or (2) $s[v] > s'[v]$ and $a[v] > a'[v]$.*

PROOF. First, $s[v] \neq s'[v]$ since the condition in Line 12 ensures that no two elements in L_v will have the same "$s[v]$" value. Then, assume that $s'[v] > s[v]$, then suppose to the contrary that $a'[v] \le a[v]$, in this case $(s[v], a[v])$ is dominated by $(s'[v], a'[v])$ and is removed in Line 16. Thus, $a'[v] > a[v]$. Case (2) is symmetric. □

We now discuss other details of Algorithm 4. We scan the edge stream of the input graph once. For each incoming edge $e = (u, v, t, \lambda)$, we check whether the earliest-arrival paths from x to u can be extended to v via e within $[t_\alpha, t_\omega]$ (Line 5). If yes, we pick the path from x to u with the largest arrival time that is at or before t (Line 9), which also has the largest starting time according to Lemma 11 and hence potentially gives the minimum duration of the resultant path.

We then update L_v as follows. If there is already a record with the same $s[v]$ in L_v, we update the corresponding $a[v]$ in L_v if the current $a[v]$ (computed in Line 11) is smaller (which means that the current $(s[v], a[v])$ pair dominates the old pair). Otherwise, we insert the new record $(s[v], a[v])$ into L_v. Then, we apply Lemma 10 to prune dominated elements in L_v. During the process, we use $f[v]$ to record the final fastest-path duration from x to v. If the minimum duration $f[v]$ changes, we update the value of $f[v]$ in Lines 17-18.

The following theorem gives our main result for fastest path computation.

THEOREM 4. *Let $S = \{t(e) : e$ is an out-edge of x, $t(e) \ge t_\alpha$, $t(e) + \lambda(e) \le t_\omega\}$, $d_{max} = \max\{d_{in}(v, G) : v \in V\}$, and $c = \min\{|S|, d_{max}\}$. Algorithm 4 correctly computes the duration of the fastest path from a source vertex x to every vertex $v \in V$ within the time interval $[t_\alpha, t_\omega]$ using only one linear scan of the graph, $O(n + M \log c)$ time and $O(\min\{n|S|, n + M\})$ space.*

PROOF. We first prove the correctness. Suppose that the fastest path from x to v within $[t_\alpha, t_\omega]$ exists. Let the fastest path starts from x at time t_x, and arrives at v at time t_y. Then, this is also an earliest-arrival path from x to v within the time interval $[t_x, t_y]$. By Lemma 6, there exists an earliest-arrival path P from x to v such that every prefix-subpath of P is an earliest-arrival path from x to some vertex on P. Let $P = \langle x = v_1, v_2, \ldots, v_k, v_{k+1} = v \rangle$. Let $t_e[v_i]$ be the earliest-arrival time from x to v_i within $[t_x, t_y]$, for $1 \le i \le k + 1$. Let e_1, e_2, …, e_k be the edges on P, where $e_i = (v_i, v_{i+1}, t_i, \lambda_i)$ for $1 \le i \le k$. Then, we have $t_i \ge t_e[v_i]$ and $t_i + \lambda_i = t_e[v_{i+1}]$ for $1 \le i \le k$.

We only need to show that the pair (t_x, t_y) is inserted into L_v, so that $f[v]$ is updated to $t_y - t_x$ in Line 18. We prove that $(t_x, t_e[v_i])$ is inserted into L_{v_i}, for $1 \le i \le k + 1$, by induction on i. When $i = 1$, $x = v_1$, (t_x, t_x) is inserted into L_x in Line 8. When $i = 2$, we insert $(s[v_2], a[v_2]) = (t_x, t_e[v_2] = t_1 + \lambda_1)$ into L_{v_2} when we process e_1. Now assume that for $i = j$, where $j < k + 1$, $(t_x, t_e[v_j])$ is inserted into L_{v_j} when we process e_{j-1}. Consider $i = j + 1$ and we want to prove that $(t_x, t_e[v_{j+1}])$ is inserted into $L_{v_{j+1}}$. According to Lemma 5, e_j comes after e_{j-1} in the stream. Thus, when the algorithm scans e_j, by Lemma 11 we obtain $a'[v_j] = t_e[v_j]$ in Line 9, which also means $s'[v_j] = t_x$. This gives $(s[v_{j+1}], a[v_{j+1}]) = (t_x, t_e[v_{j+1}] = t_j + \lambda_j)$. Thus, $(t_x, t_e[v_{j+1}])$ will be inserted into $L_{v_{j+1}}$. Thus, by induction, (t_x, t_y) will be inserted into L_v.

Next, we analyze the complexity. It is clear that the algorithm takes at most one linear scan of the edge stream. The initialization in Lines 1-3 takes $O(n)$ time. For each $v \in V$, the size of L_v is bounded by $\min\{|S|, d_{in}(v, G)\} \le c = \min\{|S|, d_{max}\}$. Searching and updating L_v take $O(\log c)$ time. The total time of removing dominated elements from L_v in Line 16 is bounded by $O(d_{in}(v, G))$. Thus, the total time for removing dominated elements from L_v for all $v \in V$ is $O(M)$. Summing up, the total time complexity is $O(n + M \log c)$. The space requirement is bounded by the total size of L_v, which is given by $O(\min(n|S|, n + M))$. Note that $|S|$ (and hence also c) is a small number in practice. □

Algorithm 4: Computing fastest-path duration (one-pass)

Input : A temporal graph $G = (V, E)$ in its edge stream representation, source vertex x, time interval $[t_\alpha, t_\omega]$
Output : The duration of the fastest path from x to every vertex $v \in V$ within $[t_\alpha, t_\omega]$

```
1  foreach v ∈ V do
2      Create a sorted list for v, L_v, where an element of L_v is a pair
       (s[v], a[v]) in which s[v] is the starting time of a path P from x
       to v, and a[v] is the time that the path P arrives at v and is used as
       the key for ordering in L_v; initially, L_v is empty;
3  Initialize f[x] = 0, and f[v] = ∞ for all v ∈ V \ {x};
4  foreach incoming edge e = (u, v, t, λ) in the edge stream do
5      if t ≥ t_α and t + λ ≤ t_ω then
6          if u = x then
7              if (t, t) ∉ L_x then
8                  Insert (t, t) into L_x;
9          Let (s'[u], a'[u]) be the element in L_u where
           a'[u] = max{a[u] : (s[u], a[u]) ∈ L_u, a[u] ≤ t};
10         s[v] ← s'[u];
11         a[v] ← t + λ;
12         if s[v] is in L_v then
13             Update the corresponding a[v] in L_v;
14         else
15             Insert (s[v], a[v]) into L_v;
16         Remove dominated elements in L_v;
17         if a[v] − s[v] < f[v] then
18             f[v] = a[v] − s[v];
19     else if t ≥ t_ω then
20         Break the for-loop and go to Line 21;
21 return f[v] for each v ∈ V;
```

4.4.2 Linear-Time Algorithm for Specific Cases

The log factor in the time complexity of Algorithm 4 is due to searching and inserting in L_v. This log factor can be removed for processing some real-world temporal graphs in which the value of λ is the same for all edges, or edges in the edge stream do not just arrive in the order of their starting time but are also further ordered by their ending time. To obtain linear-time complexity for processing such graphs, we make the following changes in Algorithm 4.

1. For each $v \in V$, replace the sorted list L_v by a queue Q_v.

2. In Line 9 when we obtain $(s'[u], a'[u])$, which is the element with largest arrival time that is less than or equal to t, we also remove all the elements before it in Q_u. For example, assume that $(3, 12)$, $(4, 14)$, and $(5, 16)$ are in Q_u; when the edge $e = (u, v, 15, 3)$ comes in, we linear scan Q_u and find that $(4, 14)$ is the element with largest arrival time and $14 < t(e) = 15$. Since the edges are in ascending order of their starting time, the edges that have not been scanned in the stream will have starting time at least 15. Thus, we can remove $(3, 12)$ from Q_u as it will never be used to extend any temporal path from u any more.

3. For the insertion of a new element $(s[v], a[v])$ into Q_v, we only need to compare $(s[v], a[v])$ with the element $(s'[v], a'[v])$ at the tail of Q_v, since for the graphs we handle here, we have $a[v] \geq a'[v]$ and other elements in Q_v are not dominated by $(s'[v], a'[v])$ (and hence not by $(s[v], a[v])$). If either $(s[v], a[v])$ or $(s'[v], a'[v])$ is dominated by the other, we only keep the element that is not dominated as the tail of Q_v; otherwise, $(s[v], a[v])$ is added after $(s'[v], a'[v])$ in Q_v. For example, assume that $(2, 14)$, $(3, 15)$, and $(5, 16)$ are currently in Q_v; when we try to insert $(4, 18)$ into Q_v, we find that $(4, 18)$ is dominated by $(5, 16)$. Thus, we do not need to insert $(4, 18)$ into Q_v.

Now we analyze the time complexity. Every time when we linear scan Q_u in Line 9, it takes $O(k + 1)$ time, where k is the number of elements being removed during the search. Let k_i be the number of removed elements from Q_u in the i-th search of Q_u. We have $O(\sum_i (k_i + 1)) = O(\sum_i k_i + d_{out}(u, G)) = O(d_{in}(u, G) + d_{out}(u, G))$. Thus, the total time for searching Q_u and removing elements from Q_u for all the vertices is $O(M)$. The total time for inserting new elements into Q_u for all the vertices is clearly bounded by $O(M)$. Thus, the total time complexity is $O(n + M)$.

4.5 Shortest Paths

We present an efficient one-pass algorithm for computing the shortest path from a source vertex x to every vertex in G.

In a static graph, the subpath of a shortest path is also shortest. However, this is not true in a temporal graph according to Lemma 4. Moreover, there may not exist a shortest path $P = \langle x, v_1, v_2, \ldots, v_k \rangle$ such that every prefix-subpath, $P_i = \langle x, v_1, v_2, \ldots, v_i \rangle$, is a shortest path from x to v_i, for $1 \leq i \leq k$. Thus, we cannot apply Dijkstra's greedy strategy directly to compute shortest paths in a temporal graph. Rather, we need to consider different combinations of paths even though these paths may not be shortest, because a non-shortest subpath may grow into a shortest path.

Certainly, we cannot consider all combinations of temporal paths from x to every vertex v and then take the one with minimum distance. Fortunately, a careful examination of temporal paths leads to the following lemma by which we can design an efficient strategy for computing shortest paths in a temporal graph.

LEMMA 12. *Let $P = \langle x = v_1, v_2, \ldots, v_k \rangle$ be a shortest path from x to v_k within the time interval $[t_\alpha, t_\omega]$. Then, for $1 \leq i \leq k$, every prefix-subpath $P_i = \langle x = v_1, v_2, \ldots, v_i \rangle$ is a shortest path from x to v_i within the time interval $[t_\alpha, end(P_i)]$.*

PROOF. Suppose to the contrary that within $[t_\alpha, end(P_i)]$, there exists another path P_i' from x to v_i with a shorter distance. Since $end(P_i') \leq end(P_i)$, we can concatenate $\langle v_i, \ldots, v_k \rangle$ to P_i' to obtain another temporal path from x to v_k, which is a shorter path than P, and contradicts to the fact that P is a shortest path. □

Algorithm 5: Computing shortest-path distance

```
Input  : A temporal graph G = (V, E) in its edge stream
         representation, source vertex x, time interval [t_α, t_ω]
Output : The distance of the shortest path from x to every vertex
         v ∈ V within [t_α, t_ω]
1  foreach v ∈ V do
2      Create a sorted list for v, L_v, where an element of L_v is a pair
       (d[v], a[v]) in which d[v] is the distance of a path P from x to v
       and is used as the key for ordering in L_v, and a[v] is the time that
       the path P arrives at v; initially, L_v is empty;
3  Initialize f[x] = 0, and f[v] = ∞ for all v ∈ V \ {x};
4  foreach incoming edge e = (u, v, t, λ) in the edge stream do
5      if t ≥ t_α and t + λ ≤ t_ω then
6          if u = x then
7              if (0, t) ∉ L_x then
8                  Insert (0, t) into L_x;
9          Let (d'[u], a'[u]) be the element in L_u where
           a'[u] = max{a[u] : (d[u], a[u]) ∈ L_u, a[u] ≤ t};
10         d[v] ← d'[u] + λ;
11         a[v] ← t + λ;
12         if a[v] is in L_v then
13             Update the corresponding d[v] in L_v;
14         else
15             Insert (d[v], a[v]) into L_v;
16         Remove dominated elements in L_v;
17         if d[v] < f[v] then
18             f[v] = d[v];
19     else if t ≥ t_ω then
20         Break the for-loop and go to Line 21;
21 return f[v] for each v ∈ V;
```

Based on Lemma 12, we can still use the greedy strategy by carefully maintaining the ending time of the paths, since there may be different shortest paths with different ending time. Furthermore, to eliminate redundant paths within the same time interval, we apply the following lemma.

LEMMA 13. *Given two temporal paths from x to v, P_1 and P_2, within $[t_\alpha, t_\omega]$, if $dist(P_1) \leq dist(P_2)$ and $end(P_1) \leq end(P_2)$, then we can safely prune P_2 in the computation of shortest paths from x to any vertex that pass through v within $[t_\alpha, t_\omega]$.*

PROOF. Let P be a shortest path from x to u such that P_2 is a prefix-subpath of P. Clearly, replacing P_2 with P_1 in P also gives a shortest path from x to u within $[t_\alpha, t_\omega]$. □

Applying the above lemmas in the classic Dijkstra's algorithm framework leads to an algorithm that takes $O(M \log \pi + M \log n)$ time and requires random access to the input graph [19]. Interestingly we find that we can apply Lemmas 12 and 13 into the framework of Algorithm 4, which requires only one linear scan of the input graph. We give our algorithm in Algorithm 5 and discuss the essential details as follows.

For each vertex v, the algorithm also uses a sorted list L_v. For every element $(d[v], a[v])$ in L_v, defined in Line 2, if there exists another element $(d'[v], a'[v])$ in L_v, where $d'[v] < d[v]$ and $a'[v] \leq a[v]$, or $d'[v] = d[v]$ and $a'[v] < a[v]$, we say that $(d'[v], a'[v])$ **dominates** $(d[v], a[v])$, and call $(d[v], a[v])$ a **dominated element** in L_v.

By Lemma 13, $(d[v], a[v])$ can be safely removed from L_v. The sorted list L_v will be used to obtain the shortest-path distance with different ending time (i.e., time arriving at v) by applying Lemma 12.

Similar to Algorithm 4, every time after Algorithm 5 removes dominated elements in L_v, we have the following property regarding L_v (proof omitted as it is similar to that of Lemma 11).

LEMMA 14. *Each time after Line 16 of Algorithm 5 is executed, for any two elements $(d[v], a[v])$ and $(d'[v], a'[v])$ in L_v, either (1) $a'[v]<a[v]$ and $d'[v]>d[v]$, or (2) $a[v]<a'[v]$ and $d[v]>d'[v]$.*

Lemma 14 implies that choosing the largest $a'[u]$ in Line 9 is equivalent to choose the smallest $d'[u]$, thus potentially giving the minimum distance of the resultant path. Other detailed steps in Algorithm 5 follow a similar procedure as in Algorithm 4, and hence we omit the details. The following theorem gives our main result for shortest path computation.

THEOREM 5. *Algorithm 5 correctly computes the distance of the shortest path from a source vertex x to every vertex $v \in V$ within the time interval $[t_\alpha, t_\omega]$ using only one linear scan of the graph, $O(n + M \log d_{max})$ time and $O(n + M)$ space, where $d_{max} = \max\{d_{in}(v, G) : v \in V\}$.*

PROOF. Suppose that a shortest path from x to v exists and let $P = \langle x = v_1, v_2, \ldots, v_{k+1} = v\rangle$ be this path, and let $e_i = (v_i, v_{i+1}, t_i, \lambda_i)$ is the i-th edge on P for $1 \leq i \leq k$. By a process similar to the proof of Theorem 4, we can prove by induction that the element $(\sum_{i=1}^{k} \lambda_i, t_k + \lambda_k)$ will be inserted into L_v by Algorithm 5 and $f[v]$ will be updated as $\sum_{i=1}^{k} \lambda_i$, which gives the correct shortest-path distance from x to v.

Now we analyze the complexity. The size of each L_v is bounded by $d_{in}(v, G)$. Searching and updating L_v thus take $O(\log d_{in}(v, G))$ time. Following a similar analysis as in the proof of Theorem 4 we obtain $O(n+M \log d_{in}(v, G)) = O(n+M \log d_{max})$ total time. The space requirement is determined by the total size of L_v, which is $O(n + M)$, though in practice the space requirement is significantly smaller. □

Finally, we can also achieve a one-pass linear-time algorithm for computing shortest-path distance in temporal graphs in which the value of λ is uniform or edges also arrive in the order of their ending time in the edge stream, as described in Section 4.4.2. We omit the details as they are similar to Section 4.4.2.

5. A GRAPH TRANSFORMATION APPROACH

In this subsection, we propose a graph transformation technique for computing the four types of minimum temporal paths.

We first present how to transform a temporal graph $G = (V, E)$ into a new graph $\tilde{G} = (\tilde{V}, \tilde{E})$. The construction of $\tilde{G}$ consists of the following two parts:

1. Vertex creation: for each vertex $v \in V$, create vertices in $\tilde{V}$ as follows:

   (a) Let $T_{in}(u, v) = \{t+\lambda : (u, v, t, \lambda) \in \Pi(u, v)\}$ where $u \in \Gamma_{in}(v, G)$, and $\mathbf{T}_{in}(v) = \bigcup_{u\in\Gamma_{in}(v,G)} T_{in}(u, v)$, i.e., $\mathbf{T}_{in}(v)$ is the set of distinct time instances at which edges from in-neighbors of v arrive at v.

   Create $|\mathbf{T}_{in}(v)|$ copies of v, each labeled with (v, t) where t is a distinct arrival time in $\mathbf{T}_{in}(v)$. Denote this set of vertices as $\tilde{V}_{in}(v)$, i.e., $\tilde{V}_{in}(v) = \{(v, t) : t \in \mathbf{T}_{in}(v)\}$. Sort vertices in $\tilde{V}_{in}(v)$ in descending order of their time, i.e., for any $(v, t_1), (v, t_2) \in \tilde{V}_{in}(v)$, (v, t_1) is ordered before (v, t_2) in $\tilde{V}_{in}(v)$ iff $t_1 > t_2$.

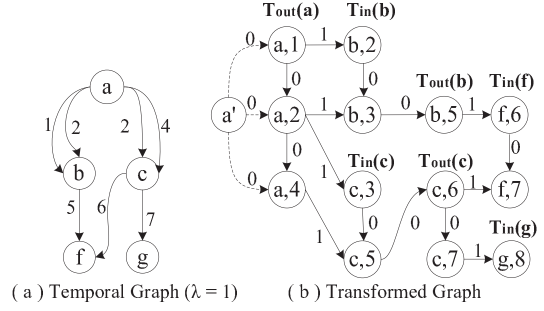

Figure 2: Graph transformation, from G in (a) to $\tilde{G}$ in (b)

   (b) Let $T_{out}(v, u) = \{t : (v, u, t, \lambda) \in \Pi(v, u)\}$ where $u \in \Gamma_{out}(v, G)$, and $\mathbf{T}_{out}(v) = \bigcup_{u\in\Gamma_{out}(v,G)} T_{out}(v, u)$. Create $|\mathbf{T}_{out}(v)|$ copies of v, each labeled with (v, t) where t is a distinct starting time in $\mathbf{T}_{out}(v)$. Denote this set of vertices as $\tilde{V}_{out}(v)$, i.e., $\tilde{V}_{out}(v) = \{(v, t) : t \in \mathbf{T}_{out}(v)\}$. Sort vertices in $\tilde{V}_{out}(v)$ in descending order of their time.

2. Edge creation: for each vertex $v \in V$, create edges in $\tilde{E}$ as follows:

   (a) For each vertex (v, t_{in}) in its order in $\tilde{V}_{in}(v)$, create a directed edge from (v, t_{in}) to $(v, t_{out}) \in \tilde{V}_{out}(v)$, where $t_{out} = \min\{t : (v, t) \in \tilde{V}_{out}(v), t \geq t_{in}\}$ and no edge from any other $(v, t'_{in}) \in \tilde{V}_{in}(v)$ to (v, t_{out}) has been created. Set the weight for each such edge as 0.

   (b) Let $\tilde{V}_{in}(v) = \{(v, t_1), (v, t_2), \ldots, (v, t_k)\}$. Create a directed edge from each (v, t_{i+1}) to (v, t_i) with weight 0, for $1 \leq i < k$. No edge is created if $k \leq 1$. Create edges for $\tilde{V}_{out}(v)$ in the same way.

   (c) For each temporal edge $e = (u, v, t, \lambda) \in E$, create a directed edge from $(u, t) \in \tilde{V}_{out}(u)$ to $(v, t + \lambda) \in \tilde{V}_{in}(v)$, with weight λ.

5.1 Earliest-Arrival Paths

We first discuss the computation of single-source earliest-arrival paths. To compute earliest-arrival paths from a source vertex x to every vertex $v \in V$, we further create a vertex x' in $\tilde{G}$ and a directed edge from x' to each vertex $(x, t) \in \tilde{V}_{out}(x)$ in $\tilde{G}$ with weight 0.

Then, we simply run the *breadth-first search* (**BFS**) algorithm in $\tilde{G}$ from the source vertex x'. During the process, if the time t of a vertex (v, t) is not in the time interval $[t_\alpha, t_\omega]$, we will stop the BFS from this vertex. The minimum time t of all visited vertices (v, t) in $\tilde{V}_{in}(v)$ is the earliest-arrival time from x to v in G.

We illustrate the graph transformation and how $\tilde{G}$ is used to compute the earliest-arrival time by the following example.

EXAMPLE 4. *Given a temporal graph G in Figure 2(a), where we assume that the traversal time λ is equal to 1 for all edges, the transformed graph $\tilde{G}$ is shown in Figure 2(b).*

Let a be the source vertex in G and thus we create a' as shown in $\tilde{G}$. Now let us start BFS from a' in $\tilde{G}$. In the 2nd step, we visit $(b, 2)$, $(b, 3)$, $(c, 3)$, and $(c, 5)$. Thus, the earliest-arrival time from a to b is 2, and from a to c is 3. In the 3rd step, we visit $(b, 5)$ and $(c, 6)$. In the 4th step, we visit $(f, 6)$, $(f, 7)$, and $(c, 7)$, from which

we obtain the earliest-arrival time from a to f as 6. Finally, we visit $(g, 8)$, and obtain the earliest-arrival time from a to g as 8. ■

5.2 Latest-Departure Paths

Similar to the computation of single-source earliest-arrival paths, we create a vertex x' in $\tilde{G}$ and a directed edge from each vertex $(x, t) \in \tilde{V}_{in}(x)$ to x' in $\tilde{G}$ with weight 0. Then, we perform a reverse BFS from x' in $\tilde{G}$. The maximum time t of all visited vertices (v, t) in $\tilde{V}_{out}(v)$ is the latest-departure time from every v to x in G.

Note that $\tilde{V}_{in}(a)$ does not exist in Figure 2(b) since there is no latest-departure path from any vertex to a in Figure 2(a). But we can easily compute the latest-departure time (or path) from every vertex to other target vertex, e.g., g, by a reverse BFS.

5.3 Fastest Paths

For the source vertex x in G, we create a vertex x' in $\tilde{G}$ and a directed edge from x' to each vertex $(x, t) \in \tilde{V}_{out}(x)$ in $\tilde{G}$ with weight 0. Let $S = \{(x, t) : (x, t) \in \tilde{V}_{out}(x), t_\alpha \leq t \leq t_\omega\}$, where elements in S are sorted in descending order of their time. From x', we first visit the vertex in S with largest time, say (x, t_1); then perform BFS from (x, t_1) to compute the earliest-arrival time $t[v]$ from x to every v and obtain the duration of this earliest-arrival path as $(t[v]-t_1)$. Then, we visit the vertex in S with second largest time, say (x, t_2); we conduct BFS from (x, t_2), but we will not continue the BFS from any vertex that has been visited previously. We repeat this process until all vertices in S are processed. The duration of the fastest path from x to every v in G is the minimum duration among all the earliest-arrival paths from x to v.

5.4 Shortest Paths

For the source vertex x in G, we create a vertex x' in $\tilde{G}$ and a directed edge from x' to each vertex $(x, t) \in \tilde{V}_{out}(x)$ in $\tilde{G}$ with weight 0. Then, we run Dijkstra's algorithm on $\tilde{G}$ from the source vertex x'. The minimum distance of the shortest-path from x' to each $(v, t) \in \tilde{V}_{in}(v)$ is the shortest-path distance from x to v in G.

5.5 Complexity Analysis

Assume that $n < M$ for a temporal graph G. From the graph transformation process, it is easy to see that both the number of vertices and edges in $\tilde{G}$ is bounded by $O(M)$. Thus, computing single source earliest-arrival paths and latest-departure paths takes $O(M)$ time since only one BFS in $\tilde{G}$ is required. For computing fastest paths in $\tilde{G}$, since we do not continue the BFS from any previously visited vertices, we visit each edge in $\tilde{G}$ only once during the entire process and hence the time complexity is also $O(M)$. Finally, for computing single source shortest paths in $\tilde{G}$, Dijkstra's algorithm uses $O(M \log M)$ time.

6. APPLICATIONS OF TEMPORAL PATHS

Shortest paths in a static graph have numerous important applications, in many of these applications, especially those related to network analysis such as centrality computation and clustering, the minimum temporal paths can be applied in place of shortest paths to analyze temporal networks [8]. In [15, 21], various definitions of "diameter" for temporal graphs were introduced based on the concepts of minimum temporal paths. In addition, the four types of minimum temporal paths can be naturally applied to many different types of metrics or applications defined based on the concepts of temporal paths [1, 10, 11, 12, 13, 14, 16, 17, 18].

Here, we briefly discuss two important applications of the minimum temporal paths.

6.1 Temporal Closeness Centrality

In the analysis of a graph, the closeness centrality of a vertex v is used to measure its importance in the graph. For example, the closeness of a person in a social network indicates the relative influence of this person in the network, or how easily it will take to spread information from this person to others in the network.

In a static graph $G_s = (V_s, E_s)$, the *closeness* of a vertex $v \in V_s$, denoted by $closeness(v)$, is defined as:

$$closeness(v) = \frac{1}{\sum_{u \in V_s \setminus \{v\}} dist(v, u)}. \quad (1)$$

In Equation 1, we use the shortest-path distance from v to u, i.e., $dist(v, u)$, to model the efficiency of the spread of information. In a temporal graph, the shortest-path distance must be replaced by the *temporal* shortest-path distance from v to u since the order of the time sequence on a temporal path defines the order of communication. However, there are also applications in which other types of minimum temporal paths, in particular fastest paths, can be applied to define meaningful temporal closeness centrality. For example, in flight scheduling or logistic/itinerary planning, the fastest path that one can travel from one place to another is often more important than the distance to be traveled. Thus, the closeness centrality should be defined based on the duration of the fastest path, i.e., $dist(v, u)$ in Equation 1 should be replaced by $dura(P_{v,u})$, where $P_{v,u}$ is a fastest path from v to u. This fastest-path based closeness of a vertex v indicates how fast, in terms of the amount of time needed, information from v can spread to other vertices.

To compute the exact closeness centrality value, we need to compute all pairs of paths which is too expensive for a large graph. However, approximation methods such as [6] can be naturally applied to compute temporal closeness centrality with guaranteed small error bound.

6.2 Top-k Nearest Neighbors

Top-k nearest neighbors have many applications such as graph clustering. We can use the four minimum temporal paths to define top-k nearest neighbors of a vertex. Formally, the top-k nearest neighbors of a vertex $x \in V$ is a subset K of V such that $\forall u \in K$ and $v \in V \setminus K$, $score(u) \geq score(v)$, where $score(u)$ is defined as (1) the earliest-arrival time, or (2) latest-departure time, or (3) duration of the fastest path, or (4) distance of the shortest path, from x to u, respectively.

Our algorithms can be straightforwardly modified to output only the top k vertices that have the highest score defined based on each of the minimum temporal paths.

7. EXPERIMENTAL RESULTS

We evaluate the performance of our algorithms and examine the usefulness of minimum temporal paths in this section. We ran all the experiments on a machine running Linux on an Intel 3.3GHz CPU and 16GB RAM.

Temporal graphs. We used 12 real temporal datasets in our experiments, which are from the Koblenz Large Network Collection (http://konect.uni-koblenz.de/), and we selected one large temporal graph from each of the following categories: `arxiv-HepPh` (`arxiv`) from the arxiv networks; `dblp-coauthor` (`dblp`) from the DBLP coauthor networks; `elec` from the network of English Wikipedia; `enron` from the email networks; `epin` from the trust and distrust network of Epinions; `facebook-wosn-links` (`fb`) from the facebook network; `flickr-growth` (`flickr`) from the social network of Flickr; `digg` from the reply network of the social news website Digg; `slashdot-threads` (`slash`) from

Table 1: Real temporal graphs ($\mathbf{K} = 10^3$)

Dataset	$\lvert V\rvert$	$\lvert E_s\rvert$	$\lvert E\rvert$	$d_{avg}(u, G_s)$	$d_{avg}(u, G)$	π	$\lvert T_G\rvert$
arxiv	28K	6297K	9194K	224.14	327.26	262	2337
dblp	1103K	8451K	11957K	7.66	10.84	38	70
elec	8K	104K	107K	12.50	12.90	5	101012
enron	87K	320K	1135K	3.67	13.01	1074	213218
epin	132K	841K	841K	6.38	6.38	1	939
fb	64K	817K	1270K	12.82	19.92	2	736675
flickr	2303K	33140K	33140K	14.39	14.39	1	134
digg	30K	85K	86K	2.80	2.84	25	82641
slash	51K	130K	140K	2.55	2.74	17	89862
conflict	118K	2054K	2918K	17.40	24.71	562	273909
growth	1871K	39953K	39953K	21.36	21.36	1	2198
youtube	3224K	9377K	12224K	2.91	3.80	2	203

Table 2: Running time in seconds (random sources)

	Earliest-arrival		Latest-departure		Fastest			Shortest		
	1-pass	Xuan	1-pass	Xuan	1-pass	Trans	Xuan	1-pass	Trans	Xuan
arxiv	0.0109	0.1636	0.0120	0.1685	0.1159	0.0264	1.5567	0.1001	0.0953	1.6510
dblp	0.0208	0.2860	0.0220	0.6443	0.1458	0.0693	0.7573	0.1523	0.5703	1.8762
elec	0.0002	0.0016	0.0002	0.0016	0.0006	0.0026	0.0302	0.0006	0.0162	0.0127
enron	0.0014	0.0099	0.0015	0.0095	0.0076	0.0154	0.1726	0.0069	0.0838	0.0378
epin	0.0011	0.0214	0.0012	0.0172	0.0039	0.0086	0.0202	0.0048	0.0415	0.1278
fb	0.0021	0.0113	0.0020	0.0087	0.0094	0.0180	0.3845	0.0093	0.1135	0.1716
flickr	0.0591	1.1019	0.0678	1.0379	0.4675	0.5288	3.8578	0.5014	2.4240	11.2012
digg	0.0001	0.0017	0.0001	0.0019	0.0003	0.0001	0.0059	0.0003	0.0070	0.0076
slash	0.0002	0.0044	0.0003	0.0049	0.0010	0.0032	0.0165	0.0011	0.0213	0.0262
conflict	0.0042	0.0504	0.0047	0.0482	0.0175	0.1371	0.0908	0.0227	0.3346	0.4253
growth	0.1432	1.6286	0.1676	1.9096	1.3442	3.8759	6.2699	1.5027	11.0289	26.3161
youtube	0.0326	0.2207	0.0355	0.1752	0.1175	0.0560	0.8352	0.1193	0.6535	1.5675

Table 3: Running time in seconds (high-degree sources)

	Earliest-arrival		Latest-departure		Fastest			Shortest		
	1-pass	Xuan	1-pass	Xuan	1-pass	Trans	Xuan	1-pass	Trans	Xuan
arxiv	0.0119	0.1829	0.0121	0.1726	0.3969	0.0565	45.5396	0.1673	0.1433	2.5805
dblp	0.0449	0.6254	0.0407	0.8303	0.7842	0.4654	9.5193	0.6706	1.6710	4.6050
elec	0.0003	0.0022	0.0003	0.0020	0.0017	0.0048	1.3386	0.0013	0.0313	0.0178
enron	0.0023	0.0217	0.0023	0.0217	0.1093	0.1053	22.8077	0.0329	0.3469	0.1563
epin	0.0014	0.0339	0.0019	0.0277	0.0155	0.0302	4.7978	0.0102	0.0928	0.2450
fb	0.0038	0.0416	0.0041	0.0350	0.0531	0.1558	16.6954	0.0400	0.4816	0.3764
flickr	0.0907	1.6400	0.1356	1.8558	2.6368	1.7497	81.5611	1.2522	5.1928	17.7573
digg	0.0002	0.0038	0.0002	0.0040	0.0009	0.0018	0.3940	0.0008	0.0177	0.0227
slash	0.0006	0.0082	0.0006	0.0088	0.0041	0.0125	1.4768	0.0032	0.0487	0.0503
conflict	0.0055	0.0578	0.0057	0.0624	0.1013	0.3264	32.6886	0.0520	0.7887	0.8003
growth	0.1794	2.0868	0.1936	2.3546	4.5404	7.7180	262.7313	2.8376	20.6097	29.6612
youtube	0.0559	1.0114	0.0610	1.1920	0.8317	0.8199	103.9525	0.6120	3.1188	5.1966

Table 4: Size of the transformed graph $\tilde{G}$

	arxiv	dblp	elec	enron	epin	fb
$\lvert\tilde{V}\rvert$	433K	5553K	212K	1367K	482K	1637K
$\lvert\tilde{E}\rvert$	9759K	16977K	313K	2505K	1219K	3037K

	flickr	digg	slash	conflict	growth	youtube
$\lvert\tilde{V}\rvert$	12600K	172K	273K	3191K	34815K	11498K
$\lvert\tilde{E}\rvert$	44358K	233K	381K	6009K	77196K	21140K

the reply network of technology website Slashdot; wikiconflict (conflict) indicating positive and negative conflicts between users of Wikipedia; wikipedia-growth (growth) from the hyperlink network of the English Wikipedia; youtube-u-growth (youtube) from the social media networks of YouTube.

Table 1 gives some statistics of the datasets. Apart from the number of vertices and edges in G and G_s, we also show the average degree in G (denoted by $d_{avg}(u, G)$) and in G_s (denoted by $d_{avg}(u, G_s)$). The table shows that the value of π varies significantly for different datasets, indicating the different levels of temporal activity between two vertices. Note that π=1 does not imply that the temporal graph is similar to the corresponding static graph, because edges on a temporal path follow an ordered time sequence. This is also revealed by the number of distinct time instances in G, denoted by $|T_G|$, which shows that G can span over a large time interval. For example, if we break G into snapshots such that all edges with the same starting time belong to the same snapshot, then the conflict graph consists of 273909 snapshots.

7.1 Efficiency of SSMTP Algorithms

To evaluate the performance of our algorithms for computing single-source minimum temporal paths (**SSMTPs**), we compare with the algorithms proposed by Xuan et al. [21], denoted by **Xuan**. Note that Xuan can only report the number of hops for shortest paths. Xuan et al. also did not study latest-departure paths and we modified their earliest-arrival path algorithm to compute latest-departure time. We denote our one-pass algorithms presented in Section 4 by **1-pass** and our graph transformation algorithms presented in Section 5 by **Trans**. All algorithms were implemented in C++ and compiled in the same way.

We use two sets of source vertices: 100 randomly selected vertices and 10 highest temporal degree vertices (note that the temporal degree, i.e., $d(u, G)$, decreases quickly beyond the top 10 highest ones). We set $[t_\alpha, t_\omega]$ to be $[0, \infty]$ in this experiment.

Tables 2 and 3 report the average running time of the algorithms. For Trans, there is no big difference between the running time of earliest-arrival and latest-departure paths and that of fastest paths, and hence we only present the results for fastest paths due to space limit. We also show the size of the transformed graph $\tilde{G}$ in Table 4.

The results show that 1-pass is significantly faster than Xuan in computing all the four types of SSMTPs for all the 12 datasets. On average, 1-pass is 13 to 18 times faster than Xuan for processing queries with randomly selected sources, and 13 to 251 times faster than Xuan for processing queries with high-degree sources. The reason for this huge difference in running time is because 1-pass is a one-pass algorithm with much lower complexity than Xuan, which is a rather straightforward adoption of Dijkstra's strategy or simply by enumeration of paths which is inefficient.

Compared with Trans, there are a number of cases 1-pass is less efficient for computing fastest paths. This is mainly because for computing fastest paths, the complexity of Trans is better than that of 1-pass, but the tradeoff is that the transformed graph is larger than the temporal graph. For computing shortest paths, the time complexity of 1-pass and Trans are similar. However, 1-pass is faster than Trans in all cases except for the arxiv dataset, which can be explained by the fact that $|\tilde{E}|$ is comparable with $|E|$ for arxiv but considerably larger than $|E|$ for other datasets. When the sizes of the input graphs are comparable, Trans and 1-pass have comparable performance for computing shortest paths.

7.2 Effect of Varying Time Intervals

For computing the minimum temporal paths, the input time interval $[t_\alpha, t_\omega]$ can affect the overall running time significantly. In this experiment, we test the effect of different $[t_\alpha, t_\omega]$ on the performance of our algorithms. We test five different time intervals, I_1 to I_5. We set $I_1 = [0, |T_G|]$, where $|T_G|$ is reported in Table 1. For each I_i, for $1 \le i \le 4$, we divide I_i into two equal sub-intervals so that I_{i+1} is the first sub-interval of I_i.

We report the average running time (in seconds) for computing fastest and shortest paths using 1-pass for the 100 randomly selected source vertices in Tables 5 and 6. The running time for computing earliest-arrival and latest-departure paths is smaller than that for fastest paths, but follows a similar trend with the varying time intervals. The running time for Trans and for the high-degree source vertices also follows the same trend. Thus, we omit the details of these results due to space limit.

Tables 5 and 6 clearly show that for all datasets, when the time interval becomes smaller, the running time is significantly reduced. Note that although the time interval is halved each time, the running time is reduced in a much faster rate in most cases. This can be explained as the reduction in the number of temporal edges can be more than halved, which causes a reduction in the values of both M and π. Another important reason is that as the number of temporal edges decreases, the number of reachable vertices that satisfy the time constraint also decreases, which also causes a reduction in the search space. Only in a few cases, the reduction of running time is

Table 5: Running time for varying intervals (fastest path)

	I_1	I_2	I_3	I_4	I_5
arxiv	0.1158	0.0037	0.0002	0.0000	0.0000
dblp	0.1440	0.0004	0.0000	0.0000	0.0000
elec	0.0006	0.0002	0.0001	0.0000	0.0000
enron	0.0075	0.0031	0.0011	0.0004	0.0001
epin	0.0040	0.0030	0.0024	0.0018	0.0015
fb	0.0095	0.0042	0.0024	0.0017	0.0013
flickr	0.4711	0.2307	0.1405	0.0496	0.0422
digg	0.0003	0.0001	0.0001	0.0000	0.0000
slash	0.0010	0.0003	0.0001	0.0000	0.0000
conflict	0.0176	0.0060	0.0020	0.0005	0.0002
growth	1.3652	0.0176	0.0018	0.0001	0.0000
youtube	0.1172	0.0291	0.0194	0.0099	0.0087

Table 6: Running time for varying intervals (shortest path)

	I_1	I_2	I_3	I_4	I_5
arxiv	0.0978	0.0036	0.0002	0.0000	0.0000
dblp	0.1520	0.0004	0.0000	0.0000	0.0000
elec	0.0006	0.0002	0.0001	0.0000	0.0000
enron	0.0069	0.0031	0.0012	0.0004	0.0001
epin	0.0049	0.0036	0.0027	0.0019	0.0015
fb	0.0093	0.0043	0.0024	0.0016	0.0013
flickr	0.5015	0.2362	0.1431	0.0494	0.0421
digg	0.0003	0.0001	0.0001	0.0000	0.0000
slash	0.0011	0.0003	0.0001	0.0000	0.0000
conflict	0.0227	0.0072	0.0022	0.0006	0.0002
growth	1.4803	0.0177	0.0018	0.0001	0.0000
youtube	0.1204	0.0292	0.0195	0.0100	0.0088

less than 2. This is mainly because the reduction in the number of temporal edges in those cases is less than halved (i.e., more edges are in the other half of the time period).

This result is encouraging as in real applications, often users may be only interested in temporal paths within a specific time period, e.g., a recent time window or the peak season last year.

7.3 Temporal Analysis vs. Static Analysis

In this experiment, we study the two applications of minimum temporal paths discussed in Section 6, by comparing with the results from the static graphs. Our objective is to show that analytic results on temporal graphs can be dramatically different from those on static graphs, and hence will carry additional important or even the real accurate information about the temporal data.

Closeness centrality. We compute closeness centrality based on fastest and shortest paths in a temporal graph G, respectively, and then compute closeness centrality based on the classic shortest paths in the condensed static graph G_s of G. Then, we use *Pearson correlation coefficient* (*PCC*) to measure the degree of linear correlation between $X = \{closeness(v, G) : v \in V\}$ and $Y = \{closeness(v, G_s) : v \in V_s\}$, i.e., X and Y are the closeness values of the vertices in G and in G_s, respectively (note that $V = V_s$).

Table 7 reports the PCC values, in which PCC_f and PCC_s indicate that closeness is defined based on fastest and shortest paths in G, respectively. The results show that the closeness values computed from the static graphs have low correlation with those computed from the temporal graphs for most datasets. This is not surprising if fastest paths are used to compute the closeness values; however, the results show that even if shortest paths are used to compute the closeness values in temporal graphs, the closeness values are also significantly different from those computed from the static graphs. The youtube dataset is an exception for which we found that the closeness values of all vertices are close to zero because the vertices in youtube cannot reach the majority of vertices in the graph. Note that for unreachable vertices, $dist(v, u)$ or $dura(P_{v,u})$ in the denominator of Equation 1 is set to n or $|T_G|$, resulting in a small reciprocal that can be close to 0 if most vertices are not reachable from v.

Table 7: Correlation between temporal and static closeness

	arxiv	dblp	elec	enron	epin	fb
PCC_f	0.2143	0.0491	-0.6833	0.6739	0.3753	-0.6277
PCC_s	0.4097	0.1134	0.7152	0.6305	0.3748	0.6906

	flickr	digg	slash	conflict	growth	youtube
PCC_f	0.2333	-0.3791	0.4284	0.4147	0.2568	1
PCC_s	0.2318	0.4332	0.5681	0.5019	0.3958	1

Table 8: NDCG values (fastest path)

k	100	200	300	400	500
arxiv	0.9135	0.8451	0.7906	0.7415	0.7030
dblp	0.5348	0.4564	0.4145	0.3863	0.3662
elec	0.4979	0.4604	0.4621	0.4711	0.4835
enron	0.4522	0.4285	0.4157	0.4023	0.3962
epin	0.3667	0.3184	0.3091	0.3163	0.3241
fb	0.6778	0.5518	0.4899	0.4563	0.4351
flickr	0.4107	0.3145	0.2662	0.2363	0.2165
digg	0.5026	0.4301	0.4029	0.3889	0.3820
slash	0.3088	0.2698	0.2567	0.2541	0.2567
conflict	0.4395	0.3818	0.3501	0.3312	0.3160
growth	0.4677	0.3446	0.2909	0.2638	0.2471
youtube	0.3955	0.2995	0.2557	0.2299	0.2113

Table 9: NDCG values (shortest path)

k	100	200	300	400	500
arxiv	0.9808	0.9596	0.9394	0.9204	0.9047
dblp	0.7484	0.6734	0.6255	0.5919	0.5679
elec	0.7136	0.6495	0.6202	0.6038	0.5968
enron	0.8048	0.7716	0.7466	0.7281	0.7115
epin	0.5440	0.5087	0.5044	0.5049	0.5078
fb	0.8589	0.7920	0.7536	0.7298	0.7142
flickr	0.4958	0.4085	0.3712	0.3521	0.3369
digg	0.6968	0.6103	0.5633	0.5346	0.5149
slash	0.5736	0.5162	0.4912	0.4727	0.4603
conflict	0.6708	0.6338	0.6076	0.5936	0.5869
growth	0.6509	0.5772	0.5438	0.5240	0.5100
youtube	0.6328	0.5584	0.5215	0.4967	0.4778

Top-k nearest neighbors. We next compute the top-k nearest neighbors using the four minimum temporal paths, and compare with the top-k nearest neighbors using shortest paths in the condensed static graph G_s. To assess the effectiveness of shortest paths in G_s in capturing the ranking defined by minimum temporal paths in G, we compute the *normalized discounted cumulative gain* (*NDCG*) of the top-k ranking in G_s. The relevance of each top-k vertex in G_s is given as the corresponding ranking of the vertex in G computed based on each of the four minimum temporal paths. The NGCG value varies from 0 to 1, with 1 representing the same ranking as in the temporal graph. We compute the top-k nearest neighbors for the 100 randomly selected source vertices and report the average NDCG values in Tables 8-9 (the results for earliest-arrival and latest-departure paths can be found in [19] which lead to a similar conclusion as follows).

The results show that the ranking obtained from the static graph can be significantly different from that obtained from the temporal graph. The difference becomes particularly obvious when k increases. Thus, the temporal information is critical in determining the top-k nearest neighbors in a temporal graph.

The results of both the closeness measure and the top-k ranking may not suggest that the results from the static graphs are totally meaningless. However, the results clearly reveal that analyzing temporal graphs may obtain results that are very different from that obtained from static graphs. This calls the need for studying temporal graphs directly, which is particularly necessary for analyzing temporal properties of the graph.

8. RELATED WORK

The closest related work is [21], and we have explained in Section 3 that our path definitions are more general than theirs. Compared with our one-pass algorithms, their algorithm for computing earliest-arrival path is rather straightforward adoption of Dijkstra's strategy, while their algorithms for computing fastest and shortest paths are essentially by enumeration of paths which is inefficient. Thus, even though we solve more general problems, our algorithms attain much lower time complexity than theirs (see Theorem 1, Propositions 2 and 3 in [21]). Our experimental results also verify that our algorithms are one to two orders of magnitude faster than theirs on average. Minimum temporal paths were also studied in our previous work [9]. The focus of that work is on temporal graph traversals, but we also applied temporal DFS/BFS to compute minimum temporal paths with linear time complexity.

Many applications of temporal paths were proposed, which we briefly discuss as follows. Temporal paths were applied to study the connectivity of a temporal network [10], for which disjoint temporal paths between any two vertices are computed. In [11], a similar definition of latest-departure path (without the information of λ for the edges) was proposed to study information latency. In [12], four definitions of temporal proximity were introduced, which are no more than finding earliest-arrival, latest-departure, and fastest paths, but they also did not consider the information of λ, which is useful in many applications such as flight scheduling and logistic/itinerary planning. They also did not propose any algorithm for path computation. In [16, 17], the earliest-arrival time was applied to define metrics such as temporal efficiency (i.e., how easy information flows from one vertex to another) and temporal clustering coefficient. Temporal paths were also applied to find temporal connected components in [13, 17]. In [18], small-world behavior was analyzed in temporal networks using temporal paths. In [15], betweenness and closeness based on the three types of temporal paths in [21] were briefly mentioned but not studied. In [14], empirical studies were conducted to measure correlation between temporal paths and closeness defined based on earliest-arrival time averaged over all starting time instances. Their results provide some insights about real temporal graphs, but the datasets they used are much smaller than those we used. In [20], a temporal graph is used to model users' long-term and short-term preferences over time and the temporal information is used for recommendation. Apart from that, there are surveys [1, 8] that cover most of the prior proposed concepts of temporal graphs.

9. CONCLUSIONS

We presented four types of minimum temporal paths. Among them, only shortest path is a well-known concept in normal static graphs, but we have shown that the concept of shortest path in temporal graphs is very different from that in static graphs. The other three types, i.e., earliest-arrival paths, latest-departure paths and fastest paths, are unique in temporal graphs, and all carry new, different and important temporal information about the graph. We first proposed efficient one-pass algorithms that use only one linear scan of the input graph for computing the minimum temporal paths, which is scalable for massive temporal graphs. We next proposed an alternative solution that transforms a temporal graph into a non-temporal one with no information loss. Experiments on a wide range of real-world temporal graphs show that our algorithms are one order to two orders of magnitude faster than the existing algorithms [21]. We also demonstrated, through the applications of closeness centrality computation and top-k nearest neighbors, that minimum temporal paths lead to analytic results that are significantly different from shortest paths in static graphs. This shows the need for studying temporal graphs directly instead of condensing them into static graphs, and thus we believe that many applications can be developed from minimum temporal paths.

For future work, we plan to develop indexes for answering queries on temporal paths, by applying indexing techniques for non-temporal graphs [2, 3, 4, 5, 7].

Acknowledgments. We thank the reviewers for their many useful comments that have helped improve the paper significantly. This research is supported in part by the CUHK Direct Grant No. 4055017.

10. REFERENCES

[1] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.
[2] J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *SIGMOD Conference*, pages 193–204, 2013.
[3] J. Cheng, Y. Ke, S. Chu, and C. Cheng. Efficient processing of distance queries in large graphs: a vertex cover approach. In *SIGMOD Conference*, pages 457–468, 2012.
[4] J. Cheng, Z. Shang, H. Cheng, H. Wang, and J. X. Yu. K-reach: Who is in your small world. *PVLDB*, 5(11):1292–1303, 2012.
[5] J. Cheng, Z. Shang, H. Cheng, H. Wang, and J. X. Yu. Efficient processing of k-hop reachability queries. *VLDB J.*, 23(2):227–252, 2014.
[6] D. Eppstein and J. Wang. Fast approximation of centrality. In *SODA*, pages 228–229, 2001.
[7] A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong. Is-label: an independent-set based labeling scheme for point-to-point distance querying. *PVLDB*, 6(6):457–468, 2013.
[8] P. Holme and J. Saramäki. Temporal networks. *CoRR*, abs/1108.1780, 2011.
[9] S. Huang, J. Cheng, and H. Wu. Temporal graph traversals: Definitions, algorithms, and applications. *CoRR*, abs/1401.1919, 2014.
[10] D. Kempe, J. M. Kleinberg, and A. Kumar. Connectivity and inference problems for temporal networks. *J. Comput. Syst. Sci.*, 64(4):820–842, 2002.
[11] G. Kossinets, J. M. Kleinberg, and D. J. Watts. The structure of information pathways in a social communication network. In *KDD*, pages 435–443, 2008.
[12] V. Kostakos. Temporal graphs. *Physica A: Statistical Mechanics and its Applications*, 388(6):1007–1023, 2009.
[13] V. Nicosia, J. Tang, M. Musolesi, G. Russo, C. Mascolo, and V. Latora. Components in time-varying graphs. *CoRR*, abs/1106.2134, 2011.
[14] R. K. Pan and J. Saramäki. Path lengths, correlations, and centrality in temporal networks. *Phys. Rev. E*, 84:016105, 2011.
[15] N. Santoro, W. Quattrociocchi, P. Flocchini, A. Casteigts, and F. Amblard. Time-varying graphs and social network analysis: Temporal indicators and metrics. *CoRR*, abs/1102.0629, 2011.
[16] J. Tang, M. Musolesi, C. Mascolo, and V. Latora. Temporal distance metrics for social network analysis. In *Proceedings of the ACM Workshop on Online Social Networks*, pages 31–36, 2009.
[17] J. Tang, M. Musolesi, C. Mascolo, and V. Latora. Characterising temporal distance and reachability in mobile and online social networks. *Computer Communication Review*, 40(1):118–124, 2010.
[18] J. Tang, S. Scellato, M. Musolesi, C. Mascolo, and V. Latora. Small-world behavior in time-varying graphs. *Physical Review E*, 81(5):055101, 2010.
[19] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs (preliminary version). `www.cse.cuhk.edu.hk/~hhwu/temsp.pdf`, 2013.
[20] L. Xiang, Q. Yuan, S. Zhao, L. Chen, X. Zhang, Q. Yang, and J. Sun. Temporal recommendation on graphs via long- and short-term preference fusion. In *KDD*, pages 723–732, 2010.
[21] B.-M. B. Xuan, A. Ferreira, and A. Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *Int. J. Found. Comput. Sci.*, 14(2):267–285, 2003.