Efficient Algorithms for Temporal Path Computation

Huanhuan Wu, James Cheng, Yiping Ke, Silu Huang, Yuzhen Huang, and Hejun Wu

Abstract—Shortest path is a fundamental graph problem with numerous applications. However, the concept of classic shortest path is insufficient. In this paper, we study various concepts of "shortest" path in temporal graphs, called minimum temporal paths. Computing these minimum temporal paths is challenging as subpaths of a "shortest" path may not be "shortest" in a temporal graph. We propose efficient algorithms to compute minimum temporal paths and verified their efficiency using large real-world temporal graphs.

Index Terms—Temporal graphs, temporal path, parallel temporal path algorithms

1 Introduction

Research on graph data has been extensively conducted in recent years thanks to the increasing popularity of many online social networks and mobile communication networks. Existing research has mainly focused on the study of non-temporal graphs, while some have also considered dynamic graphs as a sequence of updates to non-temporal graphs. However, many real-world graphs are actually *temporal graphs*, in which edges have temporal labels. For example, assume that Fig. 1a shows a flight network, then the two 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 in 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, information spreads from A to B at time t in information dissemination networks, to name but a few. Readers may also find various temporal networks in cell biology, neural and brain connections, ecological systems, infra-structural networks, physical proximity, and among others, in a survey of temporal networks [1].

Temporal graphs are commonly condensed into non-temporal graphs, by removing all time information from the temporal graph and collapsing multiple edges between any two vertices into a single edge, because their non-temporal version is much easier to handle [2]. Condensing a temporal graph into a non-temporal 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 non-temporal graph often presents erroneous information that leads to serious incorrect understanding of the graph or relationship between objects. We illustrate the problems by the following example.

Example 1. Fig. 1a shows a temporal graph G. Assume that G is a flight 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 one day. Fig. 1b shows the condensed non-temporal graph G_s of G. We can see some paths in G_s may not be a meaningful path in G. For example, $\langle a, b, g, j\rangle$ is a path in G_s, but $\langle a, b, g, j\rangle$ in G 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 one day to fly from b to g). Now consider a shortest path from a to l in the two graphs. In G_s, the shortest path is $\langle a, i, l\rangle$ with distance 2. But in G, 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 non-temporal 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 algorithms for studying temporal graphs are 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 [3], [4], graph clustering [5], etc.).

Due to the presence of temporal information, different forms of "shortest" paths exist and each has its own meaning and significance. We study four types of temporal paths

- *H. Wu, J. Cheng, and S. Huang are with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Shatin, NT, Hong Kong. E-mail: {hhwu, jcheng, slhuang}@cse.cuhk.edu.hk.*
- *Y. Ke is with the School of Computer Engineering, Nanyang Technological University, Singapore 639798. E-mail: ypke@acm.org.*
- *Y. Huang and H. Wu are with the Guangdong Province Key Laboratory of Big Data Analysis and Processing, Sun Yat-sen University, Guangzhou 510275, China. E-mail: hyuzhen2@mail2.sysu.edu.cn, wuhejun@mail.sysu.edu.cn.*

Manuscript received 4 Aug. 2015; revised 11 July 2016; accepted 13 July 2016. Date of publication 26 July 2016; date of current version 3 Oct. 2016. Recommended for acceptance by F. Afrati. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TKDE.2016.2594065

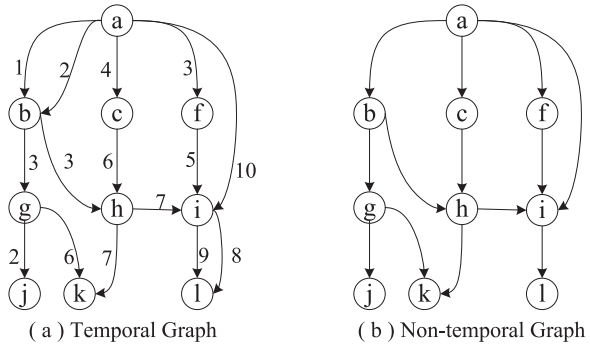


Fig. 1. Temporal graph G and its non-temporal version G_s.

(also commonly called "journeys" in a temporal graph) [6], [7], collectively we call them *minimum temporal paths*, as they give the minimum value for different measures: (1) *foremost path* (i.e., a path that gives the earliest time one can reach a target y from a source x); (2) *reverse-foremost path* (i.e., a path that gives the latest time one must leave x in order to reach y before a given time); (3) *fastest path* (i.e., a path by which one goes from x to y using the minimum elapsed time); and (4) *shortest path* (i.e., a path that is shortest from x to y in terms of the overall traversal time needed on the edges).

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 non-temporal 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 investigate useful properties of temporal paths to devise efficient sequential algorithms to compute the minimum temporal paths, including one-pass algorithms and a graph transformation approach. Since the size of temporal graphs can be large, we also propose scalable parallel algorithms to compute the minimum temporal paths. We evaluate the performance of our algorithms using a wide spectrum of real-world temporal graphs, showing that they significantly outperform existing algorithms. Our results also verify the scalability of our parallel algorithms.

Note that in this paper, we assume that the entire temporal graph is given, i.e., it is available to the algorithms so that we can design efficient off-line algorithms. When the full knowledge of the input temporal graph is not available, Casteigts et al. [8] proposed an algorithm for the problem of measuring temporal lags (similar to reverse-foremost path) in a distributed setting, and they also showed that the algorithm can be applied to compute foremost and fastest broadcast trees in periodically-varying graphs. In addition, Casteigts et al. [9] further studied the distributed version of three problems (shortest, fastest, and foremost broadcast) in three different classes of dynamic graphs. By comparing the feasibility of these problems within different classes, they obtained a knowledge requirement hierarchy among the three problems, and a computational relationship hierarchy among the different classes of dynamic graphs. Since the focuses of [8], [9] and our work are different, it is difficult to apply the algorithms of [8], [9] for efficient off-line analytics, while it is also non-trivial to extend our algorithms for the online setting (which we leave to future work).

TABLE 1
Frequently-Used Notations

Notation	Description
$G = (V, E)$	A temporal graph
$e = (u, v, t, \lambda) \in E$	A temporal edge
$t(e)$	The starting time of edge e
$\lambda(e)$	The traversal time of edge e
$\Pi(u, v)$	The set of temporal edges from u to v
$\pi(u, v)$	The number of temporal edges from u to v
π	Max. # of temporal edges between any two vertices in G
$G_s = (V_s, E_s)$	The corresponding non-temporal graph of G
n	The number of vertices in G (G_s)
m	The number of edges in G_s
M	The number of edges in G
$\Gamma_{out}(u, G)$ ($\Gamma_{in}(u, G)$)	The set of out-neighbors (in-neighbors) of a vertex u in G
$d_{out}(u, G)(d_{in}(u, G))$	The out-degree (in-degree) of a vertex u in G
P	A temporal path $P = \langle v_1, v_2, \ldots, v_k, v_{k+1} \rangle$
$e_i = (v_i, v_{i+1}, t_i, \lambda_i)$	The ith temporal edge on P
$end(P)$	The ending time of P, $end(P) = t_k + \lambda_k$
$start(P)$	The starting time of P, $start(P) = t_1$
$dura(P)$	The duration of P, $dura(P) = end(P) - start(P)$
$dist(P)$	The distance of P, $dist(P) = \sum_{i=1}^{k} \lambda_i$
$nextST((u, v), t)$	The earliest time u can traverse to v starting at time t
$prevST((u, v), t)$	The latest time v can be traversed from u arriving by time t

Paper Outline. Section 2 gives the notations of temporal graphs. Section 3 presents the minimum temporal paths. Sections 4 and 5 discuss the one-pass algorithms and the graph transformation approach. Section 6 presents the parallel algorithms. Section 7 reports experimental results. Section 8 discusses related work and Section 9 concludes the work.

2 Notations of Temporal Graphs

The frequently used notations are listed in Table 1. 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 there exists and edge 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 gave 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 call duration or the message transmission 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 at time t and lasts for a duration of 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, $\pi = 1,643,088$).

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 non-temporal edges, we obtain the corresponding *non-temporal graph* $G_s = (V_s, E_s)$ of G, where $V_s = V$ and $E_s = \{(u, v) : (u, v, t, \lambda) \in E\}$, i.e., all temporal information is removed 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 $d_{in}(u, G_s) = |\Gamma_{in}(u, G_s)|$.

Fig. 1a shows a temporal graph G and its corresponding non-temporal graph G_s is shown in Fig. 1b. 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 [6] (also commonly called a "journey") in a temporal graph G is a sequence of edges $P = \langle e_1, e_2, \ldots, e_k \rangle$, where $e_i = (v_i, v_{i+1}, t_i, \lambda_i) \in E$ is the ith 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 $dist(P) = \sum_{i=1}^{k} \lambda_i$. We illustrate the concepts as follows.

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 Fig. 1a. 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 this paper, we study the following four types of minimum temporal paths [7].

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*:*

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

Reverse-Foremost Path. $P \in \mathbf{P}(x, y, [t_\alpha, t_\omega])$ *is a reverse-foremost 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* $dist(P) = \min\{dist(P') : P' \in \mathbf{P}(x, y, [t_\alpha, t_\omega])\}$.

When a time interval $[t_\alpha, t_\omega]$ is not explicitly specified for the minimum temporal paths, we simply take it 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. Note that some recent papers (e.g., [10], [11]) use the term "earliest-arrival path" for "foremost path", and "latest-departure path" for "reverse-foremost path". In this paper we use the terms defined in earlier work [7].

The concept of temporal path was introduced in [6]. Three out of the four types of paths we study here (i.e., foremost, fastest, and shortest paths) were studied in [7]. Compared with [7], our temporal path problems are more general: first, in [7] 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 foremost path from x to every $v \in V$, or (2) the reverse-foremost 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) *foremost*

time $end(P)$, or (2) *reverse-foremost 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.

Properties of Minimum Temporal Paths. The following lemmas give some properties of minimum temporal paths.

Lemma 1. *A prefix-subpath of an foremost path may not be an foremost path.*

Consider the temporal graph G in Fig. 1a, $P = \langle(a, b, 2, 1), (b, g, 3, 1), (g, k, 6, 1)\rangle$ is an foremost path from a to k with $end(P) = 6 + 1 = 7$; but its prefix subpath $P_1 = \langle(a, b, 2, 1)\rangle$ is not an foremost path from a to b, since $end(P_1) = 2 + 1 = 3$ while $P_2 = \langle(a, b, 1, 1)\rangle$ has $end(P_2) = 1 + 1 = 2$.

Lemma 2. *A postfix-subpath of a reverse-foremost path may not be a reverse-foremost path.*

For example, $P = \langle(a, c, 4, 1), (c, h, 6, 1), (h, i, 7, 1), (i, l, 8, 1)\rangle$ in Fig. 1a is a reverse-foremost path from a to l with $start(P) = 4$; but its postfix subpath $P_1 = \langle(i, l, 8, 1)\rangle$ is not a reverse-foremost path from i to l, as $P_2 = \langle(i, l, 9, 1)\rangle$ is a path from i to l with $start(P_2) = 9 > start(P_1) = 8$.

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

For example, $P = \langle(a, c, 4, 1), (c, h, 6, 1), (h, k, 7, 1)\rangle$ in Fig. 1a is a fastest path from a to k with $dura(P) = (7 + 1) - 4 = 4$; but its prefix subpath $P_1 = \langle(a, c, 4, 1), (c, h, 6, 1)\rangle$ is not a fastest path from a to h, instead $P_2 = \langle(a, b, 2, 1), (b, h, 3, 1)\rangle$ is a fastest path from a to h with $dura(P_2) = (3 + 1) - 2 = 2$.

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

For example, $P = \langle(a, f, 3, 1), (f, i, 5, 1), (i, l, 8, 1)\rangle$ in Fig. 1a is a shortest path from a to l with $dist(P) = 1 + 1 + 1 = 3$; but its prefix subpath $P_1 = \langle(a, f, 3, 1), (f, i, 5, 1)\rangle$ is not a shortest path from a to i, instead $P_2 = \langle(a, i, 10, 1)\rangle$ is a shortest path from a to i with $dist(P_2) = 1$.

Lemmas 1, 2, 3, and 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. Due to the space limit, the proofs of some lemmas and theorems are given in the appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2016.2594065, while other less obvious proofs are kept in the paper to help understanding. Since the one-pass algorithm for computing reverse-foremost time is similar to that of computing foremost time, and the algorithm for computing shortest-path distance is similar to that of computing fastest-path duration, we also give the one-pass algorithms for computing reverse-foremost time and shortest-path distance in the appendix, available in the online supplemental material.

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. In this paper, we assume that the temporal graph is in edge stream representation. If not, the pre-processing takes $O(M \log M)$ time to sort the edges into the edge stream representation.

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 $e_1, e_2, \ldots, e_k$ are in strictly ascending order, and hence e_i comes before e_j in the edge stream of G. □

4.2 Foremost Paths

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

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 foremost path may not be an foremost 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 foremost paths, though the following observation shows otherwise.

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

Proof. Given any foremost path $P \in \mathbf{P}$, if not every prefix-subpath in it is an foremost 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 foremost path from x to v_i. Thus, there exists another path $\hat{P}_i$ that is an foremost 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 foremost 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 foremost path and the resulting $\hat{P}$ is in **P**, which proves the lemma. □

Based on Lemma 6, we can apply the greedy strategy to grow the foremost 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 [12], which is inefficient for processing large temporal graphs in practice.

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 foremost paths with only one scan of the graph. We present our one-pass algorithm in Algorithm 1 and elaborate as follows.

Algorithm 1. Computing Foremost Time

Input: A temporal graph $G = (V, E)$ in its edge stream representation, source vertex x, time interval $[t_\alpha, t_\omega]$
Output: The foremost 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;
```

We use an array $t[v]$ to keep the current foremost 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 Fig. 1a, where we assume that the traversal time λ is 1 for all edges. Let a be the source vertex. We compute the foremost 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 foremost 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 foremost time from x to v.

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

The following theorem states our main result for foremost path computation.

Theorem 1. *Algorithm 1 correctly computes the foremost 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.*

4.3 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 8. *Let **P** be the set of temporal paths from x to v with the same starting time t. Then, $P \in$ **P** is a fastest path from x to v starting at t if P is an foremost path from x to v starting at t.*

Lemma 8 implies that we can compute the fastest path from x by finding the foremost 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 2.

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

We give the correctness and complexity of Algorithm 2 below.

Theorem 2. *Algorithm 2 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, $t(e) \geq t_\alpha, t(e) + \lambda(e) \leq t_\omega\}$.*

A One-Pass Algorithm with Better Time Bound.

In Algorithm 2, 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 3.

Algorithm 2. 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 $f[v] = \infty$ for all $v \in V \setminus \{x\}$;
2 Let S be the set of distinct starting time of the out-edges of x within $[t_\alpha, t_\omega]$, i.e., $S = \{t(e) : e \text{ is an out-edge of } x, t(e) \geq t_\alpha, t(e) + \lambda(e) \leq t_\omega\}$;
3 **foreach** $t \in S$ **do**
4 Call Algorithm 1 with input G, x, and time interval $[t, t_\omega]$; let $t[v]$ be the foremost time from x to v returned by Algorithm 1, then update $f[v] \leftarrow \min\{f[v], t[v] - t\}$;
5 **return** $f[v]$ for each $v \in V$;

Algorithm 3. 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** $v \in V$ **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 $f[v] = \infty$ for all $v \in V \setminus \{x\}$;
4 **foreach** *incoming edge* $e = (u, v, t, \lambda)$ *in the edge stream* **do**
5 **if** $t \geq t_\alpha$ *and* $t + \lambda \leq t_\omega$ **then**
6 **if** $u = x$ **then**
7 **if** $(t, t) \notin L_x$ **then**
8 Insert (t, t) into L_x;
9 Let $(s'[u], a'[u])$ be the element in L_u where $a'[u] = \max\{a[u] : (s[u], a[u]) \in L_u, a[u] \leq t\}$;
10 $s[v] \leftarrow s'[u]$;
11 $a[v] \leftarrow t + \lambda$;
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** $t \geq t_\omega$ **then**
20 Break the for-loop and go to Line 21;
21 **return** $f[v]$ for each $v \in V$;

The algorithm uses a sorted list for each vertex v, denoted by L_v, to keep the foremost 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 9. *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 $a'[v] \leq a[v]$, 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 3, every time after removing dominated elements in L_v, we have the following property regarding L_v.

Lemma 10. *Each time after Line 16 of Algorithm 3 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 $a'[v] \leq a[v]$, 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 3. We scan the edge stream of the input graph once. For each incoming edge $e = (u, v, t, \lambda)$, we check whether the foremost 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 10 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 9 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 3. *Let $S = \{t(e) : e \text{ is an out-edge of } x,\ t(e) \geq t_\alpha, t(e) + \lambda(e) \leq t_\omega\}$, $d_{max} = \max\{d_{in}(v, G) : v \in V\}$, and $c = \min\{|S|, d_{max}\}$. Algorithm 3 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 foremost path from x to v within the time interval $[t_x, t_y]$. By Lemma 6, there exists an foremost path P from x to v such that every prefix-subpath of P is an foremost 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 foremost time from x to v_i within $[t_x, t_y]$, for $1 \le i \le k+1$. Let e_1, e_2, $\ldots$, 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 10 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. □

5 A Graph Transformation Approach

In this section, 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 $t_1 > t_2$.
   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 $t_{out} = \min\{t : (v, t) \in \tilde{V}_{out}(v), t \ge t_{in}\}$ 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 $1 \le i < k$. No edge is created if $k \le 1$. 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 Foremost Paths

We first discuss the computation of single-source foremost paths. To compute foremost 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 foremost time from x to v in G.

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

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

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

5.2 Reverse-Foremost Paths

Similar to the computation of single-source foremost 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

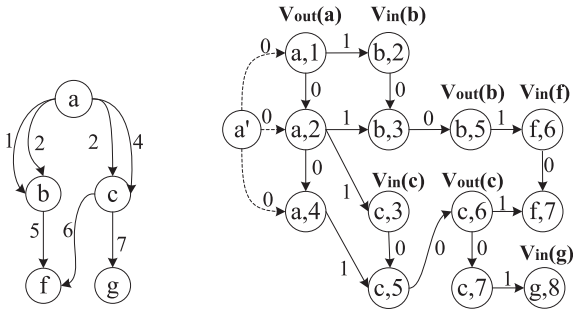


Fig. 2. Graph transformation, from G in (a) to $\tilde{G}$ in (b).

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 reverse-foremost time from every v to x in G.

Note that $\tilde{V}_{in}(a)$ does not exist in Fig. 2b since there is no reverse-foremost path from any vertex to a in Fig. 2a. But we can easily compute the reverse-foremost 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 foremost time $t[v]$ from x to every v and obtain the duration of this foremost 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 foremost 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 foremost paths and reverse-foremost 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 Parallel Algorithms

To process massive graphs that are too large to be handled by algorithms running on a single computer, parallel algorithms offer a good recourse. In this section, we present vertex-centric parallel algorithms based on Pregel's computing model [13].

In Pregel model, each vertex is an independent computational unit. The system (e.g., Pregel) distributes (by hashing) vertices to workers in a cluster, where each vertex v is associated with its adjacency list (i.e., the set of v's neighbors). A vertex v may also assign extra fields depending on the algorithm. A program in Pregel implements a user-defined *compute*() function and proceeds in iterations, called *supersteps*, based on the bulk synchronous parallel (BSP) model. In each superstep, the program runs *compute*() for each active vertex, v, in which v receives incoming messages from its neighbors sent in the previous superstep, modifies its value, sends messages to its neighbors (to be received in the next superstep), and finally votes to halt. The program terminates when all vertices vote to halt and there is no pending message for the next superstep.

Before we present our algorithms, we first describe the adjacency list representation [14] data format for storing a temporal graph in the vertex-centric setting. First, each vertex $u \in V$ is assigned a unique ID, and keeps the set of u's out-neighbors, $\Gamma_{out}(u,G)$, where the out-neighbors are sorted by their IDs. As for each neighbor $v \in \Gamma_{out}(u,G)$, the edge (u,v) may have different "starting time"s. Thus, for each $v \in \Gamma_{out}(u,G)$, v is associated with a set of "starting time"s, sorted in ascending order of the time values. In addition, each starting time t is also associated with the traversal time from u to v starting at time t.

For simplicity of discussion, for each distinct pair (u,v), given any two temporal edges (u,v,t_1,λ_1) and (u,v,t_2,λ_2), we assume $\lambda_1 = \lambda_2$. Thus, in the following discussions, we simply use $\lambda(u,v)$ to denote the traversal time from u to v at any starting time. We note that our algorithms can be easily extended to handle the case $\lambda_1 \neq \lambda_2$.

In this section, we only present the parallel algorithms for computing foremost time and fastest-path duration, by adopting the idea of the algorithms in our preliminary work [12]. The parallel algorithms for computing reverse-foremost time and shortest-path distance are in the appendix, available in the online supplemental material, which are similar to that of computing foremost time and fastest-path duration.

6.1 Foremost Paths

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

We define $nextST((u,v),t) = \min\{t(e) + \lambda(e) : e$ is an edge from u to $v, t(e) \geq t\}$, and $nextST((u,v),t) = \infty$ if no e with $t(e) \geq t$ exists. Intuitively, $nextST((u,v),t)$ is the earliest time at v that u can traverse to v starting at time t, while the value ∞ indicates that there is no such edge from u to v with starting time at or after t.

Based on Lemma 6, we can design a parallel algorithm to grow the foremost paths as shown in Algorithm 4. Every vertex v keeps two fields: $t(v)$ and L_v, where $t(v)$ keeps the foremost time from the source vertex x to v within time interval $[t_\alpha, t_\omega]$, and L_v keeps a value $p[u]$ for every $u \in \Gamma_{out}(v, G)$, which is an upper bound of the foremost time from x to u within $[t_\alpha, t_\omega]$.

Algorithm 4. Parallel Computation of Foremost Time

Input: A temporal graph $G = (V, E)$, source vertex x, time interval $[t_\alpha, t_\omega]$
Output: The foremost time $t(v)$ from x to every vertex $v \in V$ within $[t_\alpha, t_\omega]$

1 Each vertex $v \in V$ keeps two fields: $t(v)$ and L_v, where L_v keeps a value $p[u]$ for every $u \in \Gamma_{out}(v, G)$;
2 Initially, all vertices are *active*, the master calls $v.compute()$ for each $v \in V$ to start the computation;
3 v.**compute**(***messages***):
4 **begin**
5 **if** *superstep* # $= 1$ **then**
6 **if** v *is* x **then**
7 Set $t(v) \leftarrow t_\alpha$, $p[u] \leftarrow \infty$ for every $u \in \Gamma_{out}(v, G)$;
8 **foreach** $u \in \Gamma_{out}(v, G)$ **do**
9 **if** $nextST((v, u), t(v)) \leq t_\omega$ **then**
10 $p[u] \leftarrow nextST((v, u), t(v))$;
11 Send a message $< p[u] >$ to u;
12 **else**
13 Set $t(v) \leftarrow \infty$, $p[u] \leftarrow \infty$ for every $u \in \Gamma_{out}(v, G)$;
14 **else**
15 Set local variable $flag \leftarrow false$;
16 **foreach** *message*, $< t >$, *received in* ***messages*** **do**
17 **if** $t(v) > t$ **then**
18 $t(v) \leftarrow t$;
19 $flag \leftarrow true$;
20 **if** *flag is true* **then**
21 **foreach** $u \in \Gamma_{out}(v, G)$ **do**
22 **if** $nextST((v, u), t(v)) < p[u]$ *and* $nextST((v, u), t(v)) \leq t_\omega$ **then**
23 $p[u] \leftarrow nextST((v, u), t(v))$;
24 Send a message $< p[u] >$ to u;
25 v votes to halt;
26 **end**

The algorithm processes in supersteps. In the first superstep (Line 13), each vertex $v \in V$, except the source vertex x, initializes $t(v)$ and $p[u]$, for every $u \in \Gamma_{out}(v, G)$, to ∞, indicating that v has not been reached from x and u has not been reached from x via v. On the other hand (Lines 7-11), the source vertex x sets its $t(x)$ value to t_α, and sets $p[u]$ to the potential foremost time, $nextST((x, u), t(x))$, for each $u \in \Gamma_{out}(x, G)$. Vertex x then sends a message, which is the new value $p[u]$, to each out-neighbor u.

In each of the subsequent supersteps, when a vertex v receives messages from its neighbors, v is activated and processes the following (Lines 15-25). Note that the messages received by v are a set of potential foremost time that x traverses to v via one of v's in-neighbors. If the minimum value, t, of the received message values is less than its current $t(v)$ value, v updates $t(v)$ to t. Then, for each out-neighbor $u \in \Gamma_{out}(v, G)$, if x can reach u via the edge (v, u) at a time earlier than $p[u]$ and t_ω, i.e., $nextST((v, u), t(v)) < p[u]$ and $nextST((v, u), t(v)) \leq t_\omega$, the algorithm updates $p[u]$ to the new potential foremost time, $nextST((v, u), t(v))$, and also sends $nextST((v, u), t(v))$ as a message to u. The above superstep repeats until all vertices vote to halt and there is no more message pending for the next superstep, at which point $t(v)$ stores the final foremost time from the source vertex x to v.

The following theorem shows that when Algorithm 4 terminates, $t(v)$ correctly keeps the foremost time from x to v.

Theorem 4. *Algorithm 4 correctly computes the foremost time from a source vertex x to every vertex $v \in V$ within the time interval $[t_\alpha, t_\omega]$.*

6.2 Fastest Paths

In this section, we present our parallel algorithm for computing the *duration of the fastest path* from a source vertex x to every vertex in G. Lemma 8 implies that we can compute the fastest path by calling Algorithm 4 to compute the foremost path starting at every time instance in the time range $[t_\alpha, t_\omega]$. However, there can be too many rounds of computation. Thus, we design a parallel algorithm, as given in Algorithm 5, to address this problem and to eliminate redundant processing.

Algorithm 5. Parallel Computation of Fastest-Path Duration

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

1 Each vertex $v \in V$ keeps two fields: $f(v)$ and a sorted list 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;
2 Initially, all vertices are *active*, the master calls $v.compute()$ for each $v \in V$ to start the computation;
3 v.**compute**(***messages***):
4 **begin**
5 **if** *superstep* # $= 1$ **then**
6 **if** v is x **then**
7 Set $f(v) \leftarrow 0$, $L_v \leftarrow \emptyset$;
8 **foreach** *out-edge* (v, u, t, λ) *of* v *with* $t \geq t_\alpha$ *and* $t + \lambda \leq t_\omega$ **do**
9 Insert (t, t) into L_v;
10 Send a message $< (t, t + \lambda) >$ to u;
11 **else**
12 Set $f(v) \leftarrow \infty$, $L_v \leftarrow \emptyset$;
13 **else**
14 **foreach** *message*, $< (s, a) >$, *received in* ***messages*** **do**
15 **if** (s, a) *is not dominated by any pair in* L_v **then**
16 Insert (s, a) into L_v;
17 Remove *dominated elements* in L_v;
18 **if** $a - s < f(v)$ **then**
19 $f(v) \leftarrow a - s$;
20 **foreach** $u \in \Gamma_{out}(v, G)$ **do**
21 **if** $nextST((v, u), a) \leq t_\omega$ **then**
22 Send a message $< (s, nextST((v, u), a)) >$ to u;
23 v votes to halt;
24 **end**

In Algorithm 5, every vertex v keeps two fields: $f(v)$ and L_v, where $f(v)$ keeps the duration of the fastest path from the source vertex x to v within time interval $[t_\alpha, t_\omega]$, and L_v is a sorted list of pairs that keeps the foremost time from the source vertex x to v at different starting time. The concepts of *dominated element* and *dominates* in Algorithm 5 are defined in the same way as in Section 4.3.

In the first superstep, the source vertex x sets $f(x)$ to 0. Then, for every out-edge (x, u, t, λ) of x, if the edge can be used to grow a temporal path from x within the time interval $[t_\alpha, t_\omega]$, x first inserts the pair (t, t) into L_x, which indicates that there is a path from x starting at time t and arriving at x at time t. Then x sends out the pair $(t, t+\lambda)$ to its out-neighbor u, which indicates that there is a path from x starting at time t and arriving at u at time $(t+\lambda)$. For each other vertex $v \neq x$, the algorithm simply initializes $f(v)$ to be ∞ and L_v to be the empty set.

In each of the subsequent supersteps, if a vertex v receives messages from its neighbors, v is activated and processes the following (Lines 14-23). If a received pair (s, a) is not dominated by any pair in L_v, it may potentially give the minimum duration $(a - s)$, and hence we insert (s, a) into L_v. After we insert the pair (s, a), we also remove dominated elements from L_v. We then check whether $(a - s)$ is smaller than $f(v)$, and update $f(v)$ to $(a - s)$ if this is the case. Then, for each out-neighbor u of v, we send out a pair $(s, nextST((v, u), a))$ to u if u can be used to grow a temporal path within the time interval $[t_\alpha, t_\omega]$. The above superstep repeats until all vertices vote to halt and there is no more message pending for the next superstep, at which point $f(v)$ stores the final fastest-path duration from the source vertex x to v.

The following theorem shows that when Algorithm 5 terminates, $f(v)$ correctly keeps the fastest-path duration from x to v.

Theorem 5. *Algorithm 5 correctly computes the fastest-path duration from a source vertex x to every vertex $v \in V$ within the time interval $[t_\alpha, t_\omega]$.*

Proof. Suppose that the fastest path from x to v within $[t_\alpha, t_\omega]$ exists. Let the fastest path start from x at time t_x, and arrive at v at time t_y. Then, this is also an foremost path from x to v within the time interval $[t_x, t_y]$. By Lemma 6, there exists an foremost path $P = \langle x = v_1, v_2, \ldots, v_k, v_{k+1} = v\rangle$ from x to v such that every prefix-subpath, $P_i = \langle x, v_1, v_2, \ldots, v_i\rangle$, of P is an foremost path from x to v_i within $[t_x, t_y]$, for $1 \leq i \leq k+1$. Let $t_e(v_i)$ be the foremost time from x to v_i within $[t_x, t_y]$, for $1 \leq i \leq k+1$. Then, we have $nextST((v_i, v_{i+1}), t_e(v_i)) = t_e(v_{i+1})$, for $1 \leq i \leq k$. Let $e_1 = (v_1, v_2, t_1, \lambda_1)$ be the first edge on P, then, we have $t_1 = t_x$, and $t_1 + \lambda_1 = t_e(v_2)$.

We only need to show that the pair (t_x, t_y) is inserted into L_v, by which $f(v)$ is correctly computed as $(t_y - t_x)$ in Line 19. We prove that $(t_x, t_e(v_i))$ is inserted into L_{v_i}, for $1 \leq i \leq k+1$, by induction on i. When $i = 1$, $x = v_1$, because of edge $e_1 = (v_1, v_2, t_1, \lambda_1) = (v_1, v_2, t_x, \lambda_1)$, (t_x, t_x) is inserted into L_x in Line 9. Then, vertex x will send a message $<(t_x, t_x + \lambda_1)>$, i.e., $<(t_x, t_e(v_2))>$ to vertex v_2. Vertex v_2 will receive the message in the next superstep, and insert $<(t_x, t_e(v_2))>$ into L_{v_2}.

Now assume that after some supersteps, for $i = j$, where $j < k+1$, $(t_x, t_e(v_j))$ is inserted into L_{v_j}. Consider $i = j+1$ and we want to prove that $(t_x, t_e(v_{j+1}))$ is inserted into $L_{v_{j+1}}$. Vertex v_j will send a message $<(t_x, t_e(v_{j+1}))>$ to v_{j+1}. When v_{j+1} receives the message, to prove that $(t_x, t_e(v_{j+1}))$ will be inserted to $L_{v_{j+1}}$, we only need to show that $(t_x, t_e(v_{j+1}))$ will satisfy the condition in Line 15 (i.e., $(t_x, t_e(v_{j+1}))$ is not dominated by any pair in $L_{v_{j+1}}$). We prove this by contradiction. Assume that $(t_x, t_e(v_{j+1}))$ is dominated by some pair (s', a') in $L_{v_{j+1}}$. It indicates that there is a path $\hat{P}_{j+1}$ starting from x at time s', and arriving at v_{j+1} at time a', where (1) $s' > t_x$ and $a' \leq t_e(v_{j+1})$, or (2) $s' = t_x$ and $a' < t_e(v_{j+1})$. We first prove Case (1). If $s' > t_x$ and $a' \leq t_e(v_{j+1})$, we replace P_{j+1} in P by $\hat{P}_{j+1}$. The new path $\hat{P}$ is still a valid temporal path because $a' \leq t_e(v_{j+1})$. The duration of $\hat{P}$ is equal to $t_y - s'$, which is smaller than $t_y - t_x$. This leads to a contradiction that P is not a fastest path from x to v within $[t_\alpha, t_\omega]$. Next we prove Case (2). If $s' = t_x$ and $a' < t_e(v_{j+1})$, it indicates that P_{j+1} is not an foremost path from x to v_{j+1} within $[t_x, t_y]$, which is a contradiction. Thus, $(t_x, t_e(v_{j+1}))$ is not dominated by any pair in $L_{v_{j+1}}$. Hence, the condition in Line 15 is satisfied, and $(t_x, t_e(v_{j+1}))$ will be inserted to $L_{v_{j+1}}$. Thus, by induction, (t_x, t_y) will be inserted into L_v. □

6.3 A Graph Transformation Approach

In this section, we present a parallel graph transformation technique for computing the four types of minimum temporal paths.

6.3.1 Foremost Paths

We first discuss the parallel algorithm of computing single source foremost paths. Similar to Section 5.1, to compute foremost paths from a source vertex x to every vertex $v \in V$, we create a vertex x' in the transformed graph $\tilde{G}$ and a directed edge from x' to each vertex $(x, t) \in \tilde{V}_{out}(x)$ in $\tilde{G}$.

Then, we simply run the parallel *breadth-first search* (*BFS*) algorithm for a non-temporal graph [13] in $\tilde{G}$ from the source vertex x', that is, if a vertex is visited, it will not be activated any more. During the process, if the time t of a vertex (v, t) is not within 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 foremost time from x to v in G.

6.3.2 Reverse-Foremost Paths

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

6.3.3 Fastest Paths

We start a multiple-source parallel BFS algorithm for a non-temporal graph [13] as follows. Each vertex $(x, t) \in \tilde{V}_{out}(x)$ in the transformed graph $\tilde{G}$, where $t_\alpha \leq t \leq t_\omega$, is a source vertex for the parallel BFS. Each vertex (u, t') in $\tilde{G}$ keeps a

TABLE 2
Real Temporal Graphs (K $= 10^3$)

Dataset	$\|V\|$	$\|E_s\|$	$\|E\|$	$d_{avg}(u, G_s)$	$d_{avg}(u, G)$	π	$\|T_G\|$
arxiv	28 K	6,297 K	9,194 K	224.14	327.26	262	2,337
dblp	1,103 K	8,451 K	11,957 K	7.66	10.84	38	70
elec	8 K	104 K	107 K	12.50	12.90	5	101,012
enron	87 K	320 K	1,135 K	3.67	13.01	1,074	213,218
epin	132 K	841 K	841 K	6.38	6.38	1	939
fb	64 K	817 K	1,270 K	12.82	19.92	2	736,675
flickr	2,303 K	33,140 K	33,140 K	14.39	14.39	1	134
digg	30 K	85 K	86 K	2.80	2.84	25	82,641
slash	51 K	130 K	140 K	2.55	2.74	17	89,862
conflict	118 K	2,054 K	2,918 K	17.40	24.71	562	273,909
growth	1,871 K	39,953 K	39,953 K	21.36	21.36	1	2,198
youtube	3,224 K	9,377 K	12,224 K	2.91	3.80	2	203
flow	103,661 K	776,456 K	3,018,067 K	7.49	29.11	1,643,088	113,899,690

field t_m. For any vertex (u, t'), t_m records the largest time instance t of any source vertex $(x, t) \in \tilde{V}_{out}(x)$ that can reach (u, t') in $\tilde{G}$. During the computation, only if the value t_m of a vertex (u, t') is updated, it sends out a message t_m to its out-neighbors. The duration from x to every vertex (u, t') is computed as $(t' - t_m)$. The duration of the fastest path from x to every v in G is the minimum $(t' - t_m)$ among all the vertices (v, t') in $\tilde{V}_{in}(v)$.

6.3.4 Shortest Paths

For the source vertex x in G, we create a vertex x' in the transformed graph $\tilde{G}$ and a directed edge from x' to each vertex $(x, t) \in \tilde{V}_{out}(x)$ in $\tilde{G}$. Then, we run the parallel single source shortest-path algorithm for a non-temporal graph [13] from the source vertex x' in $\tilde{G}$. Among the shortest-path from x' to each $(v, t) \in \tilde{V}_{in}(v)$, the minimum distance is the shortest-path distance from x to v in G.

7 Experimental Results

We evaluate the performance of our algorithms and examine the usefulness of minimum temporal paths in this section. We ran the sequential algorithms on a machine running Linux on an Intel 3.3 GHz CPU and 16 GB RAM. We implemented our parallel algorithms in Pregel+ [15] and ran on a cluster of 15 machines, where each machine has 24 cores (two 2.0 GHz Intel Xeon E5-2620 CPU) and 48 GB RAM, running 64-bit CentOS 6.5 with Linux kernel 2.6.32. The connectivity between any pair of nodes in the cluster is 1 Gbps.

Temporal Graphs. We used 13 real temporal datasets in our experiments, 12 of them are from the Koblenz Large Network Collection,[1] 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 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; `delicious-ut` (`delicious`) from the network of 'delicious'; `edit-enwiki` (`edit`) from the edit network of the English Wikipedia; `network-flow` (`flow`) is the Yahoo! Network Flows Data,[2] version 1.0. Note that `flow` cannot fit in the main memory of a single machine.

Table 2 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 $\pi = 1$ does not imply that the temporal graph is similar to the corresponding non-temporal 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 273,909 snapshots.

7.1 Efficiency of Sequential SSMTP Algorithms

To evaluate the performance of our sequential algorithms for computing single-source minimum temporal paths, we compare with the algorithms proposed by Xuan et al. [7], denoted by **Xuan**. Note that Xuan can only report the number of hops for shortest paths. Xuan et al. also did not study reverse-foremost paths and we modified their foremost path algorithm to compute reverse-foremost time. We denote our one-pass algorithms presented in Section 4 by *one-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 100 randomly selected source vertices, and set $[t_\alpha, t_\omega]$ to be $[0, \infty]$ in this experiment.

Tables 3 reports the average running time of the algorithms. We also show the size of the transformed graph $\tilde{G}$ in

1. http://konect.uni-koblenz.de/

2. http://webscope.sandbox.yahoo.com/catalog.php?datatype=g

TABLE 3
Running Time in Seconds

	Foremost			Reverse-Foremost			Fastest			Shortest		
	1-pass	Trans	Xuan	1-pass	Trans	Xuan	1-pass	Trans	Xuan	1-pass	Trans	Xuan
arxiv	0.0109	0.0283	0.1636	0.0120	0.0549	0.1685	0.1159	0.0264	1.5567	0.1001	0.0953	1.6510
dblp	0.0208	0.0689	0.2860	0.0220	0.0820	0.6443	0.1458	0.0693	0.7573	0.1523	0.5703	1.8762
elec	0.0002	0.0031	0.0016	0.0002	0.0026	0.0016	0.0006	0.0026	0.0302	0.0006	0.0162	0.0127
enron	0.0014	0.0187	0.0099	0.0015	0.0199	0.0095	0.0076	0.0154	0.1726	0.0069	0.0838	0.0378
epin	0.0011	0.0091	0.0214	0.0012	0.0032	0.0172	0.0039	0.0086	0.0202	0.0048	0.0415	0.1278
fb	0.0021	0.0232	0.0113	0.0020	0.0089	0.0087	0.0094	0.0180	0.3845	0.0093	0.1135	0.1716
flickr	0.0591	0.5692	1.1019	0.0678	0.2759	1.0379	0.4675	0.5288	3.8578	0.5014	2.4240	11.2012
digg	0.0001	0.0001	0.0017	0.0001	0.0001	0.0019	0.0003	0.0001	0.0059	0.0003	0.0070	0.0076
slash	0.0002	0.0033	0.0044	0.0003	0.0058	0.0049	0.0010	0.0032	0.0165	0.0011	0.0213	0.0262
conflict	0.0042	0.1425	0.0504	0.0047	0.1624	0.0482	0.0175	0.1371	0.0908	0.0227	0.3346	0.4253
growth	0.1432	4.2159	1.6286	0.1676	6.0869	1.9096	1.3442	3.8759	6.2699	1.5027	11.0289	26.3161
youtube	0.0326	0.0628	0.2207	0.0355	0.0105	0.1752	0.1175	0.0560	0.8352	0.1193	0.6535	1.5675

Table 4. The results show that the one-pass algorithms are significantly faster than Xuan for all the 12 datasets. On average, the one-pass algorithms are 13 to 18 times faster than Xuan. The reason for this big difference in running time is mainly because the one-pass algorithms have lower time complexity than Xuan in general, which we briefly explain as follows (more details can be found in Appendix O, available in the online supplemental material). First, the time complexity of Xuan for computing foremost and reverse-foremost time is $O(m(\log\pi + \log n))$, while that of one-pass is $O(n + M)$. However, in real-world temporal networks, M is not much larger than m (see Table 2). Also, one-pass only reads the graph once sequentially, while Xuan requires random accesses and needs to maintain a heap. Second, to compute the fastest paths, the time complexity of Xuan is $O(mn\pi(\log\pi + \log n))$, while that of one-pass is $O(m\pi(\log\pi + \log n))$. Thus, one-pass has much lower time complexity than Xuan and hence also much better performance. Third, to compute the shortest paths, the time complexity of Xuan is $O(mn \log\pi)$, while that of one-pass is $O(m\pi(\log\pi + \log n))$. Thus, our algorithm has lower complexity and hence also better performance.

Although both one-pass and Trans have linear time complexity for computing foremost and reverse-foremost paths, one-pass is significantly faster than Trans for processing many datasets. This is mainly because of the following two reasons. First, one-pass performs fewer operations in the path computation than Trans; for example, Trans needs to maintain a queue data structure during the path computation. Second, the transformed graph is larger than the original temporal graph, as shown in Tables 2 and 4. In fact, the results show that for Trans, there is no big difference between the running time of foremost/reverse-foremost paths and the running time of fastest paths, because the Trans algorithms for computing these three types of paths are all based on BFS and have the same complexity. Thus, one-pass is preferred for computing foremost and reverse-foremost paths.

For computing fastest paths, however, there are a number of cases Trans is faster than one-pass. This is mainly because for computing fastest paths, the time complexity of Trans is lower than that of one-pass, but the tradeoff is that the transformed graph is larger than the original temporal graph. As a result, although theoretically the Trans algorithm has lower time complexity than the one-pass algorithm, practically the one-pass algorithm may have better performance in some cases mainly because the input to the one-pass algorithm is smaller, i.e., the original temporal graph is smaller than the corresponding transformed graph. Thus, when the size of the transformed graph is not too much larger than that of the original temporal graph, Trans is often faster and should be used instead of one-pass for computing fastest paths; otherwise, one-pass should be used.

For computing shortest paths, the time complexity of one-pass and Trans are similar. However, one-pass is faster than Trans in all cases except for the arxiv dataset, which can be explained by the fact that the size of the transformed graph is comparable with that of the original temporal graph for arxiv. Thus, the one-pass algorithm should be used for computing shortest paths except that when the size of the transformed graph is comparable with that of the original temporal graph.

7.2 Efficiency of Parallel SSMTP Algorithms

We denote our parallel algorithms presented in Section 6.1 to Section 6.2 by *para*, and the graph-transformation-based parallel algorithms presented in Section 6.3 by *paraT*.

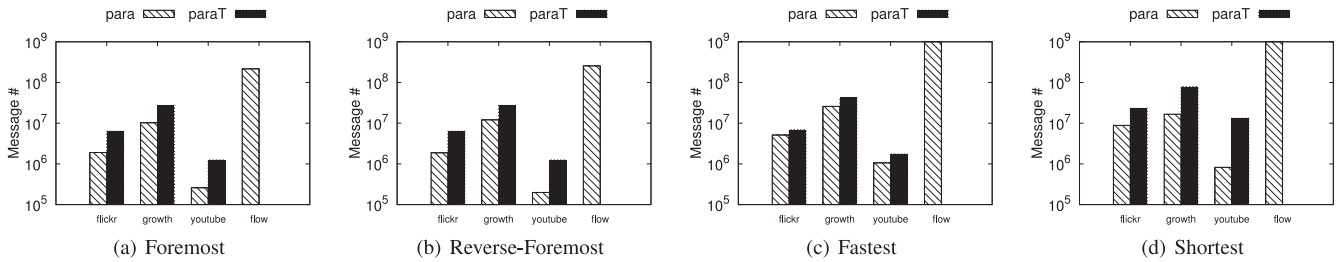
Table 5 reports the average running time of the parallel algorithms for computing the four types minimum temporal paths from 100 randomly selected source vertices. Since the dataset flow cannot fit in main memory, we cannot get its transformed graph, and thus not able to run paraT on flow. The results show that our parallel algorithm para is significantly faster than paraT in computing all the four types of SSMTPs. On average, para is around one order of

TABLE 4
Size of the Transformed Graph $\tilde{G}$

	arxiv	dblp	elec	enron	epin	fb
$\lvert\tilde{V}\rvert$	433 K	5,553 K	212 K	1,367 K	482 K	1,637 K
$\lvert\tilde{E}\rvert$	9,759 K	16,977 K	313 K	2,505 K	1,219 K	3,037 K
	flickr	digg	slash	conflict	growth	youtube
$\lvert\tilde{V}\rvert$	12,600 K	172 K	273 K	3,191 K	34,815 K	11,498 K
$\lvert\tilde{E}\rvert$	44,358 K	233 K	381 K	6,009 K	77,196 K	21,140 K

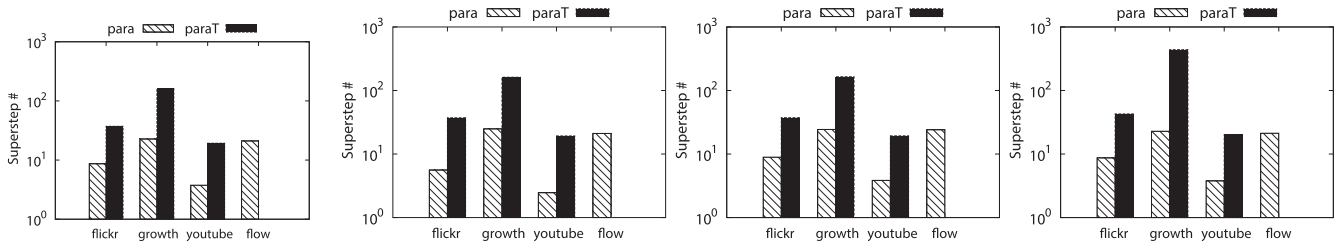
TABLE 5
Running Time in Seconds of the Para and ParaT Algorithms

	Foremost		Reverse-Foremost		Fastest		Shortest	
	para	paraT	para	paraT	para	paraT	para	paraT
flickr	0.4123	1.8051	0.2671	1.8180	0.9941	1.8247	0.5294	2.2137
growth	1.0346	9.3117	1.1756	9.2801	2.7393	9.6183	1.2670	25.6463
youtube	0.1864	0.9284	0.1351	0.9205	0.1794	0.9195	0.1804	1.0534
flow	8.8696	-	11.0731	-	596.2914	-	120.8690	-



(a) Foremost (b) Reverse-Foremost (c) Fastest (d) Shortest

Fig. 3. The number of messages of the para and paraT algorithms.



(a) Foremost (b) Reverse-Foremost (c) Fastest (d) Shortest

Fig. 4. The number of supersteps of the para and paraT algorithms.

magnitude faster than paraT, which can be explained as follows.

There are four main costs in running an algorithm in Pregel-like systems [16]: (1) computation cost per superstep; (2) communication cost per superstep; (3) memory used per superstep; and (4) the total number of supersteps. The cost of (1) is linear (or multiplied by a log factor) to the size of the input graphs for both the para and paraT algorithms, because the algorithms are simple breadth-first parallel traversal algorithms. The cost of (1) is not the dominant cost since the path computation tasks are not computation-intensive but communication-intensive. As reported in Fig. 3, the total number of messages (i.e., the communication cost) required by the para and paraT algorithms are large, which is the dominating factor in the total cost. Fig. 3 shows the total number of messages (averaged over the four types of SSMTPs) of the para and paraT algorithms, from which we can see that paraT sends considerably larger number of messages than para in all cases. This is because many messages in paraT are sent from the vertex copies of a vertex to other vertex copies (with different timestamps) of the same vertex in the transformed graph, e.g., from $(a, 1)$ to $(a, 2)$ in Fig. 2b; on the other hand, para runs on the original temporal graph and hence there are no such messages from the same vertex to its copies. Fig. 4 presents the number of supersteps taken by the para and paraT algorithms (averaged over the four types of SSMTPs), which reveals another main performance bottleneck of the paraT algorithms, i.e., the paraT algorithms use significantly larger number of supersteps than the para algorithms in all cases. This is because paraT runs on the transformed graph, in which messages propagate along a much longer path from one vertex to another vertex. For example, it takes only two steps to propagate a message from a to g by para in the temporal graph in Fig. 2a, but it takes at least five steps to propagate a message from a to g by paraT in the transformed graph in Fig. 2b. For parallel computing, the elapsed running time at each superstep is determined by the time taken by the slowest worker. Since each superstep of computation is synchronized, the total running time is the summation of the running time of the slowest worker at each superstep. Thus, the total running time can be significantly longer if some of the supersteps have a slow worker. Finally, the space used by the para and paraT algorithms are comparable, which we report in Fig. 6 in Appendix M, available in the online supplemental material.

We also observed that for paraT, the running time of computing foremost paths, reverse-foremost paths, and fastest paths are similar, because all of them are computed using algorithms similar to a parallel BFS algorithm. The running time of computing shortest paths with paraT is considerably longer, because the algorithm runs a parallel single-source shortest-path algorithm which takes more supersteps.

7.3 Scalability of Parallel SSMTP Algorithms

In this experiment, we test the scalability of our parallel algorithms. Since para is much faster than paraT, and the synthetic datasets cannot fit in memory in order to generate the

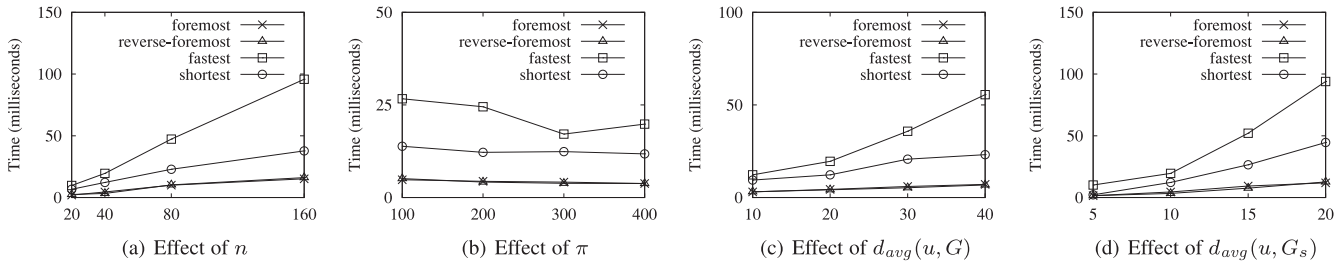


Fig. 5. Performance on synthetic graphs with different values of n, π, $d_{avg}(u, G)$, and $d_{avg}(u, G_s)$.

transformed graphs, we only study the scalability of para. We generate synthetic temporal graphs with different number of vertices n, different π value, different average degree in G_s (denoted by $d_{avg}(u, G_s)$) and in G (denoted by $d_{avg}(u, G)$). As default values, we set $n = 40M$ $(M = 10^6)$, $\pi = 200$, $d_{avg}(u, G_s) = 10$, $d_{avg}(u, G) = 20$, and $|T_G| = 100{,}000$. Then, we vary the value of n, π, $d_{avg}(u, G)$ and $d_{avg}(u, G_s)$, respectively, while fixing other parameters as their default values.

As Fig. 5a shows, when we increase the number of vertices from 20 to 160 M, the running time of para increases approximately linearly with the number of vertices. Similarly, Figs. 5c and 5d show that the running time of para increases linearly when $d_{avg}(u, G)$ and $d_{avg}(u, G_s)$ increase. However, Fig. 5b shows that the running time of para is not significantly affected by the value of π. This is because π is the maximum number of temporal edges between any two vertices, while the average number of temporal edges between any two vertices in the graph remains the same as determined by the default values $d_{avg}(u, G_s) = 10$ and $d_{avg}(u, G) = 20$. As a result, the total number of messages sent by the parallel algorithms remain roughly the same for different values of π. This also shows that our parallel algorithms are efficient in the case when some vertices (in some workers) have skewed number of temporal edges. Thus, the results verify the scalability of our parallel algorithm with respect to the increases in n, π, $d_{avg}(u, G)$, and $d_{avg}(u, G_s)$.

8 Related Work

A preliminary version of this paper [10] proposed sequential algorithms for SSMTP, which are not scalable for handling large graphs. Although the one-pass algorithms only need to scan the input graph once, they still have high memory consumption. In this paper we propose scalable parallel algorithms as a solution.

The closest related work is [7], 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 foremost 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 lower time complexity in general (see a detailed analysis in Appendix O, available in the online supplemental material). Our experimental results also verify that our algorithms are one to two orders of magnitude faster than theirs on average. Our previous work on temporal graph traversals [17] can also be applied to compute minimum temporal paths with linear time complexity, but it is not clear how the algorithms can be parallelized for handling large graphs.

Temporal paths were applied to study the connectivity of a temporal network [6], for which disjoint temporal paths between any two vertices are computed. In [18], a similar definition of reverse-foremost path (without the information of λ for the edges) was proposed to study information latency. In [19], four definitions of temporal proximity were introduced, which are no more than finding foremost, reverse-foremost, and fastest paths, but they also did not consider the information of λ. In [20], [21], the foremost time was applied to define metrics such as temporal efficiency and temporal clustering coefficient. Temporal paths were also applied to find temporal connected components in [2], [21]. In [22], small-world behavior was analyzed in temporal networks using temporal paths. In [23], betweenness and closeness based on the three types of temporal paths in [7] were discussed. In [24], empirical studies were conducted to measure correlation between temporal paths and closeness defined based on foremost 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 [11], [25], indexing techniques were proposed to answer point-point path queries, which are expensive for computing single-source minimum temporal paths. In [26], 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], [27], [28], [29] 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 non-temporal graphs, but we have shown that the concept of shortest path in temporal graphs is very different from that in non-temporal graphs. The other three types, i.e., foremost paths, reverse-foremost 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. We next proposed an alternative solution that transforms a temporal graph into a non-temporal one with no information loss. For processing very large temporal graphs, both the one-pass and graph-transformation-based algorithms may require too much memory and may not scale, hence we proposed scalable

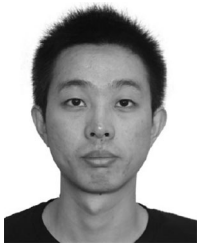
parallel algorithms for computing the minimum temporal paths. Experiments on a wide range of real-world temporal graphs show that both of our sequential algorithms are one order to two orders of magnitude faster than the existing algorithms [7]. For processing large temporal graphs, the results show the scalability of our parallel algorithms.

Acknowledgments

The authors thank the reviewers for their valuable comments. The authors are supported by the Hong Kong GRF 2150851 and ITF 6904079, Grants 3132964 and 3132821 funded by the Research Committee of CUHK, the AcRF Tier-1 Grant (RG135/14) from the Ministry of Education of Singapore, National Natural Science Foundation of China-Guangdong Government Joint Funding (2nd) for Super Computer Application Research, and the National Natural Science Foundation of China (U1501252).

References

[1] P. Holme and J. Saramäki, "Temporal networks," *CoRR abs/1108.1780*, 2011.
[2] V. Nicosia, J. Tang, M. Musolesi, G. Russo, C. Mascolo, and V. Latora, "Components in time-varying graphs," *Chaos*, vol. 22, 2012, Art. no. 023101.
[3] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating betweenness centrality," in *Proc. 5th Int. Workshop Algorithms Models Web-Graph*, 2007, pp. 124–137. [Online]. Available: http://dx.doi.org/10.1007/978–3-540-77004-6_10
[4] D. Eppstein and J. Wang, "Fast approximation of centrality," *J. Graph Algorithms Appl.*, vol. 8, pp. 39–45, 2004. [Online]. Available: http://jgaa.info/accepted/2004/EppsteinWang2004.8.1.pdf
[5] M. J. Rattigan, M. E. Maier, and D. Jensen, "Graph clustering with network structure indices," in *Proc. 24th Int. Conf. Mach. Learn.*, 2007, pp. 783–790. [Online]. Available: http://doi.acm.org/10.1145/1273496.1273595
[6] D. Kempe, J. M. Kleinberg, and A. Kumar, "Connectivity and inference problems for temporal networks," *J. Comput. Syst. Sci.*, vol. 64, no. 4, pp. 820–842, 2002.
[7] B.-M. B. Xuan, A. Ferreira, and A. Jarry, "Computing shortest, fastest, and foremost journeys in dynamic networks," *Int. J. Found. Comput. Sci.*, vol. 14, no. 2, pp. 267–285, 2003.
[8] A. Casteigts, P. Flocchini, B. Mans, and N. Santoro, "Measuring temporal lags in delay-tolerant networks," *IEEE Trans. Comput.*, vol. 63, no. 2, pp. 397–410, Feb. 2014. [Online]. Available: http://dx.doi.org/10.1109/TC.2012.208
[9] A. Casteigts, P. Flocchini, B. Mans, and N. Santoro, "Shortest, fastest, and foremost broadcast in dynamic networks," *Int. J. Found. Comput. Sci.*, vol. 26, no. 4, pp. 499–522, 2015. [Online]. Available: http://dx.doi.org/10.1142/S0129054115500288
[10] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, "Path problems in temporal graphs," *Proc. VLDB Endowment*, vol. 7, no. 9, pp. 721–732, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol7/p721-wu.pdf
[11] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou, "Efficient route planning on public transportation networks: A labelling approach," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 967–982. [Online]. Available: http://doi.acm.org/10.1145/2723372.2749456
[12] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, "Path problems in temporal graphs (preliminary version)," 2013. [Online]. Available: www.cse.cuhk.edu.hk/%7ehhwu/temsp.pdf
[13] G. Malewicz, et al., "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184
[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.
[15] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proc. 24th Int. Conf. World Wide Web*, 2015, pp. 1307–1317.
[16] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu, "Pregel algorithms for graph connectivity problems with performance guarantees," *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 1821–1832, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol7/p1821-yan.pdf
[17] S. Huang, J. Cheng, and H. Wu, "Temporal graph traversals: Definitions, algorithms, and applications," *CoRR abs/1401.1919*, 2014.
[18] G. Kossinets, J. M. Kleinberg, and D. J. Watts, "The structure of information pathways in a social communication network," in *Proc. 14th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2008, pp. 435–443.
[19] V. Kostakos, "Temporal graphs," *Physica A: Statistical Mechanics Appl.*, vol. 388, no. 6, pp. 1007–1023, 2009.
[20] J. Tang, M. Musolesi, C. Mascolo, and V. Latora, "Temporal distance metrics for social network analysis," in *Proc. ACM Workshop Online Social Netw.*, 2009, pp. 31–36.
[21] J. Tang, M. Musolesi, C. Mascolo, and V. Latora, "Characterising temporal distance and reachability in mobile and online social networks," *Comput. Commun. Rev.*, vol. 40, no. 1, pp. 118–124, 2010.
[22] J. Tang, S. Scellato, M. Musolesi, C. Mascolo, and V. Latora, "Small-world behavior in time-varying graphs," *Physical Rev. E*, vol. 81, no. 5, 2010, Art. no. 055101.
[23] 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.
[24] R. K. Pan and J. Saramäki, "Path lengths, correlations, and centrality in temporal networks," *Physical Rev. E*, vol. 84, 2011, Art. no. 016105.
[25] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke, "Reachability and time-based path queries in temporal graphs," in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2016, pp. 145–156.
[26] L. Xiang, et al., "Temporal recommendation on graphs via long- and short-term preference fusion," in *Proc. 16th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2010, pp. 723–732.
[27] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro, "Time-varying graphs and dynamic networks," *Int. J. Parallel Emergent Distrib. Syst.*, vol. 27, no. 5, pp. 387–408, 2012.
[28] O. Michail, "An introduction to temporal graphs: An algorithmic perspective," in *Proc. Algorithms Probability Netw. Games*, 2015, pp. 308–343.
[29] M. Müller-Hannemann, F. Schulz, D. Wagner, and C. D. Zaroliagis, "Timetable information: Models and algorithms," in *Proc. Int. Dagstuhl Workshop Algorithmic Methods Railway Optimization*, 2004, pp. 67–90. [Online]. Available: http://dx.doi.org/10.1007/978–3-540-74247-0_3



Huanhuan Wu received the bachelor's degree in computer science and technology from Zhejiang University. He is currently working toward the PhD degree in the Department of Computer Science and Engineering, Chinese University of Hong Kong. His research interests include analysis in massive temporal graphs and non-trivial distributed algorithms.

James Cheng is in the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong. His current research interests include big data infrastructures and applications, distributed computing, distributed machine learning, and large-scale network analysis.

Yiping Ke is an assistant professor in the School of Computer Science and Engineering, Nanyang Technological University, Singapore. Her research interests include big data analysis, data mining, and data management.

Silu Huang received the bachelor's degree in electronic engineering from Shanghai Jiao Tong University, and the MPhil degree in computer science and engineering from the Chinese University of Hong Kong. She is currently working toward the PhD degree in computer science at the University of Illinois, Urbana Champaign. Her research interests include large-scale network analysis and data management.

Yuzhen Huang received the bachelor's degree in computer science from Sun Yat-sen University. He is currently working toward the PhD degree in the Department of Computer Science and Engineering, Chinese University of Hong Kong. His research interests include big data and distributed machine learning.

Hejun Wu received the PhD degree from the Hong Kong University of Science and Technology. He is an associate professor in the Department of Computer Science, Sun Yat-sen University, China. His research interests include distributed databases, embedded systems, and cloud computing.