Reachability and Time-Based Path Queries in Temporal Graphs

Huanhuan Wu*, Yuzhen Huang*, James Cheng*, Jinfeng Li*, Yiping Ke#
*Department of Computer Science and Engineering, The Chinese University of Hong Kong
Emails: {hhwu, yzhuang, jcheng, jfli}@cse.cuhk.edu.hk
#School of Computer Engineering, Nanyang Technological University
Email: ypke@ntu.edu.sg

***Abstract*—A temporal graph is a graph in which vertices communicate with each other at specific time, e.g., A calls B at 11 a.m. and talks for 7 minutes, which is modeled by an edge from A to B with starting time "11 a.m." and duration "7 mins". Temporal graphs can be used to model many networks with time-related activities, but efficient algorithms for analyzing temporal graphs are severely inadequate. We study fundamental problems such as answering reachability and time-based path queries in a temporal graph, and propose an efficient indexing technique specifically designed for processing these queries in a temporal graph. Our results show that our method is efficient and scalable in both index construction and query processing.**

I. Introduction

Graph has been extensively used to model and study the structures of various online social networks, mobile communication networks, e-commerce networks, email networks, etc. In these graphs, vertices are users or companies, while edges model the relationship between them. However, there is one type of information that is often missing in these graphs for simplicity of analysis: in reality, a relationship occurs at a specific time and lasts for a certain period. Formally, this can be modeled as a **temporal graph**, in which each edge is represented by (u, v, t, λ), indicating that the relationship from u to v starts at time t and lasts for a duration of λ. There may be multiple edges between u and v indicating their relationship occurring in different time periods.

Temporal graphs can be used to model and study many time-related activities in the above-mentioned graphs. For example, users follow or tag other users in different periods in online social networks; friends chat with each other in different time periods in mobile phone networks; people send messages to each other at different times in email networks or instant messaging networks; customers buy products from sellers at different times in online shopping platforms, or different types of transactions happened between different parties in different periods in e-commerce networks, etc.

Research on **non-temporal graphs** (i.e., general graphs without time information) has been extensively studied. However, for temporal graphs, even some fundamental problems have not been well studied (e.g., graph traversal, connected components, reachability, "shortest paths", etc.). In this paper, we study the problems of computing the reachability and the "shortest path distance" from a vertex to another vertex in a temporal graph.

Graph reachability and shortest path both have numerous important applications. However, in a temporal graph, the problems become more complicated due to the order imposed by time. For example, consider a toy train-schedule graph shown in Figure 1(a), where the number next to each edge is the day (e.g., Day 1, Day 2, etc.) that a train departs, and assume that the duration of each train takes 2 days. Suppose now one wants to travel from `a` to `d`. If he chooses to go to `d` via `b`, then he can leave either on Day 1 or Day 2, and he will reach `d` on Day 6. Now suppose that he wants to depart later, on Day 4, then he cannot reach `d` because the train departing on Day 4 from `a` reaches `c` on Day 6, but the train leaves from `c` on Day 5 to `d`. However, if we do not consider the time information, then the traveler may still take the train on Day 4 from `a` to `c`, not realizing that he would not catch the train from `c` to `d` in this case.

Given two vertices, u and v, in a temporal graph, and a time interval T, we study how to compute (1) whether u can reach v within T, (2) the *earliest time* u can *reach* v within T, and (3) the *duration of a fastest path* from u to v within T. The *reachability*, *earliest-arrival time*, and *minimum duration* from source vertices to target vertices have been found useful in the study of temporal networks such as temporal graph connectivity [1], temporal betweenness and closeness [2], [3], temporal connected components in [4], information propagation [5], information latency [6], [7], temporal efficiency and clustering coefficient [4], temporal small-world behavior study [8], etc.

The above cited works, however, did not focus on the design of efficient algorithms to compute reachability, earliest-arrival time and minimum duration, and their results were mostly obtained from small temporal graphs. Wu et al. [9] made a significant improvement over existing algorithms [10] and their algorithms can handle much larger temporal graphs than existing works. However, their algorithms were not designed for online querying, while in many applications it is demanding to find the reachability, earliest-arrival time or minimum duration from a source vertex to a target vertex in real time. Wang et al. [11] presented an indexing method to answer online queries of earliest-arrival time or minimum duration from a source vertex to a target vertex. However, their indexing method cannot scale to large temporal graphs. In addition, their indexing method does not support dynamic update, which is practically important for temporal graphs since updates are frequent in most real-world temporal graphs. In view of this, we propose an index to support efficient online querying for large temporal graphs, which also supports efficient dynamic update.

Our method first transforms a temporal graph into a new graph which is a *directed acyclic graph* (*DAG*), on which existing indexing methods for reachability querying [12], [13], [14], [15], [16], [17], [18], [19], [20] can also be applied. However, this DAG is often significantly larger than the DAGs that are handled by existing methods. It also possesses unique properties of temporal graphs, while all existing methods were designed for handling non-temporal graphs. Thus, more scalable methods that also consider the properties of temporal graphs need to be designed.

We propose **TopChain**, which is a labeling scheme for answering reachability queries. A labeling scheme, e.g., 2-hop label [21], constructs two labels for each vertex v, $L_{in}(v)$ and $L_{out}(v)$, where $L_{in}(v)$ and $L_{out}(v)$ are the set of vertices that can reach v and that are reachable from v, respectively. A query whether u can reach v is answered by intersecting $L_{out}(u)$ and $L_{in}(v)$, since there exists a common vertex in $L_{out}(u)$ and $L_{in}(v)$ if u can reach v. However, $L_{in}(v)$ and $L_{out}(v)$ are often too large, and various methods have been proposed to reduce their sizes [12], [13], [11], [18], [20].

TopChain decomposes an input DAG into a set of chains [22], [23], where a chain is an ordered sequence of vertices such that each vertex can reach the next vertex in the chain. Thus, $L_{in}(v)$ and $L_{out}(v)$ only need to keep the last and first vertex in a chain that can reach v and that is reachable from v, respectively. However, the number of chains can still be too large for a large graph, and as a solution, TopChain ranks the chains and only uses the top k chains for each vertex. In this way, the size of the labels is kept to at most $2k$ for each vertex, and index construction takes only linear time, as k is a small constant. The k labels may not be able to answer every query, and thus online search may still be required. However, the labels can be employed to do effective pruning and online search converges quickly.

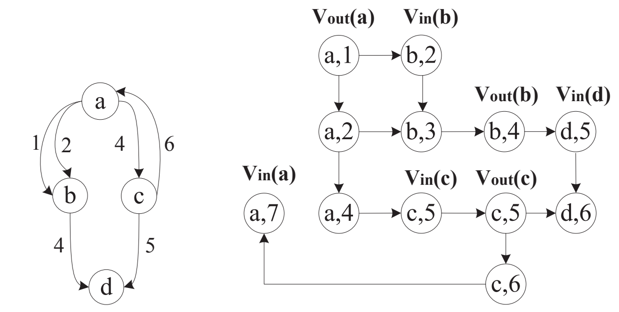
The contributions of our work are summarized as follows:

- We propose an efficient indexing method, TopChain, for answering reachability and time-based path queries in a temporal graph, which is useful for analyzing real-world networks with time-based activities.
- TopChain has a linear index construction time and linear index size. Although existing methods can be applied to our transformed graph for answering reachability queries, our method is the only one that makes use of the properties of a temporal graph to design the indexing scheme. TopChain also applies the properties of temporal graphs to devise an efficient algorithm for dynamic update of the index.
- We evaluated the performance of TopChain on a set of 15 real temporal graphs. Compared with the state-of-the-art reachability indexes [14], [16], [17], [19], [20], TopChain is from a few times to a few orders of magnitude faster in query processing, with a smaller or comparable index size and index construction cost.

Paper outline. Section II defines the problem. Section III describes graph transformation. Sections IV and V present the details of indexing and query processing. Section VI presents some improvements on labeling. Section VII reports experimental results. Section VIII discusses related work and Section IX gives the concluding remarks.

Table I. FREQUENTLY-USED NOTATIONS

Notation	Description
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	A temporal graph
$e = (u, v, t, \lambda) \in \mathcal{E}$	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 $\mathcal{G}$
$\Gamma_{out}(u, \mathcal{G})$ ($\Gamma_{in}(u, \mathcal{G})$)	The set of out-neighbors (in-neighbors) of a vertex u in $\mathcal{G}$
$d_{out}(u, \mathcal{G})(d_{in}(u, \mathcal{G}))$	The out-degree (in-degree) of a vertex u in $\mathcal{G}$
$G = (V, E)$	A directed acyclic graph (DAG)
$\mathbf{T}_{out}(v)$ ($\mathbf{T}_{in}(v)$)	The set of distinct starting (arrival) time of out-edges (in-edges) of v
$V_{out}(v)$ ($V_{in}(v)$)	The set of vertices, $\langle v, t \rangle$, for each $t \in \mathbf{T}_{out}(v)$ ($\mathbf{T}_{in}(v)$)
$L_{out}(v)$ ($L_{in}(v)$)	The set of out-labels (in-labels) of a vertex v
$\mathbb{C} = \{C_1, \ldots, C_l\}$	A chain cover of G
$code(v)$	Chain code of a vertex v
$RC(v)$ ($RC^{-1}(v)$)	The set of chains that v can reach (can reach v)
$first_v(C)$ ($last_v(C)$)	The first (last) vertex in C that v can reach (can reach v)
$RF(v)$ ($RL(v)$)	Set of first (last) reachable vertices in $RC(v)$ ($RC^{-1}(v)$)
$RFcode(v)$ ($RLcode(v)$)	Set of chain codes of vertices in $RF(v)$ ($RL(v)$)
$top_k(RFcode(v))$ ($top_k(RLcode(v))$)	The first k chain codes in $RFcode(v)$ ($RLcode(v)$)



(a) Temporal graph (b) Transformed graph

Figure 1. A temporal graph $\mathcal{G}$ and its transformed graph G

II. Problem Definition

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a temporal graph, where $\mathcal{V}$ is the set of vertices in $\mathcal{G}$ and $\mathcal{E}$ is the set of edges in $\mathcal{G}$. An edge $e \in \mathcal{E}$ is a quadruple (u, v, t, λ), where $u, v \in \mathcal{V}$, t is the **starting time**, and λ is the **traversal time** to go from u to v starting at time t. We denote the starting time of e by $t(e)$ and the traversal time of e by $\lambda(e)$. Alternatively, we can consider that e is **active** during the period $[t, t + \lambda]$.

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.

We denote the set of temporal edges from u to v in $\mathcal{G}$ by $\Pi(u, v)$, and the number of temporal edges from u to v in $\mathcal{G}$ 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 $\mathcal{G}$, by $\pi = \max\{\pi(u, v) : (u, v) \in (\mathcal{V} \times \mathcal{V})\}$.

We define the set of **out-neighbors** of a vertex u in $\mathcal{G}$ as $\Gamma_{out}(u, \mathcal{G}) = \{v : (u, v, t, \lambda) \in \mathcal{E}\}$, and the **out-degree** of u in $\mathcal{G}$ as $d_{out}(u, \mathcal{G}) = \sum_{v \in \Gamma_{out}(u, \mathcal{G})} \pi(u, v)$. Similarly, we define the **in-neighbors** and **in-degree** of u as $\Gamma_{in}(u, \mathcal{G}) = \{v : (v, u, t, \lambda) \in \mathcal{E}\}$ and $d_{in}(u, \mathcal{G}) = \sum_{v \in \Gamma_{in}(u, \mathcal{G})} \pi(v, u)$.

A **temporal path** P in a temporal graph $\mathcal{G}$ is a sequence of edges $P = \langle e_1, e_2, \ldots, e_p \rangle$, such that $e_i = (v_i, v_{i+1}, t_i, \lambda_i)$

$\in \mathcal{E}$ is the i-th temporal edge on P for $1 \leq i \leq p$, and $(t_i + \lambda_i) \leq t_{i+1}$ for $1 \leq i < p$. Note that for the last edge $(v_p, v_{p+1}, t_p, \lambda_p)$ on P, we do not put a constraint on $(t_p + \lambda_p)$ since t_{p+1} is not defined for the path P. In fact, $(t_p + \lambda_p)$ 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 $dura(P) = end(P) - start(P)$.

Based on the temporal paths, we give the definitions of minimum temporal paths [9] and temporal reachability as follows.

Definition 1 (Minimum Temporal Paths [9]): Let $\mathbf{P}(u, v, [t_\alpha, t_\omega]) = \{P : P$ is a temporal path from u to v such that $start(P) \geq t_\alpha$, $end(P) \leq t_\omega\}$.

A temporal path $P \in \mathbf{P}(u, v, [t_\alpha, t_\omega])$ is an **earliest-arrival path** if $end(P) = \min\{end(P') : P' \in \mathbf{P}(u, v, [t_\alpha, t_\omega])\}$. The **earliest-arrival time** to reach v from u within $[t_\alpha, t_\omega]$ is given by $end(P)$.

A temporal path $P \in \mathbf{P}(u, v, [t_\alpha, t_\omega])$ is a **fastest path** if $dura(P) = \min\{dura(P') : P' \in \mathbf{P}(u, v, [t_\alpha, t_\omega])\}$. The **minimum duration** taken to go from u to v within $[t_\alpha, t_\omega]$ is given by $dura(P)$.

Definition 2 (Temporal Reachability): Given two vertices u and v, and a time interval $[t_\alpha, t_\omega]$, u can reach v (or v is reachable from u) within $[t_\alpha, t_\omega]$ if $\mathbf{P}(u, v, [t_\alpha, t_\omega]) \neq \emptyset$, i.e., there exists a temporal path P from u to v such that $start(P) \geq t_\alpha$ and $end(P) \leq t_\omega$.

Example 1: Figure 1(a) shows a temporal graph $\mathcal{G}$. For simplicity, we assume that the traversal time for every edge is 1 time unit. In $\mathcal{G}$, a can reach d within time interval $[2, 5]$ since there is a temporal path $P = \langle(a, b, 2, 1), (b, d, 4, 1)\rangle$, while a cannot reach d within $[1, 3]$ since there is no temporal path from a to d within $[1, 3]$. Given source vertex a, target vertex d, and a time interval $[1, 10]$, $P_1 = \langle(a, b, 2, 1), (b, d, 4, 1)\rangle$ is an earliest-arrival path with arrival time $4 + 1 = 5$, $P_2 = \langle(a, c, 4, 1), (c, d, 5, 1)\rangle$ is a fastest path with duration $(5 + 1) - 4 = 2$. Note that P_1 is not a fastest path, and P_2 is not an earliest-arrival path within $[1, 10]$.

Problem definition: Given a temporal graph $\mathcal{G}$, we propose to construct an index, such that given a source vertex u and a target vertex v, and a time interval $[t_\alpha, t_\omega]$, we can efficiently answer the following queries: (1) whether u can reach v within $[t_\alpha, t_\omega]$, (2) the earliest-arrival time going from u to v within $[t_\alpha, t_\omega]$, and (3) the minimum duration taken to go from u to v within $[t_\alpha, t_\omega]$.

Our method can also be applied to compute another type of minimum temporal path called ***latest-departure path*** [9]. However, the concept of latest-departure path is symmetric to that of earliest-arrival path. We hence omit the details.

III. Graph Transformation

In addition to indexing and querying efficiency, another key consideration in designing an indexing method for querying temporal graphs is that the index must support efficient dynamic update, since temporal edges are created and added frequently over time in most real applications. We found that the graph transformation method in [9] can be applied to design an efficient index that also allows efficient dynamic update.

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

1) Construction of vertex set: for each vertex $v \in \mathcal{V}$, create vertices in V as follows.
    a) Let $T_{in}(u, v) = \{t + \lambda : (u, v, t, \lambda) \in \Pi(u, v)\}$ for each $u \in \Gamma_{in}(v, \mathcal{G})$, and $\mathbf{T}_{in}(v) = \bigcup_{u \in \Gamma_{in}(v, \mathcal{G})} T_{in}(u, v)$, 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 $\langle v, t\rangle$ where t is a distinct arrival time in $\mathbf{T}_{in}(v)$. Denote this set of vertices as $V_{in}(v)$, i.e., $V_{in}(v) = \{\langle v, t\rangle : t \in \mathbf{T}_{in}(v)\}$. Sort vertices in $V_{in}(v)$ in descending order of their time, i.e., for any $\langle v, t_1\rangle, \langle v, t_2\rangle \in V_{in}(v)$, $\langle v, t_1\rangle$ is ordered before $\langle v, t_2\rangle$ in $V_{in}(v)$ iff $t_1 > t_2$.
    b) Let $T_{out}(v, u) = \{t : (v, u, t, \lambda) \in \Pi(v, u)\}$ for each $u \in \Gamma_{out}(v, \mathcal{G})$, and $\mathbf{T}_{out}(v) = \bigcup_{u \in \Gamma_{out}(v, \mathcal{G})} T_{out}(v, u)$.
    Create $|\mathbf{T}_{out}(v)|$ copies of v, each labeled with $\langle v, t\rangle$ where t is a distinct starting time in $\mathbf{T}_{out}(v)$. Denote this set of vertices as $V_{out}(v)$, i.e., $V_{out}(v) = \{\langle v, t\rangle : t \in \mathbf{T}_{out}(v)\}$. Sort vertices in $V_{out}(v)$ in descending order of their time.
2) Construction of edge set: create edges in E as follows.
    a) Let $V_{in}(v) = \{\langle v, t_1\rangle, \langle v, t_2\rangle, \ldots, \langle v, t_h\rangle\}$, where $t_i > t_{i+1}$ for $1 \leq i < h$. Create a directed edge from each $\langle v, t_{i+1}\rangle$ to $\langle v, t_i\rangle$, for $1 \leq i < h$. No edge is created if $h \leq 1$. Create edges for $V_{out}(v)$ in the same way.
    b) For each vertex $\langle v, t_{in}\rangle$ according to its order in $V_{in}(v)$, create a directed edge from $\langle v, t_{in}\rangle$ to vertex $\langle v, t_{out}\rangle \in V_{out}(v)$, where $t_{out} = \min\{t : \langle v, t\rangle \in V_{out}(v), t \geq t_{in}\}$ and no edge from another vertex $\langle v, t'_{in}\rangle \in V_{in}(v)$ to $\langle v, t_{out}\rangle$ has been created.
    c) For each temporal edge $e = (u, v, t, \lambda) \in \mathcal{E}$, create a directed edge from the vertex $\langle u, t\rangle \in V_{out}(u)$ to the vertex $\langle v, t + \lambda\rangle \in V_{in}(v)$.

The following example illustrates graph transformation.

Example 2: Figure 1(b) shows the transformed graph G of the temporal graph $\mathcal{G}$ in Figure 1(a), where $\lambda = 1$ for all edges. Although there is a cycle $\langle(a, c, 4, 1), (c, a, 6, 1)\rangle$ in $\mathcal{G}$, G is a DAG.

IV. Top-k Chain Labeling

In this section, we present the *top-k chain labeling scheme*, which we name as **TopChain**. We focus on our discussion on reachability queries, and we discuss how TopChain is applied to answer time-based queries in Section V-B.

Most labeling schemes first transform the input directed graph into a *directed acyclic graph* (*DAG*) by collapsing each *strongly connected component* (*SCC*) into a super vertex, and then construct labels on the much smaller DAG. Here, we prove that a transformed graph is a DAG.

Given a transformed graph $G = (V, E)$, for each vertex $\langle w, t\rangle \in V$, let $v = \langle w, t\rangle$. For simplicity, we often simply use v instead of $\langle w, t\rangle$ in our discussion. We use $\langle w, t\rangle$ only when we need to refer to v's *time stamp* t.

Due to space limit, all the proofs for the lemmas and theorems are given in the full version of this paper [24].

Lemma 1: Let $G = (V, E)$ be the transformed graph of a temporal graph $\mathcal{G}$, if the traversal time of each edge in $\mathcal{G}$ is non-zero, then G is a DAG.

The size of G is $O(|\mathcal{V}| + |\mathcal{E}|)$, and is even considerably larger than $\mathcal{G}$ (see Table II). Thus, the DAGs we handle in this paper are significantly larger than those used to test existing indexes. In the following, we propose a more efficient method that can handle much larger DAGs.

A. Label Construction

Given a DAG (not necessarily a transformed graph), $G = (V, E)$, we use $u \to v$ to denote that u can reach v in G, and $u \nrightarrow v$ if u cannot reach v. We assume $v \to v$ for any $v \in V$. We define the set of **out-neighbors** of a vertex u in G as $\Gamma_{out}(u, G) = \{v : (u, v) \in E\}$, and the **out-degree** of u in G as $d_{out}(u, G) = |\Gamma_{out}(u, G)|$. Similarly, we define $\Gamma_{in}(u, G) = \{v : (v, u) \in E\}$ and $d_{in}(u, G) = |\Gamma_{in}(u, G)|$.

TopChain constructs two sets of labels for each $v \in V$, denoted by $L_{in}(v)$ and $L_{out}(v)$, called **in-labels** and **out-labels** of v. TopChain takes an input parameter k, which is used to control the label set size, label construction time and query processing time.

The main idea is to compute the top k labels based on a ranking defined on a chain cover of the input DAG $G = (V, E)$, for $L_{in}(v)$ and $L_{out}(v)$ of each $v \in V$. Here, a *chain* C is an ordered sequence of h vertices $C = \langle v_1, v_2, \ldots, v_h\rangle$, such that $v_i \to v_{i+1}$ for $1 \le i < h$. A *chain cover* of G is a disjoint partition $\{C_1, \ldots, C_l\}$ of V, where C_i is a chain for $1 \le i \le l$.

1) Definition of Labels: Given a chain cover $\mathbb{C} = \{C_1, \ldots, C_l\}$ of G, and a rank $rank(C)$ for each chain $C \in \mathbb{C}$, the labels of each $v \in V$ are defined as follows.

Define a **chain code** for v as $code(v) = (v.x, v.y)$, where $v.x = rank(C)$ and v in C, and $v.y$ is the position of v in C. Let $RC(v) \subseteq \mathbb{C}$ be the set of chains that v can reach; formally, for each $C \in RC(v)$, there exists a vertex u in C such that $v \to u$. For each $C \in RC(v)$, let $first_v(C)$ be the first vertex in C that v can reach, i.e., if $C = \langle v_1, v_2, \ldots, v_h\rangle$ and $first_v(C) = v_i$, then $v \to v_j$ for $i \le j \le h$ and $v \nrightarrow v_{j'}$ for $1 \le j' < i$. Note that $first_v(C) = v$, where v in C, since we assume $v \to v$.

Define $RF(v)$ as the set of first reachable vertices in the set of reachable chains from v, i.e., $RF(v) = \{first_v(C) : C \in RC(v)\}$. Define $RFcode(v) = \{code(u) : u \in RF(v)\}$. Sort the chain codes in $RFcode(v)$ in ascending order of $u.x$ for each $(u.x, u.y) \in RFcode(v)$, and let $top_k(RFcode(v))$ be the first k chain codes in $RFcode(v)$ and $top_k(RFcode(v)) = RFcode(v)$ if $|RFcode(v)| \le k$. Then, $L_{out}(v) = top_k(RFcode(v))$.

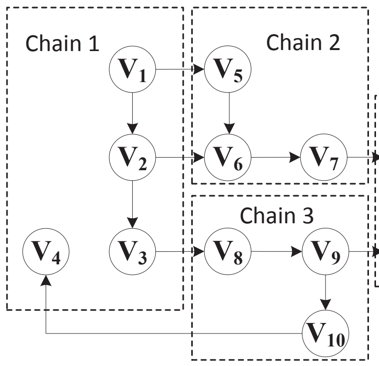


Figure 2. Chain cover of G in Figure 1(b)

Similarly, let $RC^{-1}(v) \subseteq \mathbb{C}$ be the set of chains that can reach v, $last_v(C)$ be the last vertex in C that can reach v. Define $RL(v) = \{last_v(C) : C \in RC^{-1}(v)\}$. Define $RLcode(v) = \{code(u) : u \in RL(v)\}$. Sort the chain codes in $RLcode(v)$ in ascending order of $u.x$ for each $(u.x, u.y) \in RLcode(v)$, and let $top_k(RLcode(v))$ be the first k chain codes in $RLcode(v)$ and $top_k(RLcode(v)) = RLcode(v)$ if $|RLcode(v)| \le k$. Then, $L_{in}(v) = top_k(RLcode(v))$.

Example 3: Figure 2 shows a DAG G, which is in fact G in Figure 1(b) by relabeling the vertices, and a chain cover of G, $\mathbb{C} = \{C_1, C_2, C_3, C_4\}$, where $C_1 = \langle v_1, v_2, v_3, v_4\rangle$, $C_2 = \langle v_5, v_6, v_7\rangle$, $C_3 = \langle v_8, v_9, v_{10}\rangle$, $C_4 = \langle v_{11}, v_{12}\rangle$. The chain code of v_3 is $code(v_3) = (1, 3)$ since v_3 is at the 3-rd position in C_1. And $RC(v_3) = \{C_1, C_3, C_4\}$, since v_3 can reach C_1, C_3 and C_4. $RF(v_3) = \{v_3, v_8, v_{12}\}$, $RFcode(v_3) = \{(1, 3), (3, 1), (4, 2)\}$, since $code(v_3) = (1, 3)$, $code(v_8) = (3, 1)$, $code(v_{12}) = (4, 2)$. Let $k = 2$, then $L_{out}(v_3) = top_2(RFcode(v_3))=\{(1, 3), (3, 1)\}$. Similarly, $RC^{-1}(v_3)=\{C_1\}$, $RLcode(v_3)=\{(1, 3)\}$, $L_{in}(v_3)=top_2(RLcode(v_3))=\{(1, 3)\}$.

2) Algorithm for Labeling: We now present our method to compute the above-defined labels, as outlined in Algorithm 1. We discuss how to compute a chain cover of G and chain ranking in Section IV-B. Given a chain cover $\mathbb{C}$ and a rank $rank(C)$ for each chain $C \in \mathbb{C}$, Lines 1-4 first assign the chain code of each vertex v to both $L_{out}(v)$ and $L_{in}(v)$.

Assume that each vertex in V is assigned a topological order, which can be obtained by topological sort on G. Then, Lines 5-8 compute $L_{out}(v)$ for each $v \in V$ in reverse topological order. For each v, the algorithm computes the top k labels with the smallest chain rank from the set of all out-labels of v's out-neighbors and the out-label of v. This can be done by scanning $L_{out}(u)$ for each $u \in \Gamma_{out}(v, G)$ as the merge phase in merge-sort until the top k labels are obtained, since the labels in each $L_{out}(u)$ are ordered according to chain rank. In addition, Line 7 ensures that at most one vertex from each chain can have its chain code included as one of the top k labels. Then, the top k labels are assigned to $L_{out}(v)$. Similarly, $L_{in}(v)$ for each $v \in V$ is computed in Lines 9-12.

We now discuss the correctness and complexity of the algorithm (see proofs in [24]).

Lemma 2: Algorithm 1 correctly computes $L_{out}(v)$ and $L_{in}(v)$ for every vertex $v \in V$.

Algorithm 1: TopChain: Label Construction

```
Input  : A DAG $G = (V, E)$, a chain cover
         $\mathbb{C} = \{C_1, \ldots, C_l\}$ of $G$, where $C_i$ is ranked before
         $C_j$ for $1 \le i < j \le l$, and an integer $k$
Output : $L_{out}(v)$ and $L_{in}(v)$ for every vertex $v \in V$
1  foreach vertex, $v \in V$, do
2      Let $C$ be the chain that contains $v$;
3      Assign a chain code $(v.x, v.y)$ to $v$, where
       $v.x = rank(C)$ and $v.y$ is the position of $v$ in $C$;
4      $L_{in}(v) \leftarrow \{(v.x, v.y)\}$, $L_{out}(v) \leftarrow \{(v.x, v.y)\}$;
5  foreach vertex, $v \in V$, in reverse topological order do
6      Let $L = L_{out}(v) \cup \bigcup_{u \in \Gamma_{out}(v,G)} L_{out}(u)$;
7      Let $L_k$ be the top $k$ labels with the smallest chain rank
       from $L$ such that: if there exist two labels $(u.x, u.y)$ and
       $(w.x, w.y)$ in $L$ such that $u.x = w.x$ and $u.y < w.y$, i.e.,
       $u$ and $w$ are in the same chain, then $\underline{(w.x, w.y)} \notin L_k$;
8      $L_{out}(v) \leftarrow L_k$;
9  foreach vertex, $v \in V$, in topological order do
10     Let $L = L_{in}(v) \cup \bigcup_{u \in \Gamma_{in}(v,G)} L_{in}(u)$;
11     Let $L_k$ be the top $k$ labels with the smallest chain rank
       from $L$ such that: if there exist two labels $(u.x, u.y)$ and
       $(w.x, w.y)$ in $L$ such that $u.x = w.x$ and $u.y < w.y$, i.e.,
       $u$ and $w$ are in the same chain, then $\underline{(u.x, u.y)} \notin L_k$;
12     $L_{in}(v) \leftarrow L_k$;
```

Theorem 1: Given a DAG $G = (V, E)$ and a chain cover of G, Algorithm 1 correctly computes $L_{out}(v)$ and $L_{in}(v)$ for all $v \in V$ in $O(k(|V| + |E|))$ time, and the total label size is given by $\sum_{v \in V}(|L_{out}(v)| + |L_{in}(v)|) = O(k|V|)$.

We show how Algorithm 1 computes the labels as follows.

Example 4: Given the chain cover $\mathbb{C} = \{C_1, C_2, C_3, C_4\}$ of G in Figure 2, and $k = 2$, Algorithm 1 first initializes $L_{in}(v)$ and $L_{out}(v)$ to contain v itself. One topological order is as follows: $\langle v_1, v_2, v_3, v_5, v_6, v_7, v_8, v_9, v_{10}, v_4, v_{11}, v_{12} \rangle$. Then, in reverse topological order, we compute $L_{out}(v)$ for every v. We show how $L_{out}(v_3)$ is computed. First, we compute $L_{out}(v_{12}) = code(v_{12}) = \{(4, 2)\}$ and $L_{out}(v_4) = code(v_4) = \{(1, 4)\}$. Next, we compute $L_{out}(v_{10}) = code(v_{10}) \cup L_{out}(v_4) = \{(1, 4), (3, 3)\}$, $L_{out}(v_9) = top_2(code(v_9) \cup L_{out}(v_{10}) \cup L_{out}(v_{12})) = top_2\{(3, 2), (1, 4), (3, 3), (4, 2)\} = \{(1, 4), (3, 2)\}$, $L_{out}(v_8) = top_2(code(v_8) \cup L_{out}(v_9)) = \{(1, 4), (3, 1)\}$, $L_{out}(v_3) = top_2(code(v_3) \cup L_{out}(v_8)) = \{(1, 3), (3, 1)\}$. Similarly, we compute $L_{in}(v_3) = \{(1, 3)\}$.

B. Chain Cover and Chain Ranking

One input to Algorithm 1 is a chain cover of G. An optimal chain cover, which is one that consists of the minimum number of chains, can be computed by a min-flow based method in $O(|V|^3)$ time [23]. And the time is reduced to $O(|V|^2 + |V|l\sqrt{l})$ based on bipartite matching [25], where l is the number of chains. Both of these two methods are too expensive for processing large graphs.

Since our labeling scheme presented in Section IV-A is not limited to a transformed graph but any general DAG, we apply a greedy algorithm [22] to compute a chain cover if the application is to answer reachability queries in a non-temporal graph. The greedy algorithm grows a chain by recursively adding the smallest-ranked out-neighbor of the last vertex in the chain, where the ranking is defined based on a topological ordering of the vertices. The algorithm uses $O(|V| \log d_{max} + |E|)$ time, where d_{max} is the maximum degree of a vertex in the DAG.

For processing a temporal graph $\mathcal{G}$, we adopt a simple and efficient method based on the property of $\mathcal{G}$ as follows. In the transformed graph $G = (V, E)$ of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, each set of vertices $V_{in}(v)$ or $V_{out}(v)$ naturally appears as a chain. Thus, we can obtain a natural chain cover of G, i.e., $\mathbb{C} = \{V_{in}(v) : v \in \mathcal{V}\} \cup \{V_{out}(v) : v \in \mathcal{V}\}$. In fact, we can reduce the number of chains in $\mathbb{C}$ by half as follows. We merge $V_{in}(v)$ and $V_{out}(v)$ into one single chain in ascending order of the time stamp of the vertices. If there exist $\langle v, t_{in} \rangle \in V_{in}(v)$ and $\langle v, t_{out} \rangle \in V_{out}(v)$ such that $t_{in} = t_{out}$, we order $\langle v, t_{in} \rangle$ before $\langle v, t_{out} \rangle$.

Merging $V_{in}(v)$ and $V_{out}(v)$ into a single chain C can approximately reduce query response time by half, since logically $V_{in}(v)$ and $V_{out}(v)$ belong to a single vertex v in the original temporal graph. For example, the chain cover in Figure 2 is constructed in this way.

However, theoretically there is one small problem, which can be easily fixed, though we need to show a rigorous proof to show the correctness of our indexing method. We first present the problem as follows. Let $C = \langle u_1, u_2, \ldots, u_h \rangle$. The definition of chain requires that $u_i \rightarrow u_j$ for $1 \le i < j \le h$. However, in a temporal graph $\mathcal{G}$, it is possible that a vertex cannot reach itself, i.e., u_i may not reach u_j in the transformed graph G, for some $u_i \in V_{out}(v)$ and $u_j \in V_{in}(v)$. Essentially, merging $V_{in}(v)$ and $V_{out}(v)$ into a single chain C creates a new graph G_{new}, by adding an edge from $\langle v, t_{out} \rangle \in V_{out}(v)$ to $\langle v, t_{in} \rangle \in V_{in}(v)$ to $\mathcal{G}$ for each $v \in \mathcal{V}$ if $t_{out} < t_{in}$.

However, G_{new} only exists conceptually used to define the chain cover, and we never really use G_{new} in our algorithm for label construction. Note that we do not compute the chain cover from G_{new}, but simply form a chain C from each $V_{in}(v)$ and $V_{out}(v)$ in G, although the reachability of the vertices in C is defined based on G_{new} instead of G.

In the following theorem (see proof in [24]), we show the correctness of Algorithm 1 when the input is G and the chain cover is based on G_{new}, even though there exists false reachability information in G_{new}. In Section V-B, we will also show that the labels give correct answers to time-based queries.

Theorem 2: Let $\mathbb{C}$ be the chain cover defined based on G_{new}, where each $V_{in}(v)$ and $V_{out}(v)$ in G_{new} are merged into a single chain $C \in \mathbb{C}$. Given G and $\mathbb{C}$ as input, Algorithm 1 constructs the same labels as the labels defined based on G_{new} and $\mathbb{C}$ (i.e., constructed by Algorithm 1 with G_{new} and $\mathbb{C}$ as input).

This chain cover can be naturally computed at no extra cost during the process of graph transformation, and thus the whole process takes only linear time. In addition, for the chain code $(v.x, v.y)$ of each vertex $\langle v, t \rangle \in V$, instead of assigning $v.y$ as the position of v in its chain, we can directly use the time stamp of v, i.e., $v.y = t$. This new assignment of $v.y$ is in fact significant when update maintenance of the labels is considered. There can be frequent edge insertions in a temporal graph over time and in this case the labels need to be updated as well. If $v.y$ is assigned as the position of v in its chain, then updating the labels is more difficult since inserting a vertex in a chain affects the position of all following vertices in the chain, which can in turn affect the labels of a large number of vertices in the graph. On the other hand, if $v.y = t$, then we can simply insert the vertex in the chain and $u.y$ for any vertex u following v in the chain needs not be updated. Dynamic update of labels will be discussed in Section IV-C.

Each chain in $\mathbb{C}$ is assigned a rank for labeling. There are many different strategies to rank the chains. We only discuss strategies with a low computation cost, that is, they are practical for large graphs. Two such strategies are discussed as follows.

- Random ranking: We rank the chains randomly. We use this method as a baseline.
- Ranking by degree: Let $\Phi(C)$ denote the sum of out-edges and in-edges of all the vertices in a chain C. We rank the chains in descending order of their Φ value, where the top-ranked chain has a rank of 1. The rationale for this ranking is that the higher the value of $\Phi(C)$, the higher is the probability that C can reach and are reachable from a larger set of vertices in G. Thus, assigning a top rank to C enables more vertices to contain $rank(C)$ in their labels, thus allowing a more efficient query processing. We use this method in our TopChain method. We apply radix sort to sort the chains in order to assign ranks, and hence maintain the linear index construction time complexity.

C. Dynamic Update of Labels

New edges and vertices may be added to a temporal graph $\mathcal{G}$ over time. Since adding an isolated vertex is trivial, we only discuss the addition of a new edge $e = (a, b, t, \lambda)$. We need to update G by inserting $u = \langle a, t\rangle$ into $V_{out}(a)$ and $v = \langle b, t+\lambda\rangle$ into $V_{in}(b)$, and adding an edge from u to v. Consequently, the labels should be updated as follows.

First, we need to insert u into the chain C that is formed from $V_{out}(a)$ and $V_{in}(a)$. If C does not exist, we create u as a new chain, assign it a rank l that is larger than that of existing chains, and initialize $L_{out}(u)$=$L_{in}(u)$=(l, t). If C exists, we insert u into the right position in C according to t, and initialize $L_{out}(u)$=$L_{in}(u)$= $(rank(C), t)$. Let u_1 be the vertex ordered before u in C and u_2 be the vertex ordered after u in C. We compute $L_{in}(u)$ as the top k labels from $L_{in}(u) \cup L_{in}(u_1)$, and $L_{out}(u)$ as the top k labels from $L_{out}(u) \cup L_{out}(u_2)$. Similarly, we compute $L_{out}(v)$ and $L_{in}(v)$.

Second, after inserting a new edge (u, v) into G, we update the labels as follows. We perform a reverse BFS starting from vertex u in G to update the out-labels of vertices that are visited, since only these vertices may change their out-labels. For any vertex w visited, let w' be the parent of w in the reverse BFS, we update $L_{out}(w)$ as the top k labels from $L_{out}(w) \cup L_{out}(w')$. If $L_{out}(w)$ remains unchanged, then we do not continue the search from w. Similarly, we conduct a BFS starting from vertex v to update the in-labels of the visited vertices.

The algorithm completes label updating in $O(k(|V|+|E|))$ time, which is the optimal worst case time. In practice, the update is very efficient, as we demonstrate by experiments.

Algorithm 2: ReachQ$(G, \mathbb{L}, (u, v))$: Reachability Querying

```
Input  : A DAG $G = (V, E)$,
         $\mathbb{L} = \{(L_{out}(v), L_{in}(v)) : v \in V\}p$, and a pair of
         query vertices $(u, v)$
Output : The answer whether $u$ can reach $v$
1  if $u.x = v.x$ then
2      if $u.y \leq v.y$ then
3          return true;
4      return false;
5  if $L_{out}(u) \gg L_{out}(v)$ or $L_{in}(v) \gg L_{in}(u)$ then
6      return false;
7  if $L_{out}(u) \oplus L_{in}(v) = 1$ then
8      return true;
9  foreach $w \in \Gamma_{out}(u, G)$ do
10     if $w$ has been not visited then
11         if ReachQ$(G, \mathbb{L}, (w, v))$ returns true then
12             return true;
13 return false;
```

V. Query Processing by TopChain

We now discuss how we use the labels constructed in Section IV to answer reachability queries and minimum temporal path queries.

A. Reachability Queries

We process a reachability query whether $u \to v$ as shown in Algorithm 2. We first define a few operators used in the algorithm.

We first define the operator $\oplus$:

$$L_{out}(u) \oplus L_{in}(v) = \begin{cases} 1 & \text{if } \exists(r.x, r.y) \in L_{out}(u), \\ & \quad (s.x, s.y) \in L_{in}(v), \text{s.t.} \\ & \quad r.x = s.x \text{ and } r.y \leq s.y \\ 0 & \text{otherwise} \end{cases}$$

Intuitively, $L_{out}(u) \oplus L_{in}(v)$ tests whether there exist two vertices r and s, where $(r.x, r.y) \in L_{out}(u)$ and $(s.x, s.y) \in L_{in}(v)$, such that r and s are in the same chain, and either $r = s$ or r is ordered before s in the chain. The following lemma (see proof in [24]) shows how the operator can be used in reachability query processing.

Lemma 3: If $L_{out}(u) \oplus L_{in}(v) = 1$, then $u \to v$.

The following example illustrates how the operator $\oplus$ works.

Example 5: Consider G in Figure 2 and $k = 2$, Algorithm 1 computes $L_{out}(v_3) = \{(1, 3), (3, 1)\}$ and $L_{in}(v_{12}) = \{(1, 3), (3, 2)\}$. Consider a reachability query that asks whether $v_3 \to v_{12}$. Since $L_{out}(v_3) \oplus L_{in}(v_{12}) = 1$ as $\exists(1, 3) \in L_{out}(v_3)$ and $(1, 3) \in L_{in}(v_{12})$, we conclude $v_3 \to v_{12}$.

Next, we define another operator $\gg$ as follows. We say $L_{out}(u) \gg L_{out}(v)$ if one of the following two cases is true:

- Case (1): $\exists(r.x, r.y) \in L_{out}(v)$, $\nexists(w.x, w.y) \in L_{out}(u)$ such that $w.x = r.x$, and $\exists(s.x, s.y) \in L_{out}(u)$ such that $s.x > r.x$;
- Case (2): $\exists(r.x, r.y) \in L_{out}(v)$ and $(w.x, w.y) \in L_{out}(u)$ such that $w.x = r.x$ and $w.y > r.y$.

Intuitively, $L_{out}(u) \gg L_{out}(v)$ tests whether (1) there exists a vertex r in a chain C_1 in $L_{out}(v)$, not exists any vertex in $L_{out}(u)$ in the same chain C_1, and exists a vertex s in a chain C_2 in $L_{out}(u)$, such that the chain rank of C_2 is larger than that of C_1, which indicates that v can reach at least one vertex in C_1, while u cannot reach any vertex in C_1; or (2) there exists a vertex r in a chain C in $L_{out}(v)$, and a vertex w in same chain C in $L_{out}(u)$, such that r is ordered before w in C, which indicates that the first vertex in C that v can reach is r, the first vertex in C that u can reach is w, and $r \to w$.

Similarly, we say $L_{in}(v) \gg L_{in}(u)$ if one of the following two cases is true:

- Case (1): $\exists(r.x, r.y) \in L_{in}(u)$, $\nexists(w.x, w.y) \in L_{in}(v)$ such that $w.x{=}r.x$, and $\exists(s.x, s.y) \in L_{in}(v)$ such that $s.x{>}r.x$;
- Case (2): $\exists(r.x, r.y) \in L_{in}(u)$ and $(w.x, w.y) \in L_{in}(v)$ such that $w.x = r.x$ and $w.y < r.y$.

The following lemma (see proof in [24]) shows how the operator $\gg$ can be used in reachability query processing.

Lemma 4: If $L_{out}(u) \gg L_{out}(v)$ or $L_{in}(v) \gg L_{in}(u)$, then $u \nrightarrow v$.

We illustrate how the operator $\gg$ works as follows.

Example 6: Consider G in Figure 2 and $k = 2$, Algorithm 1 computes $L_{out}(v_2) = \{(1,2),(2,2)\}$, $L_{out}(v_3) = \{(1,3),(3,1)\}$ and $L_{out}(v_5) = \{(2,1),(4,1)\}$. Since $\exists(3,1) \in L_{out}(v_3)$, $\nexists(w.x, w.y) \in L_{out}(v_5)$ such that $w.x = 3$, and $\exists(2,1) \in L_{out}(v_5)$ such that it satisfies Case (1) for $L_{out}(v_3) \gg L_{out}(v_5)$, we have $v_3 \nrightarrow v_5$. Since $\exists(2,2) \in L_{out}(v_2)$ and $(2,1) \in L_{out}(v_5)$ such that it satisfies Case (2) for $L_{out}(v_2) \gg L_{out}(v_5)$, we conclude that $v_2 \nrightarrow v_5$.

Algorithm 2 first uses the chain code of the query vertices, u and v, to check whether u and v are in the same chain. If u and v are in the same chain, then by the definition of chain and the fact that G is a DAG, we have $u \to v$ if $u.y \leq v.y$ and $u \nrightarrow v$ if $u.y > v.y$. Then, the algorithm applies the operators $\oplus$ and $\gg$ on the labels of u and v to further examine whether the query answer can be determined. If not, then the algorithm processes the query by testing if any of the descendants of u can reach v, by visiting the descendants in a depth-first manner. If a descendant of u can reach v, then it implies $u \to v$. Otherwise, the algorithm finally returns $u \nrightarrow v$. Note that we can prune some descendants of u in the search, which will be discussed in Section VI.

Base on Lemmas 3 and 4, we have the following theorem.

Theorem 3: Algorithm 2 correctly answers a reachability query whether $u \to v$.

B. Time-Based Queries

We now discuss how to answer temporal reachability queries and minimum temporal path queries.

Temporal reachability queries. To answer a reachability query whether a source vertex a can reach a target vertex b in a temporal graph $\mathcal{G}$ within a time interval $[t_\alpha, t_\omega]$, we process the query in the transformed graph G of $\mathcal{G}$ as follows.

We first find $\langle a, t_{out}\rangle$ in $V_{out}(a)$, where $t_{out} = \min\{t : \langle a, t\rangle \in V_{out}(a), t \geq t_\alpha\}$. Since the vertices in $V_{out}(a)$ are ordered by their time stamp, we find $\langle a, t_{out}\rangle$ by binary search. Similarly, we find $\langle b, t_{in}\rangle$ in $V_{in}(b)$, where $t_{in} = \max\{t : \langle b, t\rangle \in V_{in}(b), t \leq t_\omega\}$.

Let $u = \langle a, t_{out}\rangle$ and $v = \langle b, t_{in}\rangle$. If u or v does not exist, then the answer to the query is false. Otherwise, Algorithm 2 is called to answer whether u can reach v in G. If Algorithm 2 returns true, then a can reach b in $\mathcal{G}$ within $[t_\alpha, t_\omega]$. Otherwise, a cannot reach b within $[t_\alpha, t_\omega]$.

In addition, if Lines 9-12 of Algorithm 2 need to be executed, we can employ the time interval $[t_\alpha, t_\omega]$ for search space pruning as follows. For any descendant $w = \langle c, t\rangle$ of u visited during the search, if $t > t_\omega$, we can directly terminate the search from w.

The above procedure, however, may give an incorrect query answer in the case when $a = b$, i.e., $u.x = v.x$, u and v are in the same chain. This is because the chains of G obtained in Section IV-B may present false reachability information, i.e., u is ordered before v in a chain but u cannot reach v in G. However, this can be easily addressed as follows. In the case when $u \in V_{out}(a)$ and $v \in V_{in}(a)$, where $u.x = v.x$, we simply call Algorithm 2 to answer whether $\exists w \in W$, where $W = \{w : w \in \Gamma_{out}(u', G), u'.x = u.x, u'.y \geq u.y, w.x \neq u.x\}$ (i.e., u' is u or any vertex ordered after u in the same chain, and w is an out-neighbor of u' that is not in the same chain of u'), such that $w \to v$. We have $u \to v$ if and only if w exists.

The following theorem (see proof in [24]) proves the correctness of processing a temporal reachability query.

Theorem 4: The algorithm described above correctly answers a reachability query whether a can reach b in $\mathcal{G}$ within $[t_\alpha, t_\omega]$.

Earliest-arrival time. We compute the earliest-arrival time going from vertex a to vertex b within $[t_\alpha, t_\omega]$ as follows. We first find $\langle a, t_{out}\rangle$ in $V_{out}(a)$, where $t_{out} = \min\{t : \langle a, t\rangle \in V_{out}(a), t \geq t_\alpha\}$. Then, we compute the set of vertices, $B = \{\langle b, t\rangle : \langle b, t\rangle \in V_{in}(b), t_\alpha \leq t \leq t_\omega\}$.

Let $u = \langle a, t_{out}\rangle$. We want to find $v = \langle b, t\rangle \in B$ such that $u \to v$, where $\nexists v'{=}\langle b, t'\rangle \in B$ such that $u \to v'$ and $t'{<}t$. If such a vertex v can be found, then the earliest-arrival time going from a to b by any path in $\mathcal{G}$ within $[t_\alpha, t_\omega]$ is given by t. If v is not found, then the corresponding earliest-arrival path does not exist in $\mathcal{G}$.

As vertices in B are ordered according to their time stamp, we can employ a binary-search-like process to find v, instead of querying whether $u \to w$ for each $w \in B$. Let $B = \{w_1, \ldots, w_h\}$ and $w_i = \langle b, t_i\rangle$, where $t_i < t_{i+1}$ for $1 \leq i < h$. We start with w_h. If $u \nrightarrow w_h$, then $u \nrightarrow w_i$ for

$1 \leq i \leq h$; thus, we can conclude that the earliest-arrival path from a to b does not exist in $\mathcal{G}$ within $[t_\alpha, t_\omega]$. If $u \rightarrow w_h$, then we choose the middle vertex in B, i.e., $w_{h/2}$, and process the query whether $u \rightarrow w_{h/2}$. In this way, we stop until we find the first vertex $w_i \in B$ where $u \rightarrow w_i$, and return t_i as the query answer.

We process each query $u \rightarrow w_i$ by Algorithm 2. The correctness of the query answer follows from the fact that an earliest-arrival path from a to b is simply a path starting from a that *reaches* b at the earliest time.

Minimum duration. We compute the minimum duration taken to go from vertex a to vertex b within $[t_\alpha, t_\omega]$ as follows. We first compute $A = \{\langle a, t\rangle : \langle a, t\rangle \in V_{out}(a), t_\alpha \leq t \leq t_\omega\}$. Then, from each $u_i = \langle a, t_i\rangle \in A$, we obtain a starting time t_i, and find the earliest-arrival time going from a to b within $[t_i, t_\omega]$ by the same binary-search-like process discussed above for computing earliest-arrival time. Let t'_i be the earliest-arrival time obtained starting at time t_i. Then, the minimum duration is given by $\min\{(t'_i - t_i) : u_i = \langle a, t_i\rangle \in A\}$. The correctness of the query answer follows from the fact that a fastest path from a (starting at time t) to b is also an earliest-arrival path from a (starting at time t) to b.

VI. Improvements on Labeling

We present two improvements on our labeling scheme.

Label reduction. With a close investigation of the property of the transformed graph, we can reduce the label size by half.

Given a vertex $\langle a, t_{out}\rangle \in V_{out}(a)$, let $\langle a, t_{in}\rangle \in V_{in}(a)$ where $t_{in} = \max\{t : \langle a, t\rangle \in V_{in}(a), t \leq t_{out}\}$. Let $u = \langle a, t_{out}\rangle$ and $v = \langle a, t_{in}\rangle$. Then, we only need to keep $L_{out}(u)$ for u, and keep a pointer to $L_{in}(v)$. When $L_{in}(u)$ is needed for query processing, we simply use $L_{in}(v)$ instead.

Similarly, given a vertex $\langle a, t_{in}\rangle \in V_{in}(a)$, let $\langle a, t_{out}\rangle \in V_{out}(a)$ where $t_{out} = \min\{t : \langle a, t\rangle \in V_{out}(a), t \geq t_{in}\}$. Let $u = \langle a, t_{in}\rangle$ and $v = \langle a, t_{out}\rangle$. We only need to keep $L_{in}(u)$ for u, and keep a pointer to $L_{out}(v)$. When $L_{out}(u)$ is needed for query processing, we simply use $L_{out}(v)$ instead.

The following lemma shows the correctness of label reduction (see proof in [24]).

Lemma 5: Label reduction does not affect the correctness of processing a temporal reachability query.

Topological-sort-based labels. We can further prune unreachable vertices to reduce the querying cost by some *light-weight* labels.

We first introduce the *topological level number* [12], [14], [17], [19] for a vertex v, denoted by $\ell(v)$:

- If $\Gamma_{in}(v, G) = \emptyset$: $\ell(v) = 1$;
- Else: $\ell(v) = \max\{(\ell(u) + 1) : u \in \Gamma_{in}(v, G)\}$.

We use $\ell(v)$ in processing a reachability query as follows. If $\ell(u) \geq \ell(v)$ and $u \neq v$, then $u \nrightarrow v$. This is true because if $u \rightarrow v$, then v is a descendant of u and hence $\ell(u) < \ell(v)$. We can compute $\ell(v)$ for each $v \in V$ in linear time using a single topological sort of G.

Table II. Datasets

Dataset	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	π	$\lvert T_\mathcal{G}\rvert$	$\lvert V\rvert$	$\lvert E\rvert$
austin	2,676	320,652	659	100,928	629,664	1,253,961
berlin	12,845	2,093,977	2,221	109,500	3,175,993	6,753,520
houston	9,848	1,123,580	783	98,820	2,205,384	4,396,434
madrid	4,636	1,917,090	2,406	110,347	3,793,545	7,590,572
roma	8,779	2,290,762	2,170	109,392	4,431,239	8,881,221
toronto	10,790	3,310,871	1,664	109,660	6,415,493	12,875,896
amazon	2,146,057	5,776,660	28	3,329	9,883,393	13,166,635
arxiv	28,093	9,193,606	262	2,337	433,412	9,759,445
dblp	1,103,412	11,957,392	38	70	5,553,200	16,976,956
delicious	4,535,197	219,581,041	1,070	1,583	73,792,065	293,632,816
enron	87,273	1,134,990	1,074	213,218	1,366,786	2,504,928
flickr	2,302,925	33,140,017	1	134	12,600,099	44,358,410
wikiconf	118,100	2,917,777	562	273,909	3,191,271	6,009,300
wikipedia	1,870,709	39,953,145	1	2,198	34,814,941	77,196,220
youtube	3,223,589	12,223,774	2	203	11,497,869	21,139,520

A topological sort also gives an ordering of the vertices in V. Let $\sigma(v)$ be the position of a vertex v in a topological ordering of V, where a vertex is ordered before its out-neighbors. We can use $\sigma(v)$ in processing a reachability query as follows. If $\sigma(u) > \sigma(v)$, then $u \nrightarrow v$. Note that topological ordering of V may not be unique, and this can be employed to increase the pruning power. We compute topological sort by DFS, and generate two topological orderings of V by visiting the out-neighbors of a vertex v according to their original order in $\Gamma_{out}(v, G)$ as well as their reverse order in $\Gamma_{out}(v, G)$. Let $\sigma_1(v)$ and $\sigma_2(v)$ be the value of $\sigma(v)$ obtained from the two topological orderings of V. If either $\sigma_1(u) > \sigma_1(v)$ or $\sigma_2(u) > \sigma_2(v)$, then $u \nrightarrow v$.

VII. Performance Evaluation

We now report the performance of TopChain. We ran all the experiments on a machine with an Intel 2.0GHz CPU and 128GB RAM, running Linux.

Datasets. We use 15 real temporal graphs, 6 of them, `austin`, `berlin`, `houston`, `madrid`, `roma` and `toronto`, are from Google Transit Data Feed project (code.google.com/p/googletransitdatafeed/wiki/PublicFeeds), where each dataset represents the public transportation network of a city. The other 9 of them are from the Koblenz Large Network Collection (konect.uni-koblenz.de/), and we selected one large temporal graph from each of the following categories: `amazon-ratings` (`amazon`) from the Amazon online shopping website; `arxiv-HepPh` (`arxiv`) from the arxiv networks; `dblp-coauthor` (`dblp`) from the DBLP coauthor networks; `delicious-ut` (`delicious`) from the network of "delicious"; `enron` from the email networks; `flickr-growth` (`flickr`) from the social network of Flickr; `wikiconflict` (`wikiconf`) indicating the conflicts between users of Wikipedia; `wikipedia-growth` (`wikipedia`) from the English Wikipedia hyperlink network; `youtube` from the social media networks of YouTube.

Table II gives some statistics of the datasets. We show the number of vertices and edges in each temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and the transformed graph $G = (V, E)$ of $\mathcal{G}$. The value of π varies significantly for different graphs, indicating the different levels of temporal activity between two vertices in each $\mathcal{G}$. We also show the number of atomic time intervals in each $\mathcal{G}$, denoted by $|T_\mathcal{G}|$. If we break $\mathcal{G}$ into snapshots by atomic time intervals, the `wikiconf` graph consists of as many as 273,909 snapshots.

Table III. INDEX SIZE (IN MB)

Dataset	TopChain	IP+	Ferrari	GRAIL++	PWAH8	TOL	TTL
austin	31	**29**	34	38	296	497	110
berlin	157	**145**	173	354	3793	1443	439
houston	109	**101**	120	135	1993	2129	503
madrid	188	**174**	206	232	8997	6907	870
roma	219	**203**	241	270	11097	6075	1062
toronto	318	**294**	349	392	11646	5347	905
amazon	**373**	395	482	603	31886	2532	-
arxiv	21	**19**	24	26	53	6921	306
dblp	235	**216**	256	339	47390	-	-
delicious	3410	**3293**	4010	4504	-	-	-
enron	**60**	61	72	83	321	312	94
flickr	524	**491**	558	769	-	-	-
wikiconf	**124**	142	176	195	1107	1105	214
wikipedia	1631	**1542**	1948	2125	-	-	-
youtube	464	**441**	496	702	-	-	-

Table IV. INDEXING TIME (IN SECONDS)

Dataset	TopChain	IP+	Ferrari	GRAIL++	PWAH8	TOL	TTL
austin	**0.98**	1.14	2.88	2.71	38.78	79.39	50.57
berlin	**5.34**	6.06	13.59	13.99	493.97	350.39	566.33
houston	**3.91**	4.05	9.58	10.72	247.98	368.61	306.20
madrid	**6.14**	7.10	17.56	18.07	1158.39	3078.81	1702.01
roma	**7.40**	8.51	19.17	21.06	1376.84	1574.39	2062.25
toronto	**11.63**	13.11	29.21	30.72	1405.75	1173.33	1061.96
amazon	**28.03**	26.54	43.35	72.02	2495.06	817.75	-
arxiv	**2.37**	4.78	4.80	8.08	27.73	6730.17	2761.00
dblp	17.82	**17.31**	31.40	45.91	7260.61	-	-
delicious	**384.48**	489.63	848.24	778.89	-	-	-
enron	2.37	**2.26**	4.57	6.07	33.44	64.79	603.73
flickr	55.69	**53.82**	83.57	139.10	-	-	-
wikiconf	**5.50**	6.11	11.95	15.65	111.96	246.12	2530.21
wikipedia	151.29	**142.57**	259.06	298.07	-	-	-
youtube	**30.82**	33.43	54.04	78.32	-	-	-

Table V. TOTAL QUERYING TIME (IN MILLISECONDS)

Dataset	TopChain	TC1	TC2	IP+	Ferrari	GRAIL++	PWAH8	TOL
austin	1.07	2.23	3.84	465.04	282.21	437.00	**1.04**	4.58
berlin	**1.23**	4.55	5.31	10867.10	5520.82	4063.79	1.64	2.74
houston	**0.54**	0.56	0.91	3325.21	2837.12	1619.67	1.20	3.53
madrid	**0.52**	0.53	1.46	3198.42	2974.85	1971.84	1.15	3.17
roma	**0.69**	0.85	1.21	7806.06	4326.89	4066.42	1.52	3.32
toronto	4.81	7.87	11.55	13899.10	7397.83	4631.84	**1.53**	3.76
amazon	7.38	34.36	29.24	28.29	65.13	44.29	39.08	**5.22**
arxiv	**3.33**	14.26	46.83	425.46	65.65	1154.70	23.77	4.39
dblp	**88.12**	981.26	1777.74	4353.38	3335.07	2644.30	161.03	-
delicious	**27.05**	1246.72	2648.84	60544.40	39117.00	4002.09	-	-
enron	**2.36**	10.84	47.88	147.39	14.79	113.46	24.13	2.81
flickr	**21.46**	91.52	254.40	4073.06	2160.87	1495.67	-	-
wikiconf	7.23	24.02	176.65	701.14	51.65	406.60	38.26	**3.16**
wikipedia	**288.50**	4074.23	15235.40	44810.10	8891.74	15758.72	-	-
youtube	**23.29**	528.81	165.51	974.43	1099.03	324.17	-	-

Table VI. TOTAL QUERYING TIME OF TOPCHAIN, TTL AND 1-PASS (IN SECONDS)

	Earliest-arrival			Fastest		
	TopChain	TTL	1-pass	TopChain	TTL	1-pass
austin	**0.006**	0.016	0.888	**0.023**	0.035	6.492
berlin	**0.009**	0.021	5.192	**0.012**	0.032	9.057
houston	**0.015**	0.025	2.542	**0.015**	0.046	12.782
madrid	**0.003**	0.059	4.551	**0.019**	0.136	17.029
roma	**0.047**	0.060	5.679	**0.036**	0.117	18.005
toronto	0.058	**0.036**	8.472	0.406	**0.061**	27.576
amazon	**0.011**	-	88.440	**0.048**	-	206.499
arxiv	**0.006**	0.086	16.502	**0.027**	0.086	276.874
dblp	**1.095**	-	39.427	**1.673**	-	310.934
delicious	**0.010**	-	611.542	**0.153**	-	5471.470
enron	**0.001**	0.004	3.325	**0.002**	**0.002**	14.777
flickr	**0.109**	-	125.018	**0.469**	-	1007.670
wikiconf	0.018	**0.012**	8.201	0.066	**0.009**	43.992
wikipedia	**0.638**	-	222.182	**1.507**	-	3766.520
youtube	**0.076**	-	60.800	**0.127**	-	233.982

A. *Performance on Reachability Queries*

Existing reachability indexes can be categorized into three groups: (1) *Transitive Closure*, (2) *2-Hop Labels*, and (3) *Label+Search*. We compare with the state-of-the-art indexes in each category: **PWAH8** [16] in (1); **TOL** [20] in (2); and **GRAIL++** [19], **Ferrari** [14] and **IP+** [17] in (3). We obtained the source codes from the authors. All the source codes are in C++, and we compiled them and TopChain using the same g++ compiler and optimization option.

We report the index size, indexing time, and querying time in Table III, Table IV, and Table V, respectively (note that TTL is to be discussed in Section VII-B; TC1 and TC2 are variants of TopChain to be discussed in Section VII-C). The best results are highlighted in **bold**. The sign "-" in the tables indicates that PWAH8 or TOL or TTL cannot be constructed within time $\max(x, y)$, where x is 100 times of the indexing time of TopChain and y is 10,000 seconds. For example, PWAH8, TOL and TTL cannot be constructed in $x = 38,448$ seconds for delicious.

We set k to 5 for all the *Label+Search* indexes, i.e., TopChain, IP+, Ferrari and GRAIL++. The effect of k will be studied in Section VII-C, in general, query performance improves if we use a larger k, but a larger k also leads to a larger index and longer indexing time. Since k sets the number of labels for each vertex, with the same k value, the four *Label+Search* indexes have comparable sizes, as shown in Table III. In comparison, the index sizes of PWAH8, TOL and TTL are much larger. This is because that the index size of TopChain, IP+, Ferrari and GRAIL++ are linear to the graph size, while PWAH8, TOL and TTL may take quadratic space of the graph size.

For indexing efficiency, Table IV shows that TopChain is the fastest in 11 out of 15 datasets, while for the other 4 datasets, the indexing time of TopChain is close to the best one. Compared with PWAH8 and TOL, TopChain is clearly much more scalable. PWAH8 and TOL have much worse performance than TopChain in terms of both index size and indexing time.

For query processing, we randomly generated 1000 queries, and set $[t_\alpha, t_\omega]$ to be $[0, \infty]$ for all queries so that query processing accesses the whole transformed graph. For all the indexes tested, we applied the same procedure of processing temporal reachability queries described in Section V-B.

Table V reports the total querying time by using each index. TopChain is the fastest in 11 out of 15 datasets. Among the four *Label+Search* indexes, TopChain is the fastest in all cases and is from a few times to over two orders of magnitude faster than IP+, Ferrari and GRAIL++. It is particularly important for handling the larger datasets such as delicious and wikipedia, while other methods have long querying time, TopChain remains to be very efficient. This demonstrates the effectiveness and better scalability of TopChain's labeling scheme for querying reachability in temporal graphs.

B. *Performance on Time-based Path Queries*

We compare TopChain with **1-pass**[9], and the state-of-the-art indexing method, **TTL**[11], for computing earliest-arrival time and the duration of a fastest path from a vertex u to another vertex v within time interval $[t_\alpha, t_\omega]$.

Table VI reports the total querying time of 1000 queries. TopChain is orders of magnitude faster than 1-pass. TTL is only able to handle 9 small datasets. The indexing time of TTL is orders of magnitude longer than that of TopChain, as reported in Table IV; while TTL also has a larger index size than TopChain, as shown in Table III. Given such significantly

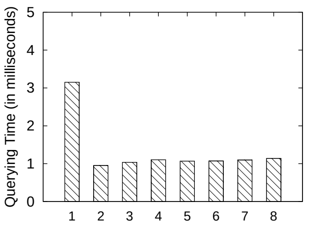


Figure 3. Querying time of `austin`

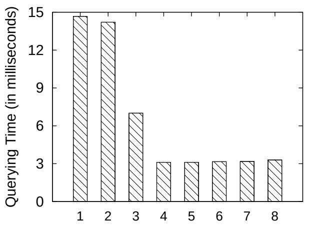


Figure 4. Querying time of `arxiv`

Table VII. Total querying time for varying intervals (in milliseconds)

Dataset	I_1	I_2	I_3	I_4
`austin`	1.067	0.310	0.020	0.015
`berlin`	1.230	0.551	0.430	0.044
`houston`	0.544	0.051	0.044	0.026
`madrid`	0.516	0.333	0.223	0.060
`roma`	0.693	0.367	0.052	0.032
`toronto`	4.812	1.091	0.158	0.076
`amazon`	7.377	0.084	0.039	0.037
`arxiv`	3.332	0.024	0.011	0.009
`dblp`	645.109	0.085	0.037	0.035
`delicious`	27.048	0.339	0.073	0.067
`enron`	2.356	0.539	0.130	0.026
`flickr`	21.459	19.980	18.243	0.245
`wikiconf`	7.234	4.443	0.627	0.148
`wikipedia`	288.502	0.213	0.218	0.173
`youtube`	293.286	9.769	0.204	0.061

higher indexing cost, TTL is still only faster than TopChain in just 2 of the 9 datasets that TTL can handle for querying earliest-arrival time, and in just 2 out of the 9 datasets for querying the duration of a fastest path. The result thus demonstrates the efficiency of our method for answering queries in a temporal graph in real time, while it also shows that TopChain is more scalable than existing methods for processing large temporal graphs.

C. *Study on TopChain Label*

Next, we study the strategy of chain cover and chain ranking used in our labeling scheme. As discussed in Section IV-B, TopChain merges $V_{in}(v)$ and $V_{out}(v)$ into a chain for each v and rank the chains by degree. Here, we also tested two variants of TopChain: (1)**TC1**: we compute the chains by the greedy algorithm of [22] and rank them by degree; and (2)**TC2**: we merge $V_{in}(v)$ and $V_{out}(v)$ into a chain for each v and rank the chains randomly. The index sizes of TC1 and TC2 are almost the same as TopChain. TopChain and TC2 have similar indexing time, but TC1 takes longer time due to chain computation. For query processing, as reported in Table V, TopChain is significantly faster than both TC1 and TC2. This result demonstrates the effectiveness of our labeling scheme in using the properties of temporal graphs.

We also study the effect of k. We report query performance using the two graphs, `austin` and `arxiv`, where the average degree of the transformed graph of `austin` is around 2 (representing graphs with low degree), while it is larger than 20 for `arxiv` (representing graphs with higher degree). Figures 3 and 4 show that a larger k can improve query performance, but it does not help when k is too large. For dataset `austin`, when k is larger than 2, the querying time does not decrease as k increases. Similarly, for dataset `arxiv`, the query performance does not improve when k is larger than 4. This result shows that a small k value is sufficient for good query performance.

D. *Effect of Varying Time Intervals*

The input time interval $[t_\alpha, t_\omega]$ can affect query performance since a smaller interval gives a smaller search space. We tested four different time intervals, I_1 to I_4. We set $I_1 = [0, |T_\mathcal{G}|]$, where $|T_\mathcal{G}|$ is shown in Table II. For each I_i, for $1 \leq i \leq 3$, we divide I_i into two equal sub-intervals so that I_{i+1} is the first sub-interval of I_i.

We used the same 1000 temporal reachability queries tested in Section VII-A, but with different input time interval I_i. Table VII shows that the total querying time of TopChain decreases significantly from time interval I_1 to I_2 for most datasets, and then the decrease becomes slowly when the time intervals become smaller. This is because when the time interval becomes smaller, the reachability also drops significantly and thus more queries can be answered directly by TopChain's pruning strategies. But the pruning effect becomes less and less obvious when the time interval is small enough.

E. *Performance on Dynamic Updating*

We compare TopChain with **Dagger** [26], which is an extension of GRAIL [27] that supports dynamic update. There are other methods that also handle dynamic update in reachability indexing [28], [29], [30], [31], [20], but they can handle only a few smaller graphs that we tested and Dagger is the only one that can scale.

Figure 5 reports the average update time due to edge insertions, where TopChain+ shows the update time including a re-computation of the topological-sort-based labels (presented in Section VI) for each edge insertion. The result shows that re-computing the topological-sort-based labels dominates the updating time of TopChain, but TopChain is still significantly faster than Dagger. Dagger only performs well when the graph has low average degree, e.g., `amazon`. For query performance, Dagger is worse than GRAIL++ [19], which is significantly slower than TopChain as shown in Table V.

F. *Scalability Tests*

We generated synthetic temporal graphs for scalability tests. We varied the number of vertices $|\mathcal{V}|$ from 1M to 8M (M $= 10^6$), the value of π from 50 to 200 which controls the number of multiple temporal edges between two vertices, and the average vertex degree $d_{avg}(u, \mathcal{G})$ from 5 to 20. We generated three sets of datasets by varying one of $|\mathcal{V}|$, π and $d_{avg}(u, \mathcal{G})$, while fixing the other two to their default values: $|\mathcal{V}| = $ 2M, $\pi = 100$ and $d_{avg}(u, \mathcal{G}) = 10$.

We compare TopChain with IP+, Ferrari and GRAIL++ since only these methods can scale to large graphs. Since the indexing time and index size of these four methods are comparable, we only report the total querying time for 1000 randomly generated queries.

Figure 6(a) shows that all the methods scale roughly linearly as $|\mathcal{V}|$ increase, but the querying time of TopChain increases more slowly when $|\mathcal{V}|$ becomes larger, i.e., from 4M to 8M. Figure 6(b) shows that the querying time does not

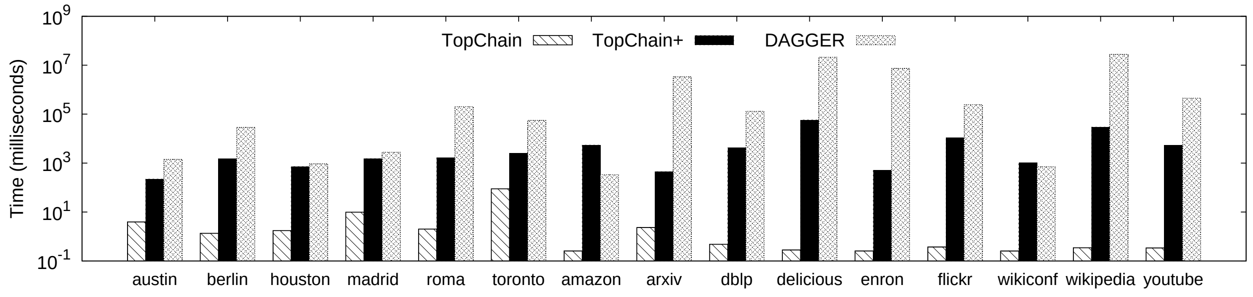


Figure 5. Average update time due to edge insertion

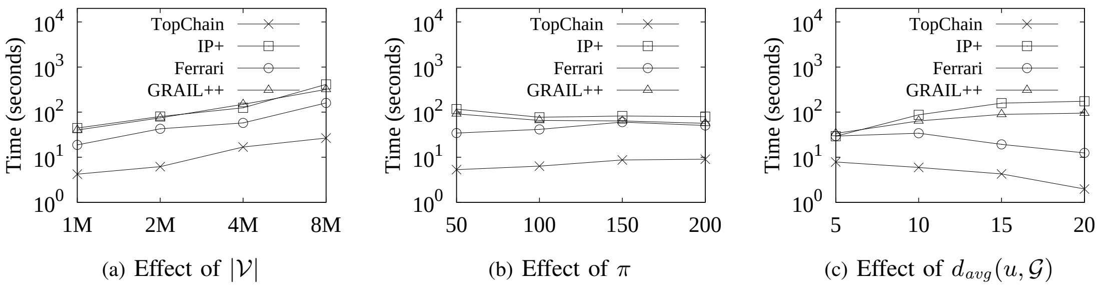


Figure 6. Total querying time on synthetic power-law temporal graphs

change significantly with the increase in π. But the effect of $d_{avg}(u, \mathcal{G})$ is very different on the four methods, as shown in Figure 6(c). While the querying time of IP+ and GRAIL++ increases linearly as $d_{avg}(u, \mathcal{G})$ increases, the query time of TopChain and Ferrari even decreases. This is because as $d_{avg}(u, \mathcal{G})$ increases, more vertices become reachable from others. TopChain and Ferrari can directly answer queries where the source query vertex can reach the target query vertex, while IP+ and GRAIL++ cannot avoid online search for such queries.

Overall, the results show that TopChain and Ferrari have better scalability than IP+ and GRAIL++, while TopChain is significantly faster than the other methods in all cases.

VIII. Related Work

We focus our discussion on work related to reachability indexing, as our method is also an indexing scheme for reachability querying. Readers may refer to the full version of this paper [24] for related work on temporal graphs.

Existing reachability indexes can be mainly categorized into three groups: *Transitive Closure*, *2-Hop Labels*, *Label+Search*. The transitive closure (TC) of a vertex v is the set of vertices that v can reach in G. Since the TCs are too large, existing methods in this category mainly attempt to reduce the TCs by various compression schemes [32], [25], [33], [23], [34], [16], [35]. These methods are not scalable due to the high indexing cost. The 2-hop labeling scheme was first introduced in [21], which proved that computing a 2-hop label with minimum size is NP-hard, and proposed a $(\log |V|)$-approximation. Many following works have attempted to reduce the label size by various heuristics [36], [37], [38], [12], [13], [39], [18], [20], but all these methods are costly and cannot scale to large graphs. TTL [11] is also a 2-hop indexing method, which is designed to answer queries of earliest-arrival time and the duration of a fastest path between two vertices within a time interval in a temporal graph. TTL cannot scale to large temporal graphs due to its expensive indexing cost, and its performance was only verified using small temporal graphs in [11]. TTL also does not support dynamic update, which is necessary for temporal graphs.

The methods [40], [14], [15], [41], [17], [19] in the category of Label+Search construct a small index with a small construction cost, but their query performance is generally much worse than methods in the other two categories. However, the recent methods, *Ferrari* [14] and *IP+* [17], are able to achieve comparable query performance with methods in the other two categories. Ferrari applies tree cover to derive intervals so that reachability queries can be answered by checking the intervals [32]. However, the number of intervals is too large and Ferrari keeps only up to k approximate intervals for every vertex. IP+ selects the top k vertices, ranked based on independent permutation, that a vertex v can reach or that can reach v to be in $L_{out}(v)$ or $L_{in}(v)$. Thus, online search is needed to process some queries for both Ferrari and IP+.

TopChain is also a Label+Search approach, and the key idea of bounding the label size is similar to IP+ and Ferrari. However, our method is the only one that uses the properties of a temporal graph to design the labeling scheme. There are non-trivial issues in using these properties, and TopChain also supports efficient dynamic update based on the graph properties, while both Ferrari and IP+ do not support update.

IX. Conclusions

We presented TopChain, an efficient labeling scheme that employs the properties of a temporal graph for answering temporal reachability queries and time-based path queries. TopChain has a linear index construction time and linear index size, which makes the method scalable. TopChain significantly outperforms the state-of-the-art indexes [14], [16], [11], [17], [19], [20], and supports efficient dynamic update. As temporal graphs can be used to model many networks with time-ordered activities, TopChain is a useful tool for analyzing these graphs. We also plan to apply TopChain to develop scalable systems for processing temporal graphs based on our existing work such as Husky [42], Quegel [43], Blogel [44] and Pregel+ [45].

Acknowledgments. We thank the reviewers for their valuable comments. The authors are partially supported by the Hong Kong GRF 2150851, the AcRF Tier-1 Grant (RG135/14) from Ministry of Education of Singapore and the SUG Grant M4081416.020 from Nanyang Technological University, and project #8115048 of the Shun Hing Institute of Advanced Engineering, the Chinese University of Hong Kong.

REFERENCES

[1] 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.

[2] R. K. Pan and J. Saramäki, "Path lengths, correlations, and centrality in temporal networks," *Phys. Rev. E*, vol. 84, p. 016105, 2011.

[3] N. Santoro, W. Quattrociocchi, P. Flocchini, A. Casteigts, and F. Amblard, "Time-varying graphs and social network analysis: Temporal indicators and metrics," *CoRR*, vol. abs/1102.0629, 2011.

[4] J. Tang, M. Musolesi, C. Mascolo, and V. Latora, "Characterising temporal distance and reachability in mobile and online social networks," *Computer Communication Review*, vol. 40, no. 1, pp. 118–124, 2010.

[5] A. Clementi and F. Pasquale, "Information spreading in dynamic networks: An analytical approach," *Theoretical Aspects of Distributed Computing in Sensor Networks*, 2010.

[6] A. Casteigts, P. Flocchini, B. Mans, and N. Santoro, "Measuring temporal lags in delay-tolerant networks," in *IPDPS*, 2011, pp. 209–218.

[7] G. Kossinets, J. M. Kleinberg, and D. J. Watts, "The structure of information pathways in a social communication network," in *KDD*, 2008, pp. 435–443.

[8] J. Tang, S. Scellato, M. Musolesi, C. Mascolo, and V. Latora, "Small-world behavior in time-varying graphs," *Physical Review E*, vol. 81, no. 5, p. 055101, 2010.

[9] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, "Path problems in temporal graphs," *PVLDB*, vol. 7, no. 9, pp. 721–732, 2014.

[10] 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.

[11] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou, "Efficient route planning on public transportation networks: A labelling approach," in *SIGMOD*, 2015, pp. 967–982.

[12] J. Cheng, S. Huang, H. Wu, and A. W. Fu, "TF-Label: a topological-folding labeling scheme for reachability querying in a large graph," in *SIGMOD*, 2013, pp. 193–204.

[13] R. Jin and G. Wang, "Simple, fast, and scalable reachability oracle," *PVLDB*, vol. 6, no. 14, pp. 1978–1989, 2013.

[14] S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum, "FERRARI: flexible and efficient reachability range assignment for graph indexing," in *ICDE*, 2013, pp. 1009–1020.

[15] S. Trißl and U. Leser, "Fast and practical indexing and querying of very large graphs," in *SIGMOD*, 2007, pp. 845–856.

[16] S. J. van Schaik and O. de Moor, "A memory efficient reachability data structure through bit vector compression," in *SIGMOD*, 2011, pp. 913–924.

[17] H. Wei, J. X. Yu, C. Lu, and R. Jin, "Reachability querying: An independent permutation labeling approach," *PVLDB*, vol. 7, no. 12, pp. 1191–1202, 2014.

[18] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida, "Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths," in *CIKM*, 2013.

[19] H. Yildirim, V. Chaoji, and M. J. Zaki, "GRAIL: a scalable index for reachability queries in very large graphs," *VLDB J.*, vol. 21, no. 4, pp. 509–534, 2012.

[20] A. D. Zhu, W. Lin, S. Wang, and X. Xiao, "Reachability queries on large dynamic graphs: a total order approach," in *SIGMOD*, 2014, pp. 1323–1334.

[21] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," in *SODA*, 2002, pp. 937–946.

[22] K. Simon, "An improved algorithm for transitive closure on acyclic digraphs," *TCS*, vol. 58, pp. 325–346, 1988.

[23] H. V. Jagadish, "A compression technique to materialize transitive closure," *TODS*, vol. 15, no. 4, pp. 558–598, 1990.

[24] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke, "Efficient processing of reachability and time-based path queries in a temporal graph," *CoRR*, vol. abs/1601.05909, 2016.

[25] Y. Chen and Y. Chen, "An efficient algorithm for answering graph reachability queries," in *ICDE*, 2008, pp. 893–902.

[26] H. Yildirim, V. Chaoji, and M. J. Zaki, "DAGGER: A scalable index for reachability queries in large dynamic graphs," *CoRR*, 2013.

[27] ——, "GRAIL: scalable reachability index for large graphs," *PVLDB*, vol. 3, no. 1, pp. 276–284, 2010.

[28] R. Bramandia, B. Choi, and W. K. Ng, "Incremental maintenance of 2-hop labeling of large graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 5, pp. 682–698, 2010.

[29] C. Demetrescu and G. F. Italiano, "Fully dynamic all pairs shortest paths with real edge weights," *J. Comput. Syst. Sci.*, vol. 72, no. 5, pp. 813–837, 2006.

[30] L. Roditty and U. Zwick, "A fully dynamic reachability algorithm for directed graphs with an almost linear update time," in *STOC*, 2004, pp. 184–191.

[31] R. Schenkel, A. Theobald, and G. Weikum, "Efficient creation and incremental maintenance of the HOPI index for complex XML document collections," in *ICDE*, 2005, pp. 360–371.

[32] R. Agrawal, A. Borgida, and H. V. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases," in *SIGMOD*, 1989, pp. 253–262.

[33] Y. Chen and Y. Chen, "Decomposing DAGs into spanning trees: A new way to compress transitive closures," in *ICDE*, 2011, pp. 1007–1018.

[34] R. Jin, N. Ruan, Y. Xiang, and H. Wang, "Path-tree: An efficient reachability indexing scheme for large directed graphs," *ACM Trans. Database Syst.*, vol. 36, no. 1, p. 7, 2011.

[35] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu, "Dual labeling: Answering graph reachability queries in constant time," in *ICDE*, 2006, p. 75.

[36] R. Bramandia, B. Choi, and W. K. Ng, "On incremental maintenance of 2-hop labeling of graphs," in *WWW*, 2008, pp. 845–854.

[37] J. Cai and C. K. Poon, "Path-hop: efficiently indexing large graphs for reachability queries," in *CIKM*, 2010, pp. 119–128.

[38] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu, "Fast computing reachability labelings for large graphs with high compression rate," in *EDBT*, 2008, pp. 193–204.

[39] R. Schenkel, A. Theobald, and G. Weikum, "HOPI: an efficient connection index for complex XML document collections," in *EDBT*, 2004, pp. 237–255.

[40] L. Chen, A. Gupta, and M. E. Kurul, "Stack-based algorithms for pattern matching on dags," in *VLDB*, 2005, pp. 493–504.

[41] R. R. Veloso, L. Cerf, W. M. Junior, and M. J. Zaki, "Reachability queries in very large graphs: A fast refined online search approach," in *EDBT*, 2014, pp. 511–522.

[42] F. Yang, J. Li, and J. Cheng, "Husky: Towards a more efficient and expressive distributed computing framework," *PVLDB*, vol. 9, no. 5, pp. 420–431, 2016.

[43] D. Yan, J. Cheng, T. Ozsu, F. Yang, Y. Lu, J. C. S. Lui, Q. Zhang, and W. Ng, "A general-purpose query-centric framework for querying big graphs [innovative systems and applications]," *PVLDB*, vol. 9, no. 7, 2016.

[44] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," *PVLDB*, vol. 7, no. 14, pp. 1981–1992, 2014.

[45] ——, "Effective techniques for message reduction and load balancing in distributed graph computation," in *WWW*, 2015, pp. 1307–1317.