The VLDB Journal (2022) 31:629–647
https://doi.org/10.1007/s00778-021-00715-z

REGULAR PAPER

Span-reachability querying in large temporal graphs

Dong Wen[1] · Bohua Yang[2] · Ying Zhang[2] · Lu Qin[2] · Dawei Cheng[3] · Wenjie Zhang[1]

Received: 25 February 2021 / Revised: 22 August 2021 / Accepted: 27 October 2021 / Published online: 23 November 2021
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2021

Abstract

Reachability is a fundamental problem in graph analysis. In applications such as social networks and collaboration networks, edges are always associated with timestamps. Most existing works on reachability queries in temporal graphs assume that two vertices are related if they are connected by a path with non-decreasing timestamps (time-respecting) of edges. This assumption fails to capture the relationship between entities involved in the same group or activity with no time-respecting path connecting them. In this paper, we define a new reachability model, called span-reachability, designed to relax the time order dependency and identify the relationship between entities in a given time period. We adopt the idea of two-hop cover and propose an index-based method to answer span-reachability queries. Several optimizations are also given to improve the efficiency of index construction and query processing. We conduct extensive experiments on eighteen real-world datasets to show the efficiency of our proposed solution.

Keywords Temporal graph · Reachability · Dynamic graph

1 Introduction

Computing the reachability between vertices is a fundamental problem in network analysis. A *true* result is returned if there exists a path connecting two query vertices. Extensive studies have been done to answer the reachability queries in graphs [2,9,11,13,19,24,27,29,31,35,36], a problem which has applications across a wide range of domains such as road networks, social networks, collaboration networks, PPI (protein-protein-interaction) networks, XML and RDF databases.

In real-world applications, edges in graphs are often associated with temporal information. For example, in collaboration networks, each vertex is a researcher, and an edge represents the co-authorship of two researchers at a time. In social networks, an edge with a timestamp t represents a communication (sending a message or leaving a comment) between two users at t. Due to the widely spread temporal information in entity relationships, research problems in temporal graphs have recently drawn a lot of attention.

Motivation. In this paper, we study the vertex reachability problem in temporal graphs. An existing method to model the temporal reachability is based on the concept of time-respecting paths [17,18,21]. Specifically, a vertex u reaches v if there exists a path connecting u and v such that the times on the path follow a non-decreasing order. For example, in the temporal graph $\mathcal{G}$ of Fig. 1, v_6 reaches v_{10} since there exists a path $\{\langle v_6, v_2, 5\rangle, \langle v_2, v_1, 6\rangle, \langle v_1, v_{10}, 8\rangle\}$ connecting them and the times 5, 6, 8 are in a non-decreasing order. Semertzidis et al. [25] also model the temporal reachability that two vertices u, v are reachable if there exists path connecting them and the times of all edges in the path are consistent, i.e., u, v are reachable in a snapshot of the temporal graph at a given time.

✉ Dawei Cheng
dcheng@tongji.edu.cn

Dong Wen
dong.wen@unsw.edu.au

Bohua Yang
bohua.yang@student.uts.edu.au

Ying Zhang
ying.zhang@uts.edu.au

Lu Qin
lu.qin@uts.edu.au

Wenjie Zhang
zhangw@cse.unsw.edu.au

1 The University of New South Wales, Kensington, Australia

2 AAII, University of Technology Sydney, Ultimo, Australia

3 Department of Computer Science and Technology, Tongji University, Shanghai, China

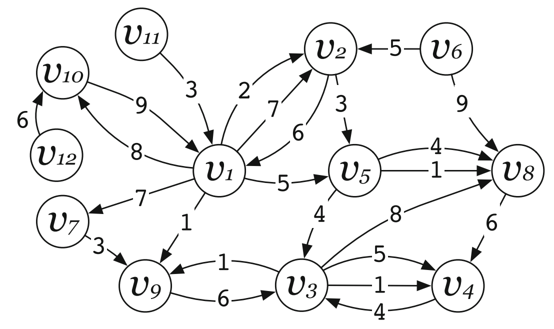

Fig. 1 A temporal graph $\mathcal{G}$ where each number represents the timestamp of the edge below

In many scenarios of temporal graph mining, we may only focus on the relationship between vertices in the projected graph of a small time interval without addressing any order limitation in the edge sequence. Here, the projected graph is the static graph containing all edges at times falling in the interval. For example, Gurukar el al. [16] compute the communication motifs in temporal graphs and show that two edges sharing a common vertex are related if the difference of their timestamps is very small. Authors in [22,28] compute the community structures called Δ-clique and (θ, k)-persistent-core, respectively, in temporal graphs. Their models require that the resulting subgraph satisfies some structural properties (e.g., vertex degree threshold) in the projected graph of a time interval. The aforementioned two reachability models are too strict and might fail to capture entity relationship in these scenarios.

Span-reachability. In this paper, we define a span-reachability model. Given a temporal graph and a time interval $\mathcal{I}$, we say a vertex u span-reaches v if u reaches v in the projected graph of $\mathcal{I}$. We investigate the problem of efficiently answering span-reachability queries for an arbitrary pair of vertices and a time interval.

Example 1 In the temporal graph $\mathcal{G}$ of Fig. 1, we have v_1 span-reaches v_8 in the time interval [3, 5], since there exists a path $\{\langle v_1, v_5, 5\rangle, \langle v_5, v_8, 4\rangle\}$ from v_1 to v_8 in the projected graph of [3, 5].

Applications. Using this model, we can effectively analyze the potential relationship between entities by focusing on the item interactions in a specific period. Several real-world applications can benefit from this study. We provide several examples as follows.

- *Biology analysis.* In PPI networks, it is important to identify whether two proteins participate in a common biological process or molecular function. According to [20], proteins or RNA can be described as vocabulary terms. The relationships between vocabulary terms can be modeled as a gene ontology (GO) directed acyclic graph (DAG) in which each vertex is a concept or vocabulary term. In monitoring the protein activities, our model can be used to identify the relationship between proteins based on GO DAG.

- *Security assessment and recommendation.* In the context of assessing security, we need to understand whether certain person are related to a known terrorist [5]. In organizing a terrorist activity, there may exist several phone calls among the suspects with a short period. We may be not able to find a time-respecting path from the known terrorist to others, especially when not all people in the organization take orders from this terrorist. Our model can be used to capture the related suspects of a targeted terrorist. Similarly, in social networks, our model can be used to detect whether two users are involved in a group in the period of big social events, such as FIFA World Cup and Olympic Games.

- *Money transaction monitor.* In e-commerce platforms and bank systems, we have a graph in which each vertex represents a user account and each edge with a timestamp represents a money transaction between two user accounts. In monitoring money transactions, or some other illegal financial activities, such as money laundering and fake transactions, it is crucial to detect whether there exists a path between two user accounts. Normally, a series of money transactions should follow an increasing order of timestamps. However, some skilled users may borrow some untraceable money to finish the transfer and try to dodge any monitoring. For example, an account in the transaction path may transfer the money to the next account in advance and receive the money from the prior account later. The existing order-dependent reachability model cannot capture this activity, but our model can be used here by setting a specified time interval.

Based on the concept of span-reachability, we also study a θ-reachability problem, which is a generalized version of the span-reachability. Given a time interval $\mathcal{I}$ and a length threshold θ, two vertices are θ-reachable in $\mathcal{I}$ if they are span-reachable in a θ-length subinterval of $\mathcal{I}$. Taking the above application of monitoring money transactions, a more general task is to identify whether there exists a transaction chain between two accounts finished in a short period over a long monitoring period. Note that when the length of query interval equals to θ, θ-reachability is equivalent to span-reachability. When θ is 1, it is equivalent to the disjunctive historical reachability model studied in [25].

Online solution. Given a time interval $\mathcal{I}$, a straightforward method to answer span-reachability queries is to perform a bidirectional modified breath-first search between two query vertices. We only scan the edges in the query interval and return *true* if a common vertex is found in the searches of two query vertices. This method works but incurs high computational cost especially when the graph is very large.

Index-based solution. To efficiently and scalable process the span-reachability query, we propose an index-based solution based on the concept of two-hop cover (sometimes called hop labeling) [1,3]. The index is called Time Interval Labeling (TILL-Index). Specifically, for each vertex u in the temporal graph, we maintain an out-label set $\mathcal{L}_{out}(u)$ and an in-label set $\mathcal{L}_{in}(u)$. Each item in $\mathcal{L}_{out}(u)$ (resp. $\mathcal{L}_{in}(u)$) is a triplet $\langle w, t_s, t_e\rangle$ which means that u span-reaches (resp. is span-reachable from) w in the interval $[t_s, t_e]$. Given a query interval $[t_1, t_2]$, we answer the span-reachability from a vertex u to a vertex v by checking $\mathcal{L}_{out}(u)$ and $\mathcal{L}_{in}(v)$. We have u span-reaches v if there exists a common vertex w such that u span-reaches w in a subinterval of $[t_1, t_2]$ and v is also span-reachable from w in a subinterval of $[t_1, t_2]$.

Efficiently computing a small-size TILL-Index is not a trivial task. We construct the index in n iterations where n is the number of vertices in the temporal graph. In each iteration, we pick a vertex u to compute all its reachable vertices with corresponding time interval and add u to the in-label or out-label set of other vertices if necessary. This index construction algorithm incorporates several optimizations. First, we use a priority queue to explore the reachable vertices of the picked vertex u in each iteration. Based on the priority queue, our first step is always to process the vertex with the shortest time interval that is reachable from u. This guarantees that each found vertex reachable from u with a corresponding interval is never dominated by others and significantly reduces unnecessary visits. In addition, by studying the dominance relationship between different intervals, we stop exploring neighbors of a visited reachable vertex if a specific condition is satisfied. This pruning rule significantly reduces the search space for each vertex. Real-world temporal graphs are highly dynamic. We also propose algorithms to maintain our TILL-Index when inserting new edges and deleting old edges.

Note that even though the concept of the two-hop cover has been studied or used in several existing works [1,3,13,30], our method is not a naive extension of existing techniques. Unlike the previous studies, our method exploits the characteristics of temporal graphs. The proposed optimizations for index construction center mainly on the relationships between different time intervals, such as containment and intersection. We also propose several optimization techniques to improve the efficiency of query processing.

Contributions. We summarize the main contributions in this paper as follows.

- *A novel reachability model in temporal graphs.* We define a span-reachability model to capture the interactions between entities in a specific period of a temporal graph. In addition, we further study the θ-reachability problem which is a generalized version of the span-reachability.
- *A two-hop index-based solution.* We exploit the characteristics of the span-reachability model and adopt the idea of two-hop cover to propose an index-based method to answer both research problems.
- *Several optimizations to improve the efficiency of index construction and query processing.* We propose two optimizations to improve the efficiency of index construction. We also use a sliding window-like method to improve the efficiency of θ-reachability query processing.
- *Efficient index maintenance in dynamic temporal graphs.* We propose algorithms to maintain the index when inserting new edges and removing out-of-date edges, respectively.
- *Extensive performance studies on more than ten real-world datasets.* We conduct experiments on eighteen real-world datasets from different categories. The results demonstrate the effectiveness of our optimizations and the efficiency of our proposed solutions.

Organization. The rest of this paper is organized as follows. Section 2 introduces some background knowledge and defines the problem. Section 3 gives an overview of our index-based solution. Section 4 studies the index construction algorithms. Section 5 studies the query processing algorithms. Section 6 introduces algorithms for the index maintenance. Section 7 reports the experimental results. Section 8 introduces related works, and Sect. 9 concludes the paper. The paper is extended from a conference version [32]. We additionally propose algorithms to maintain the index for dynamic temporal graphs and conduct corresponding experiments. We omit the detailed proofs for several lemmas and theorems when they are straightforward.

2 Preliminary

Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be a directed temporal graph, where $\mathcal{V}$ and $\mathcal{E}$ denote the set of vertices and the set of temporal edges, respectively. Each temporal edge $e \in \mathcal{E}$ is a triplet $\langle u, v, t\rangle$, where u, v are the vertices in $\mathcal{V}$ and t is the connection time from u to v. Without loss of generality, we assume t is an integer since the timestamp in real-world applications is normally an integer. Note that there may exist multiple edges connecting the same pair of vertices at different times. We use $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$ to denote the number of vertices and the number of temporal edges, respectively. Given a vertex $u \in \mathcal{V}$, the out-neighbor set of u is defined as $N_{out}(u) = \{\langle v, t\rangle | (u, v, t) \in \mathcal{E}\}$, and the in-neighbor set is defined similarly. The out-degree (resp. in-degree) of u is denoted as $degr_{out}(u) = |N_{out}(u)|$ (resp. $degr_{in}(u) = |N_{in}(u)|$). Given a time interval $[t_s, t_e]$, the projected graph of $\mathcal{G}$ in $[t_s, t_e]$, denoted by $\mathcal{G}_{[t_s,t_e]}$, where $V(\mathcal{G}_{[t_s,t_e]}) = \mathcal{V}$ and $E(\mathcal{G}_{[t_s,t_e]}) = \{(u, v) | (u, v, t) \in \mathcal{E}, t \in [t_s, t_e]\}$. The length or width of an interval $[t_s, t_e]$ is the number of timestamps in the interval, i.e., $t_e - t_s + 1$. Given the temporal graph $\mathcal{G}$ in Fig. 1, its projected graph in the interval [2, 4] is given in Fig. 2.

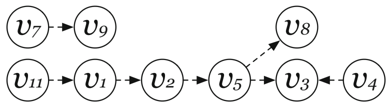

Fig. 2 The projected static graph of $\mathcal{G}$ in the time interval [2, 4]

Based on the concept of the projected graph, we define the span-reachability as follows.

Definition 1 (Span- Reachability) Given a temporal graph $\mathcal{G}$, two vertices u, v and a time interval $[t_s, t_e]$, u span-reaches v in $[t_s, t_e]$, denoted as $u \rightsquigarrow_{[t_s,t_e]} v$, if u reaches v in the projected graph $\mathcal{G}_{[t_s,t_e]}$.

Considering the temporal graph $\mathcal{G}$ in Fig. 1, we have $v_1 \rightsquigarrow_{[2,4]} v_3$ since v_1 reaches v_3 in the projected graph of [2, 4] in Fig. 2. We define the first problem studied in this paper based on Definition 1 as follows.

Problem 1 Given a temporal graph $\mathcal{G}$, two vertices u, v and a time interval $\mathcal{I}$, we aim to efficiently answer whether u span-reaches v in the interval $\mathcal{I}$.

In addition to identifying the span-reachability, we further define a generalized reachability model in a temporal graph $\mathcal{G}$. Given two intervals $[t'_s, t'_e]$ and $[t_s, t_e]$, we have $[t'_s, t'_e] \subseteq [t_s, t_e]$ iff $t'_s \geq t_s$ and $t'_e \leq t_e$.

Definition 2 (θ-Reachability) Given a temporal graph $\mathcal{G}$, two vertices u, v, a parameter θ and a time interval $[t_s, t_e]$ s.t. $t_e - t_s + 1 \geq \theta$, u θ-reaches v if there exists an interval $[t'_s, t'_e] \subseteq [t_s, t_e]$ such that $t'_e - t'_s + 1 = \theta$ and u reaches v in $\mathcal{G}_{[t'_s,t'_e]}$.

Example 2 Given the temporal graph $\mathcal{G}$ in Fig. 1, let $\theta = 3$. We have v_1 3-reaches v_8 in the interval [1, 5] since there exists an interval $[3, 5] \subseteq [1, 5]$ such that the length of [3, 5] is 3 and v_1 reaches v_8 in the projected graph $\mathcal{G}_{[3,5]}$.

Relationship of two reachability models. Given an arbitrary pair of vertices u, v, a threshold θ and a time interval $\mathcal{I}$, we also study the issue of computing θ-reachability from u to v in $\mathcal{I}$, denoted by Problem 2. Definition 1 is a special case of Definition 2 when θ is equal to the length of the input interval. We also see a growing strictness from Definition 1 to Definition 2, which is shown in the following lemma.

Lemma 1 *Given two vertices u, v and an interval $\mathcal{I}$, u span-reaches v in $\mathcal{I}$ if u θ-reaches v in $\mathcal{I}$.*

For ease of presentation, we assume the input temporal graph is a directed graph, and our proposed techniques can easily handle undirected graphs. We omit the proofs of several lemmas and theorems when they are straightforward due to space limitation.

3 Solution overview

We give an overview of our solution in this section. We start by presenting a straightforward online algorithm for our research problems and then introduce several basic ideas of our index-based method.

3.1 A straightforward online approach

Given a time interval $[t_s, t_e]$, the span-reachability of two vertices u and v in $[t_s, t_e]$ can be answered by a modified bidirectional breath-first search. Specifically, we begin by alternatively picking one of u and v in each round and exploring the unvisited vertices that are reachable from u or can reach v. We have u reaches v once the search scopes of two vertices intersect. The detailed pseudocode of this approach is given in Algorithm 1. Note that we assume $u \neq v$ in all proposed algorithms to answer the reachability queries in this paper. Alternatively, we directly return *true* without the algorithm invocation.

In line 1, R_u and R_v are used to collect all vertices that u can reach and all vertices that can reach v, respectively. In line 5, $Q_u \cup Q_v = \emptyset$ means there does not exist any unexplored vertex for both u and v. The variable $toggle$ initialized in line 4 represents the processed vertex in the last iteration, and we process u in lines 7–15 if $toggle = v$. We explore the out-neighbors of all vertices in the queue in lines 9–15. In line 11, we only access edges whose time falls into the input interval. We return *true* if a common vertex of R_u and R_v is found in line 12, or push the new found vertex into the queue in line 14. The algorithm essentially performs a bidirectional BFS in the projected graph $\mathcal{G}_{[t_1,t_2]}$. The time complexity of Algorithm 1 is given as follows.

Lemma 2 *The running time of Algorithm 1 is bounded by $O(m + n)$.*

Problem 2 can be answered by invoking Algorithm 1 as a subroutine. We can sequentially check each possible θ-length subinterval in the given query interval $[t_1, t_2]$ and return *true* immediately if u reaches v in any one of them. In the worst case, the time complexity of this algorithm is bounded by $O((t_2 - t_1 - \theta) \cdot (n + m))$.

Even though the bidirectional search method can successfully answer span-reachability queries and θ-reachability queries, the algorithms suffer from a poor scalability since the whole graph may be visited during query processing. To improve query efficiency, we propose an index-based method in the following section.

3.2 The time interval labeling index

We introduce our index structure called Time Interval Labeling (TILL-Index) in this section. TILL-Index adopts the idea

Algorithm 1: Online-Reach()

```
   Input: a temporal graph $\mathcal{G}$, two vertices $u$ and $v$ and an interval $[t_1, t_2]$
   Output: the span-reachability of $u$ and $v$ in $[t_1, t_2]$
1  $R_u \leftarrow \{u\}$, $R_v \leftarrow \{v\}$;
2  $Q_u \leftarrow$ a queue containing $u$;
3  $Q_v \leftarrow$ a queue containing $v$;
4  $toggle \leftarrow v$;
5  while $Q_u \cup Q_v \neq \emptyset$ do
6      if $toggle = v \wedge Q_u \neq \emptyset$ then
7          $toggle \leftarrow u$;
8          $l \leftarrow |Q_u|$;
9          for $1 \leq i \leq l$ do
10             $w \leftarrow Q_u.pop()$;
11             foreach $\langle w', t\rangle \in N_{out}(w) : t \in [t_1, t_2]$ do
12                 if $w' \in R_v$ then return true;
13                 if $w' \notin R_u$ then
14                     $Q_u.push(w')$;
15                     $R_u \leftarrow R_u \cup \{w'\}$;
16     else
17         repeat lines 7–15 to search the vertices that reach $v$ by
           toggling between $u$ and $v$, and replacing the subscript $out$
           with $in$
18 return false;
```

Table 1 A Time Interval Labeling of $\mathcal{G}$

$\mathbf{L_{in}(v_2)}$	$\langle v_1, 2, 2\rangle$	$\langle v_1, 7, 7\rangle$	$\mathbf{L_{out}(v_2)}$	$\langle v_1, 6, 6\rangle$
$\mathbf{L_{in}(v_3)}$	$\langle v_1, 2, 4\rangle$	$\langle v_1, 4, 5\rangle$	$\langle v_2, 3, 4\rangle$	$\mathbf{L_{in}(v_4)}$
$\langle v_1, 1, 4\rangle$	$\langle v_1, 4, 5\rangle$	$\langle v_2, 3, 5\rangle$	$\langle v_2, 1, 4\rangle$	$\langle v_3, 1, 1\rangle$
$\langle v_3, 5, 5\rangle$	$\langle v_3, 6, 8\rangle$	$\mathbf{L_{out}(v_4)}$	$\langle v_3, 4, 4\rangle$	$\mathbf{L_{in}(v_5)}$
$\langle v_1, 2, 3\rangle$	$\langle v_1, 5, 5\rangle$	$\langle v_2, 3, 3\rangle$	$\mathbf{L_{out}(v_5)}$	$\langle v_3, 4, 4\rangle$
$\mathbf{L_{out}(v_6)}$	$\langle v_1, 5, 6\rangle$	$\langle v_2, 5, 5\rangle$	$\langle v_4, 6, 9\rangle$	$\mathbf{L_{in}(v_7)}$
$\langle v_1, 7, 7\rangle$	$\mathbf{L_{out}(v_7)}$	$\langle v_3, 3, 6\rangle$	$\mathbf{L_{in}(v_8)}$	$\langle v_1, 1, 3\rangle$
$\langle v_1, 2, 4\rangle$	$\langle v_1, 4, 5\rangle$	$\langle v_2, 1, 3\rangle$	$\langle v_2, 3, 4\rangle$	$\langle v_3, 8, 8\rangle$
$\langle v_5, 1, 1\rangle$	$\langle v_5, 4, 4\rangle$	$\langle v_6, 9, 9\rangle$	$\mathbf{L_{out}(v_8)}$	$\langle v_3, 4, 6\rangle$
$\langle v_4, 6, 6\rangle$	$\mathbf{L_{in}(v_9)}$	$\langle v_1, 1, 1\rangle$	$\langle v_1, 3, 7\rangle$	$\langle v_2, 1, 4\rangle$
$\langle v_3, 1, 1\rangle$	$\langle v_7, 3, 3\rangle$	$\mathbf{L_{out}(v_9)}$	$\langle v_3, 6, 6\rangle$	$\mathbf{L_{in}(v_{10})}$
$\langle v_1, 8, 8\rangle$	$\mathbf{L_{out}(v_{10})}$	$\langle v_1, 9, 9\rangle$	$\mathbf{L_{out}(v_{11})}$	$\langle v_1, 3, 3\rangle$
$\mathbf{L_{out}(v_{12})}$	$\langle v_1, 6, 9\rangle$	$\langle v_{10}, 6, 6\rangle$		

of two-hop cover (or two-hop labeling) [1,3]. In a nutshell, for each vertex u, we maintain an in-label set $\mathcal{L}_{in}(u)$ and an out-label set $\mathcal{L}_{out}(u)$. Each item in $\mathcal{L}_{in}(u)$ is a triplet $\langle w, t_s, t_e\rangle$ which means that w reaches u in the projected graph $\mathcal{G}_{[t_s,t_e]}$. Each item in $\mathcal{L}_{out}(u)$ is a triplet $\langle w, t_s, t_e\rangle$ which means that u reaches w in $\mathcal{G}_{[t_s,t_e]}$. A triplet is called a w-triplet if the first item of the triplet is w. We call $\langle u, v, t_s, t_e\rangle$ a reachability tuple if $u \rightsquigarrow_{[t_s,t_e]} v$, and we say a vertex w covers a reachability tuple $\langle u, v, t_s, t_e\rangle$ if $u \rightsquigarrow_{[t_s,t_e]} w$ and $w \rightsquigarrow_{[t_s,t_e]} v$. For ease of presentation, we focus mainly on Problem 1 now. Problem 2 can also be solved based on the TILL-Index, and Sect. 5 will discuss its solution in detail by extending the techniques in answering Problem 1. Given two vertices u and v, u span-reaches v in an interval $[t_1, t_2]$ if any one of the following equations holds:

1. $\exists\langle v, t_s, t_e\rangle \in \mathcal{L}_{out}(u)$: $[t_s, t_e] \subseteq [t_1, t_2]$;
2. $\exists\langle u, t_s, t_e\rangle \in \mathcal{L}_{in}(v)$: $[t_s, t_e] \subseteq [t_1, t_2]$;
3. $\exists\langle w, t_s, t_e\rangle \in \mathcal{L}_{out}(u)$, $\langle w', t'_s, t'_e\rangle \in \mathcal{L}_{in}(v)$: $w = w' \wedge [t_s, t_e] \subseteq [t_1, t_2] \wedge [t'_s, t'_e] \subseteq [t_1, t_2]$.

Based on the above equations, a TILL-Index is a minimal index that can be used to answer correctly all possible span-reachability queries in $\mathcal{G}$. Here, by minimal, we mean that removing any item in the index cannot correctly determine all possible span-reachability in the graph. An example of a TILL-Index of the temporal graph $\mathcal{G}$ in Fig. 1 is given in Table 1.

Example 3 Assume that we aim to answer the span-reachability from v_6 to v_3 in the time interval [4, 8]. We first locate the out-label set of v_6 in Table 1, which are $\mathcal{L}_{out}(v_6) = \{\langle v_1, 5, 6\rangle, \langle v_2, 5, 5\rangle, \langle v_4, 6, 9\rangle\}$. The in-label set of v_3 are $\mathcal{L}_{in}(v_3) = \{\langle v_1, 2, 4\rangle, \langle v_1, 4, 5\rangle, \langle v_2, 3, 4\rangle\}$. We can see that there is a common vertex v_1 such that both $\langle v_1, 5, 6\rangle \in \mathcal{L}_{out}(v_6)$ and $\langle v_1, 4, 5\rangle \in \mathcal{L}_{in}(v_3)$ fall in the query interval [4, 8]. Therefore, the answer of this query is *true*.

Even though the idea of two hop cover is simple, it is non-trivial to efficiently compute a small TILL-Index and answer the reachability queries based on the index. We give the details about index construction and query processing in Sect. 4 and Sect. 5, respectively.

Remark 1 One may consider using some existing techniques (e.g., transitive closure) of reachability in static graphs to construct the index for span-reachability. However, the idea is hard to work since we may have an extremely large number of possible query time spans and each time span corresponds a static graph. It is not acceptable to index all possible static graphs.

4 Index construction

4.1 The labeling framework

We begin by presenting several basic concepts before introducing the details of the index construction.

Definition 3 (Dominance and Skyline Reachability Tuple) Given two vertices u and v, a reachability tuple $\langle u, v, t'_s, t'_e\rangle$ dominates $\langle u, v, t_s, t_e\rangle$ if $[t'_s, t'_e] \subset [t_s, t_e]$. A reachability tuple $\langle u, v, t_s, t_e\rangle$ is a skyline (or non-dominated) reachability tuple (SRT) if it is not dominated by other tuples.

Given a vertex u, we also use the term *skyline* in Definition 3 for the triplets in $\mathcal{L}_{out}(u)$ (resp. $\mathcal{L}_{in}(u)$) since a triplet $\langle w, t_s, t_e\rangle \in \mathcal{L}_{out}(u)$ represents a reachability tuple $\langle u, w, t_s, t_e\rangle$. In constructing TILL-Index, we only need to compute labels that can cover all SRTs since a vertex covering an SRT also covers all its dominating tuples. Therefore, our research task in the index construction is to cover all SRTs in the graph with the total index size as small as possible.

The minimum two-hop cover. [13] studies the two-hop cover for the shortest distance and reachability queries in general graphs. They proved that computing the minimum two-hop cover is NP-hard and can be transformed to a minimum cost set cover problem [12]. They use a greedy algorithm to compute a two-hop cover and achieve an $O(\log n)$ approximation factor. The proposed algorithm is inefficient since a procedure of densest subgraph computation is invoked every time they select a vertex to cover several reachability (or shortest distance) vertex pairs.

Hierarchical two-hop cover. The aforementioned theoretical results also hold in our scenario, and we omit the detailed proof. Due to the difficulty of the optimal cover computation, we adopt a hierarchical labeling approach [1,3] which follows a strict total order on the vertices in $\mathcal{G}$, and we will prove the minimality of our TILL-Index under the total order constraint. We use $\mathcal{O}$ to denote the vertex order. We say the rank of a vertex u is higher than that of a vertex v if $\mathcal{O}(u) < \mathcal{O}(v)$. By the total order, we mean to sequentially process each vertex in $\mathcal{O}$. Once we process a vertex w, we add w and corresponding intervals to the labels of u and v for all uncovered reachability tuples containing u, v covered by w. Intuitively, a vertex playing an important role in $\mathcal{G}$ should be put at the front of the order. Next, we adopt the ordering method in [19]. Given each vertex u, we use the formula $(degr_{in}(u) + 1) \times (degr_{out}(u) + 1)$ as the importance of u. We sort the vertices in a decreasing order of their importance and break the tie by selecting a vertex with smaller ID. Given the total vertex order, we immediately have the following lemmas for our TILL-Index.

Lemma 3 *Given an arbitrary vertex u, for every triplet $\langle w, *, *\rangle$ in $\mathcal{L}_{out}(u) \cup \mathcal{L}_{in}(u)$, $\mathcal{O}(w) < \mathcal{O}(u)$.*

Lemma 4 *Given an SRT $\langle u, v, t_s, t_e\rangle$ in $\mathcal{G}$, let w be the first vertex (the highest rank) in $\mathcal{O}$ that can cover $\langle u, v, t_s, t_e\rangle$. $w \neq u \neq v$. There exists a triplet $\langle w, t_s', t_e'\rangle \in \mathcal{L}_{out}(u)$ such that $[t_s', t_e'] \subseteq [t_s, t_e]$ and a triplet $\langle w, t_s'', t_e''\rangle \in \mathcal{L}_{in}(v)$ such that $[t_s'', t_e''] \subseteq [t_s, t_e]$.*

Without loss of generality, we maintain only skyline triplets in labels of TILL-Index since a dominated triplet can be always replaced by a corresponding skyline triplet without influencing calculation's accuracy. We define an important concept in computing TILL-Index as follows.

Definition 4 (Canonical Reachability Tuple) A reachability tuple $\langle u, v, t_s, t_e\rangle$ is a canonical reachability tuple (CRT) if (i) $\langle u, v, t_s, t_e\rangle$ is a skyline reachability tuple, and (ii) there does not exist a vertex w such that $u \rightsquigarrow_{[t_s,t_e]} w$, $w \rightsquigarrow_{[t_s,t_e]} v$, $\mathcal{O}(w) < \mathcal{O}(u)$ and $\mathcal{O}(w) < \mathcal{O}(v)$.

Given a vertex order $\mathcal{O}$ and a vertex u, we say a tuple is an SRT (resp. CRT) of u if the tuple is an SRT (resp. CRT) containing u and the rank of u is higher in the tuple. We have following lemmas based on Definition 4.

Lemma 5 *Given an arbitrary vertex u and any (skyline) triplet $\langle w, t_s, t_e\rangle$ in $\mathcal{L}_{out}(u)$ (resp. $\mathcal{L}_{in}(u)$), $\langle u, w, t_s, t_e\rangle$ (resp. $\langle w, u, t_s, t_e\rangle$) is a CRT.*

Lemma 6 *For each CRT $\langle u, v, t_s, t_e\rangle$ in $\mathcal{G}$, there is a triplet $\langle u, t_s, t_e\rangle$ in $\mathcal{L}_{in}(v)$ if $\mathcal{O}(u) < \mathcal{O}(v)$. If this is not the case, there is a triplet $\langle v, t_s, t_e\rangle$ in $\mathcal{L}_{out}(u)$.*

Example 4 The labels in Table 1 are computed following the total alphabetical order of the vertices in $\mathcal{G}$ of Fig. 1. For the in-labels of v_8, we can find that the rank of all vertices v_1, v_2, v_3, v_5 and v_6 appearing in $\mathcal{L}_{in}(v_8)$ have ranks higher than v_8. For an arbitrary triplet $\langle v_2, 3, 4\rangle$ in $\mathcal{L}_{in}(v_8)$, there does not exist any vertex with a higher rank than v_8 and v_2 that can cover the reachability tuple $\langle v_2, v_8, 3, 4\rangle$.

Based on Lemmas 5 and 6, there is a one-to-one correspondence between CRTs and triplets in TILL-Index. It now follows that we can construct TILL-Index by computing all CRTs. A framework to construct TILL-Index is presented in Algorithm 2.

Algorithm 2: A Framework of Index Construction

```
1 for $1 \le i \le n$ do
2     $u_i \leftarrow$ the $i$-th vertex in the order $\mathcal{O}$;
3     compute all SRTs of $u_i$;
4     compute all CRTs by refining the computed SRTs;
5     add corresponding triplet of each CRT to in-labels or
      out-labels of other vertices;
```

In the framework, we process each vertex sequentially in the vertex order. In line 3, the SRTs of u_i can be computed in two phases. One computes all vertices and corresponding time intervals that are reachable from u, while the other computes those that can reach u. Taking the first one as an example, a basic implementation uses a queue to maintain the discovered reachable triplets of u_i. To be specific, the queue is initialized as a special triplet containing u_i. We iteratively pop a triplet $\langle v, t_s, t_e\rangle$, which means u can reach v in $[t_s, t_e]$. For each out-neighbor $\langle v', t\rangle$ of v, we expand $\langle v, t_s, t_e\rangle$ to $\langle v', \min(t_s, t), \max(t_e, t)\rangle$, which means u_i reaches v' in the interval $[\min(t_s, t), \max(t_e, t)]$. We mark this new triplet $\langle v', \min(t_s, t), \max(t_e, t)\rangle$ as discovered and push it into the queue if it is not dominated by other discovered triplet, and remove all its dominating discovered triplets. In line 4, for every SRT computed in line 3, we check whether there exists a vertex with a higher rank that can cover the SRT based on Definition 4. This can be done by performing a query processing procedure based on the labels computed by higher-rank vertices. The details of query processing will be given in Sect. 5. If yes, we omit such SRT, and derive all CRTs when all SRTs are checked.

4.2 Theoretical analysis

We prove the correctness and the minimality of TILL-Index computed by Algorithm 2.

Theorem 1 (Correctness) *The span-reachability query of any pair of vertices can be correctly answered (any one of three conditions presented in Sect. 3.2 holds) based on the index computed by Algorithm 2.*

Proof The theorem can be easily derived according to Definition 4, Lemma 5 and Lemma 6. □

Theorem 2 (Minimality) *For any vertex u and any triplet $\langle w, t_s, t_e\rangle$ in $\mathcal{L}_{in}(u)$ or $\mathcal{L}_{out}(u)$ of the index computed by Algorithm 2, there exists a pair of vertices u', v' and a corresponding interval $[t'_s, t'_e]$ such that the span-reachability of u' and v' in $[t'_s, t'_e]$ cannot be correctly answered after removing $\langle w, t_s, t_e\rangle$.*

Proof Given a triplet $\langle w, t_s, t_e\rangle \in \mathcal{L}_{out}(u)$, we prove that after removing $\langle w, t_s, t_e\rangle$, the span-reachability from u to w in $[t_s, t_e]$ cannot be correctly answered. If this query can be correctly answered, then at least one of the following two conditions holds: (i) there exists a triplet $\langle u, t'_s, t'_e\rangle$ in $\mathcal{L}_{in}(w)$ such that $[t'_s, t'_e] \subseteq [t_s, t_e]$; (ii) there exists a triplet $\langle v, t'_s, t'_e\rangle \in \mathcal{L}_{out}(u)$ and a triplet $\langle v, t''_s, t''_e\rangle \in \mathcal{L}_{in}(w)$ such that $[t'_s, t'_e] \subseteq [t_s, t_e]$ and $[t''_s, t''_e] \subseteq [t_s, t_e]$.

Given that $\langle w, t_s, t_e\rangle \in \mathcal{L}_{out}(u)$, we have $\mathcal{O}(w) < \mathcal{O}(u)$ according to Lemma 3, and a triplet containing u cannot appear in $\mathcal{L}_{in}(w)$ or $\mathcal{L}_{out}(w)$. Therefore, condition i cannot hold. Condition ii holds if v covers the reachability tuple $\langle u, w, t_s, t_e\rangle$ and the rank of v is higher than those of u and w. This contradicts Lemma 5 that $\langle u, w, t_s, t_e\rangle$ is a CRT. This completes the proof of the theorem. □

As we shown earlier, computing the minimum two-hop cover for both shortest distance and reachability is NP-hard according to [13]. The property still holds for computing the two-hop cover for span-reachability. The proof is similar to that in [13] and is done by transforming the problem into the minimum cost set cover problem [12].

4.3 Implementation

The basic implementation incurs high computational cost. We discuss several techniques to efficiently compute SRTs and CRTs as follows.

4.3.1 Efficient SRT computation

We propose a priority queue based method to efficiently compute all SRTs of a given vertex. A key idea of this method is given in the following lemma.

Lemma 7 *Given a vertex u and a set of known SRTs S containing u, a reachability tuple $\langle u, v, t_s, t_e\rangle$ is an SRT if (i) $\langle u, v, t_s, t_e\rangle$ is not dominated by any other SRT in S, and (ii) the length of $[t_s, t_e]$ is the smallest among those of all tuples that are not in S.*

Example 5 We consider the temporal graph $\mathcal{G}$ in Fig. 1. Assume that we aim to compute SRTs of v_5. For ease of presentation, we only consider the SRTs starting from v_5. Initially, $S = \emptyset$ and we have several reachability tuples with the smallest interval length. They are $\langle v_5, v_3, 4, 4\rangle$, $\langle v_5, v_8, 1, 1\rangle$ and $\langle v_5, v_8, 4, 4\rangle$, and all of them are SRTs. Now we have $S = \{\langle v_5, v_3, 4, 4\rangle, \langle v_5, v_8, 1, 1\rangle, \langle v_5, v_8, 4, 4\rangle\}$. $\langle v_5, v_8, 4, 8\rangle$ is not an SRT since it is dominated by $\langle v_5, v_8, 4, 4\rangle$ in S, and $\langle v_5, v_4, 4, 5\rangle$ is an SRT since its interval length is smallest among all possible reachability tuples except the SRTs in S.

Based on Lemma 7, to compute all non-dominated reachability triplets (a target and the corresponding time interval) from a vertex u, we preserve all discovered reachability triplets in a priority queue and always pop the triplets with the smallest time interval length in the priority queue. According to Lemma 7, a popped triplet $\langle v, t_s, t_e\rangle$ must be an SRT if it is not dominated by any previously found SRT. We compute the new interval of each neighbor of v that can be reached from $\langle v, t_s, t_e\rangle$ and push the corresponding new triplet into the priority queue if necessary. Following this, we compute all SRTs when the priority queue is empty. A detailed pseudocode of our final algorithm will be given in the following section.

4.3.2 Efficient CRT computation

We reduce the CRT checks by making use of the transitive property of the dominance relationship. The following lemma provides an early termination condition in the search of SRT computation.

Lemma 8 *Given a reachability tuple $\langle u, v, t_s, t_e\rangle$ and a vertex w, for any reachability tuple $\langle u, v', t'_s, t'_e\rangle$, we have w covers $\langle u, v', t'_s, t'_e\rangle$ if (i) w covers $\langle u, v, t_s, t_e\rangle$, (ii) $[t_s, t_e] \subseteq [t'_s, t'_e]$, and (iii) v span-reaches v' in $[t'_s, t'_e]$.*

Given the i-th vertex u_i in $\mathcal{O}$, assume that we have detected a vertex v that u_i can reach in an interval $[t_s, t_e]$, and the corresponding tuple $\langle u_i, v, t_s, t_e \rangle$ has been covered. Based on Lemma 8, we immediately terminate any further exploration of v since all other vertices that are reachable from $\langle v, t_s, t_e \rangle$ must have been covered too. By adopting this pruning technique, we not only avoid a large number of CRT checks but also reduce the search scope in SRT computation. We give the pseudocode of the final algorithm for the index construction by combining two optimization techniques in Algorithm 3.

In Algorithm 3, we use a parameter ϑ to achieve a trade-off between the index size and the index coverage practically. ϑ represents the largest interval length of span-reachability query that TILL-Index can support. In most applications, users may be only interested in the span-reachability queries in a small-length interval. We will show the index size and its construction time under different ϑ selections in Sect. 7.

Lines 4–16 of Algorithm 3 compute all reachable vertices and corresponding intervals from u_i. As discussed in Section 4.3.1, we always pop a triplet $\langle v, t_s, t_e \rangle$ with the smallest value of $t_e - t_s$ in line 8. Based on Lemma 8, we check if the reachability tuple $\langle u_i, v, t_s, t_e \rangle$ has been covered in line 10. Here, $u_i \rightsquigarrow^{\mathcal{L}}_{[t_s,t_e]} v$ means the answer of the span-reachability query from u_i to v in $[t_s, t_e]$ is *true* according to the current TILL-Index $\mathcal{L}$ ($\mathcal{L}$ includes the in-label $\mathcal{L}_{in}$ and out-label $\mathcal{L}_{out}$ of every vertex). Note that $\mathcal{L}$ dynamically increases during the execution process of the algorithm. We omit this tuple and stop further exploration of it if it is covered by the previously computed index (line 10). Lemmas 7 and 8 guarantee that $\langle u_i, v, t_s, t_e \rangle$ must be an CRT, and we safely add u_i with corresponding interval to the in-labels of v in line 11. Lines 12–16 explore the out-neighbors of v. We omit the neighbor with higher rank in line 13 since their reachability tuples have been covered in previous iterations. We compute the updated reachability interval for each neighbor v' in line 14. We push the triplet into the priority queue in line 16 if the interval gap is not larger than the threshold ϑ.

Example 6 We give a running example of Algorithm 3. The default value of the parameter ϑ is $+\infty$. Given a graph $\mathcal{G}$ in Fig. 1 and an alphabetical order, assume that we have processed the first 4 vertices. We have $i = 5$ in line 3 and $u_i = v_5$ in line 4. The priority queue is initialized with one special element $\langle v_5, +\infty, -\infty \rangle$. We pop $\langle v_5, +\infty, -\infty \rangle$ in line 8 and scan out-neighbors of v_5 including $\langle v_3, 4 \rangle$, $\langle v_8, 1 \rangle$ and $\langle v_8, 4 \rangle$. We omit the out-neighbor $\langle v_3, 4 \rangle$ since $\mathcal{O}(v_3) < \mathcal{O}(v_5)$ in line 13, and push $\langle v_8, 1, 1 \rangle$ and $\langle v_8, 4, 4 \rangle$ into $\mathcal{Q}$. Assume the next popped triplet in line 8 is $\langle v_8, 1, 1 \rangle$. v_8 has only one out-neighbor $\langle v_4, 6 \rangle$ and we have $t'_s = 1, t'_e = 6$ in line 14. We push $\langle v_4, 1, 6 \rangle$ into $\mathcal{Q}$. In the next round, we pop $\langle v_8, 4, 4 \rangle$ and push $\langle v_4, 4, 6 \rangle$ into $\mathcal{Q}$. Now, $\mathcal{Q}$ contains two triplets, $\langle v_4, 4, 6 \rangle$ and $\langle v_4, 1, 6 \rangle$. We do not push any new triplet into $\mathcal{Q}$ in the following rounds since both $\langle v_4, 4, 6 \rangle$ and $\langle v_4, 1, 6 \rangle$ are covered by v_3, and the condition in line 10 holds. Till now, we have computed all CRTs of v_5 which start from v_5.

Algorithm 3: TILL-Construct*()

```
Input: a temporal graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, a vertex order $\mathcal{O}$ and a
       parameter $\vartheta$
Output: the TILL-Index of $\mathcal{G}$
1  foreach $u \in \mathcal{V}$ do
2      $\mathcal{L}_{in}(u), \mathcal{L}_{out}(u) \leftarrow \emptyset$;
3  for $1 \le i < n$ do
4      $u_i \leftarrow$ the $i$-th vertex in $\mathcal{O}$;
5      $\mathcal{Q} \leftarrow$ an empty priority queue;
6      $\mathcal{Q}.push(\langle u_i, +\infty, -\infty \rangle)$;
7      while $\mathcal{Q}$ is not empty do
8          $\langle v, t_s, t_e \rangle \leftarrow \mathcal{Q}.pop()$;
9          if $u_i \neq v$ then
10             if $u_i \rightsquigarrow^{\mathcal{L}}_{[t_s,t_e]} v$ then continue;
11             else $\mathcal{L}_{in}(v) \leftarrow \mathcal{L}_{in}(v) \cup \{\langle u_i, t_s, t_e \rangle\}$;
12         foreach $\langle v', t \rangle \in N_{out}(v)$ do
13             if $\mathcal{O}(v') \le \mathcal{O}(u_i)$ then continue;
14             $t'_s \leftarrow \min(t_s, t), t'_e \leftarrow \max(t_e, t)$;
15             if $t'_e - t'_s + 1 > \vartheta$ then continue;
16             else $\mathcal{Q}.push(\langle v', t'_s, t'_e \rangle)$;
17     repeat lines 6–16 to construct $\mathcal{L}_{out}$ of each vertex by toggling
       between the subscripts $in$ and $out$;
```

Let l be the number of all CRTs. Based on Lemmas 5 and 6, it is straightforward to see that l is also the number of all labels, and the index size is bounded by l. Let $l_q = \max_{u \in \mathcal{V}} \max(|\mathcal{L}_{in}(u)|, |\mathcal{L}_{out}(u)|)$ and d be the largest out-degree or in-degree of vertices in the graph, i.e., $d = \max_{u \in \mathcal{V}} \max(degr_{out}(u), degr_{in}(u))$. The time complexity of Algorithm 3 is given as follows.

Theorem 3 *The running time of Algorithm 3 is bounded by* $O(ld(\log ld + l_q))$.

Proof We first focus on one iteration of line 3. Based on Lemmas 5 and 6, line 11 is performed $O(l)$ times. We scan the out-neighbors of v' if line 11 holds. Therefore, lines 13–16 are performed $O(l \cdot d)$ times, and the total number of items appended to the priority queue is bounded by $O(l \cdot d)$. In line 10, we check whether $\langle u_i, v, t_s, t_e \rangle$ is covered by prior vertices. This can be done by sequentially scanning the existing out-label of u_i and in-label of v and returning true if there is a common vertex in the interval $[t_s, t_e]$. The running time can be bounded by $O(|\mathcal{L}_{out}(u_i)| + |\mathcal{L}_{in}(v)|)$ or $O(l_q)$. In line 8 and 16, it requires $O(\log l \cdot d)$ to push a new item or pop the top item in the priority queue. By combining the results, we have the total time complexity $O(ld(\log ld + l_q))$. □

Undirected graphs. In undirected graphs, we only need to maintain one label set for each vertex. Therefore, we omit line 17 of Algorithm 3 when constructing the index of an undirected graph.

5 Query processing

We study the index-based query processing strategies in this section. We discuss the algorithm to answer the span-reachability query followed by a full discourse of the algorithm for the θ-reachability query.

5.1 Span-reachability query processing

Our first step is to present several basic pruning strategies to check span-reachability. Given a vertex u, let $t_{min}(N_{out}(u))$ (resp. $t_{max}(N_{out}(u))$) be the smallest (resp. largest) timestamp in out-neighbors of u. $t_{min}(N_{in}(u))$ and $t_{max}(N_{in}(u))$ are defined similarly. We have the following lemmas.

Lemma 9 *A vertex u span-reaches a vertex v in $[t_1, t_2]$ only if there exist a neighbor $\langle w, t\rangle \in N_{out}(u)$ and $\langle w', t'\rangle \in N_{in}(v)$ such that $t \in [t_1, t_2]$ and $t' \in [t_1, t_2]$.*

Lemma 10 *A vertex u span-reaches a vertex v in $[t_1, t_2]$ only if $t_2 \geq \max(t_{min}(N_{out}(u)), t_{min}(N_{in}(v)))$ and $t_1 \leq \min(t_{max}(N_{out}(u)), t_{max}(N_{in}(v)))$.*

We can check the conditions in above two lemmas simply by scanning the neighbors of each query vertex. If the conditions do not hold, we immediately return *false* and do not invoke any query processing procedure.

Given a pair of query vertices u, v and an interval $[t_s, t_e]$, a straightforward method to answer the span-reachability of u and v is to scan $\mathcal{L}_{out}(u)$ and $\mathcal{L}_{in}(v)$. Let $\mathcal{L}_{out}(u)_{[t_s,t_e]}$ (resp. $\mathcal{L}_{in}(u)_{[t_s,t_e]}$) be the set of all triplets in $\mathcal{L}_{out}(u)$ (resp. $\mathcal{L}_{in}(u)$) falling in the interval $[t_s, t_e]$. We answer *true* if there exists a common vertex in $\mathcal{L}_{out}(u)_{[t_s,t_e]} \cup \{u\}$ and $\mathcal{L}_{in}(v)_{[t_s,t_e]} \cup \{v\}$. Otherwise, we return *false*. This can be done by using a hash table to preserve the vertices.

To improve the query efficiency, we group the triplets in the out-label or in-label of each vertex by their target vertices (the first item in the triplet). Let $\mathcal{V}(\mathcal{L}_{out}(u))$ be the set of vertices in the reachability triplet of $\mathcal{L}_{out}(u)$, i.e., $\mathcal{V}(\mathcal{L}_{out}(u)) = \{v \in \mathcal{V} | \langle v, t_s, t_e\rangle \in \mathcal{L}_{out}(u)\}$. Given a vertex w in $\mathcal{V}(\mathcal{L}_{out}(u))$, we use $\mathcal{L}_{out}(u)_w$ to denote the intervals that u can reach w in $\mathcal{L}_{out}(u)$, i.e., $\mathcal{L}_{out}(u)_w = \{[t_s, t_e] | \langle w, t_s, t_e\rangle \in \mathcal{L}_{out}(u)\}$. We check the span-reachability in two phases. In the first one, we check if there exists a common vertex in $\{u\} \cup \mathcal{V}(\mathcal{L}_{out}(u))$ and $\{v\} \cup \mathcal{V}(\mathcal{L}_{in}(v))$. This can be done in a merge sort like strategy by arranging the vertices in the label of each vertex by their ranks. Once finding a common vertex w, we further check if there exist intervals falling in the query interval in $\mathcal{L}_{out}(u)_w$ and $\mathcal{L}_{in}(v)_w$, respectively. If yes, we immediately return *true*. Otherwise, we resume the search and look for the next common vertex. Recall that in Algorithm 3, the triplets appended to the out-label or in-label of each vertex follow the order of the vertex rank. Therefore, the group operation can be done naturally in the index construction without incurring extra cost.

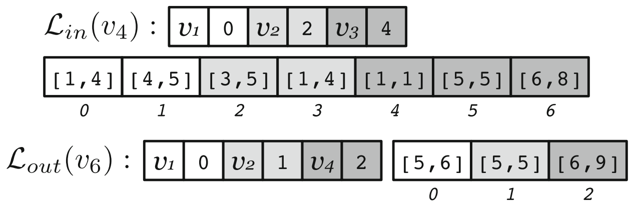

Fig. 3 The data structure of $\mathcal{L}_{in}(v_4)$ and $\mathcal{L}_{out}(v_6)$

To check whether there exists an interval falling in the query interval, we sort the intervals of each vertex in chronological order. So, given two intervals $[t_s, t_e]$ and $[t'_s, t'_e]$, $[t_s, t_e]$ is prior to $[t'_s, t'_e]$ if (i) $t_s < t'_s$, or (ii) $t_s = t'_s \wedge t_e < t'_e$. Therefore, given a query interval $[t_1, t_2]$ and an arbitrary interval $[t_s, t_e]$, if an interval $[t^*_s, t^*_e] \subseteq [t_1, t_2]$ exists, $[t^*_s, t^*_e]$ must appear after $[t_s, t_e]$ if $t_s < t_1$ or appear before $[t_s, t_e]$ if $t_e > t_2$. This sorting task can be done at the end of Algorithm 3 after all labels are completely computed, which would not increase the total time complexity of Theorem 3.

Example 7 Fig. 3 shows the data structure used to store the labels of each vertex. We take $\mathcal{L}_{in}(v_4)$ and $\mathcal{L}_{out}(v_6)$ as examples. All triplets in these two label sets can be found in Table 1. Two arrays are used to store the triplets in the label of each vertex. One interval array stores the intervals for each vertex in the label, and the other vertex array stores all vertices in the label and the start position of their intervals in the interval array. For $\mathcal{L}_{in}(v_4)$ in Fig. 3, the intervals of v_1, v_2 and v_3 are marked by white, light gray and dark gray, respectively. The intervals of v_2 in $\mathcal{L}_{in}(v_4)$ in the interval array start from the position of v_2 (i.e., 2) and end at the position of the next vertex v_3 in the vertex array (i.e., 4).

A complete pseudocode to process the span-reachability query is presented in Algorithm 4 which is self-explanatory. In lines 5, 6 and 9, we use the binary search method described above to find a subinterval of $[t_1, t_2]$. We provide a running example as follows.

Example 8 Assume that we aim to answer the span-reachability from v_6 to v_4 in $[3, 5]$. We scan the vertex array of $\mathcal{L}_{out}(v_6)$ and $\mathcal{L}_{in}(v_4)$ to look for a common vertex. We first find a common vertex v_1. However, there does not exist a subinterval of $[3, 5]$ of v_1 in the interval array of $\mathcal{L}_{out}(v_6)$. We continue to search the next common vertex and find v_2. We find there exists a subinterval $[5, 5]$ of v_2 in $\mathcal{L}_{out}(v_6)$ and a subinterval $[3, 5]$ of v_2 in $\mathcal{L}_{in}(v_4)$. Therefore, we return *true* for this query.

Algorithm 4: Span-Reach()

```
Input: TILL-Index of $\mathcal{G}$, two vertices $u$ and $v$, and an interval $[t_1, t_2]$
Output: the span-reachability of $u$ and $v$ in $[t_1, t_2]$
1  $i, i' \leftarrow 1$;
2  while $i \le |\mathcal{V}(\mathcal{L}_{out}(u))| \wedge i' \le |\mathcal{V}(\mathcal{L}_{in}(v))|$ do
3      $w \leftarrow$ the $i$-th vertex in $\mathcal{V}(\mathcal{L}_{out}(u))$;
4      $w' \leftarrow$ the $i'$-th vertex in $\mathcal{V}(\mathcal{L}_{in}(v))$;
5      if $w = v \wedge \exists [t_s, t_e] \in \mathcal{L}_{out}(u)_w : [t_s, t_e] \subseteq [t_1, t_2]$ then return true;
6      else if $w' = u \wedge \exists [t'_s, t'_e] \in \mathcal{L}_{in}(v)_{w'} : [t'_s, t'_e] \subseteq [t_1, t_2]$ then return true;
7      else if $\mathcal{O}(w) < \mathcal{O}(w')$ then $i \leftarrow i + 1$;
8      else if $\mathcal{O}(w) > \mathcal{O}(w')$ then $i' \leftarrow i' + 1$;
9      else if $\exists [t_s, t_e] \in \mathcal{L}_{out}(u)_w : [t_s, t_e] \subseteq [t_1, t_2] \wedge$
       $\exists [t'_s, t'_e] \in \mathcal{L}_{in}(v)_{w'} : [t'_s, t'_e] \subseteq [t_1, t_2]$ then
10         return true;
11     else $i \leftarrow i + 1, i' \leftarrow i' + 1$;
12 return false;
```

Theorem 4 *Given two query vertices u and v, the running time of Algorithm 4 is bounded by* $O(|\mathcal{L}_{out}(u)| + |\mathcal{L}_{in}(v)|)$.

5.2 θ-Reachability

Based on the idea for the span-reachability query processing, we study the θ-reachability query in this subsection. Given two vertices u, v, a threshold θ and an interval $[t_1, t_2]$, a straightforward idea to answer the θ-reachability query is to invoke Algorithm 4 for every possible interval (from $[t_1, t_1 + \theta - 1]$ to $[t_2 - \theta + 1, t_2]$). The time complexity of this method is $O((t_2 - t_1 - \theta) \cdot (|\mathcal{L}_{out}(u)| + |\mathcal{L}_{in}(v)|))$. We improve the time complexity to $O(|\mathcal{L}_{out}(u)| + |\mathcal{L}_{in}(v)|)$ by taking a sliding window based approach. Before discussing the details of the algorithm, we show that u θ-reaches v in $[t_1, t_2]$ if one of the following equations holds:

1. $\exists \langle v, t_s, t_e \rangle \in \mathcal{L}_{out}(u)$: $[t_s, t_e] \subseteq [t_1, t_2] \wedge t_e - t_s + 1 \le \theta$;
2. $\exists \langle u, t_s, t_e \rangle \in \mathcal{L}_{in}(v)$: $[t_s, t_e] \subseteq [t_1, t_2] \wedge t_e - t_s + 1 \le \theta$;
3. $\exists \langle w, t_s, t_e \rangle \in \mathcal{L}_{out}(u)$, $\langle w', t'_s, t'_e \rangle \in \mathcal{L}_{in}(v)$: $w = w' \wedge [t_s, t_e] \subseteq [t_1, t_2] \wedge [t'_s, t'_e] \subseteq [t_1, t_2] \wedge \max(t_e, t'_e) - \min(t_s, t'_s) + 1 \le \theta$.

Based on the conditions above, we can follow the same framework of Algorithm 4. We add the limitation $t_e - t_s + 1 \le \theta$ in line 5 and line 6 of Algorithm 4, respectively, to check the first two conditions. To check the third condition of finding a common vertex w in $\mathcal{V}(\mathcal{L}_{out}(u))$ and $\mathcal{V}(\mathcal{L}_{in}(v))$, we first filter out all intervals in $\mathcal{L}_{out}(u)_w$ and $\mathcal{L}_{in}(v)_w$ not found in $[t_1, t_2]$. With the concept of sliding window, the window is always θ. Recall that the intervals in each label are sorted in chronological order. The initial start time of the window is the smallest start time of the remaining intervals in the labels. If both the first intervals of two labels fall in the sliding window,

Algorithm 5: ES-Reach*()

```
Input: TILL-Index of $\mathcal{G}$, a parameter $\theta$, two vertices $u$ and $v$ and an interval $[t_1, t_2]$
Output: the $\theta$-reachability of $u$ and $v$ in $[t_1, t_2]$
1  $i, i' \leftarrow 1$;
2  while $i \le |\mathcal{V}(\mathcal{L}_{out}(u))| \wedge i' \le |\mathcal{V}(\mathcal{L}_{in}(v))|$ do
3      $w \leftarrow$ the $i$-th vertex in $\mathcal{V}(\mathcal{L}_{out}(u))$;
4      $w' \leftarrow$ the $i'$-th vertex in $\mathcal{V}(\mathcal{L}_{in}(v))$;
5      if
       $w = v \wedge \exists [t_s, t_e] \in \mathcal{L}_{out}(u)_w : [t_s, t_e] \subseteq [t_1, t_2], t_e - t_s + 1 \le \theta$
       then return true;
6      else if $w' = u \wedge \exists \langle w', t'_s, t'_e \rangle \in \mathcal{L}_{in}(v) : [t'_s, t'_e] \subseteq$
       $[t_1, t_2], t'_e - t'_s + 1 \le \theta$ then return true;
7      else if $\mathcal{O}(w) < \mathcal{O}(w')$ then $i \leftarrow i + 1$;
8      else if $\mathcal{O}(w) > \mathcal{O}(w')$ then $i' \leftarrow i' + 1$;
9      else if $\exists [t_s, t_e] \in \mathcal{L}_{out}(u)_w : [t_s, t_e] \subseteq [t_1, t_2] \wedge$
       $\exists [t'_s, t'_e] \in \mathcal{L}_{in}(v)_{w'} : [t'_s, t'_e] \subseteq [t_1, t_2]$ then
10         $k \leftarrow$ the position of the first interval $[t_s, t_e] \in \mathcal{L}_{out}(u)_w$
           s.t. $[t_s, t_e] \subseteq [t_1, t_2]$;
11         $k' \leftarrow$ the position of the first interval $[t'_s, t'_e] \in \mathcal{L}_{in}(v)_{w'}$
           s.t. $[t'_s, t'_e] \subseteq [t_1, t_2]$;
12         while $k \le |\mathcal{L}_{out}(u)_w| \wedge k' \le |\mathcal{L}_{in}(v)_{w'}|$ do
13             $[t_s, t_e]$ the $k$-th interval in $\mathcal{L}_{out}(u)_w$;
14             $[t'_s, t'_e]$ the $k'$-th interval in $\mathcal{L}_{in}(v)_{w'}$;
15             if $[t_s, t_e] \nsubseteq [t_1, t_2] \vee [t'_s, t'_e] \nsubseteq [t_1, t_2]$ then
16                 break;
17             else if $\max(t_e, t'_e) - \min(t_s, t'_s) + 1 \le \theta$ then
18                 return true;
19             else if $t_e - t_s + 1 > \theta \vee t_s < t'_s$ then
20                 $k \leftarrow k + 1$;
21             else $k' \leftarrow k' + 1$;
22         $i \leftarrow i + 1, i' \leftarrow i' + 1$;
23     else $i \leftarrow i + 1, i' \leftarrow i' + 1$;
24 return false;
```

we return *true*. Alternatively, we filter out the interval with the smallest start time and move the sliding window forward to the next smallest start time of the intervals. This step is repeated until no interval remains.

The pseudocode to answer the θ-reachability query is given in Algorithm 5. Lines 5 and 6 correspond to the θ-reachability conditions 1 and 2, respectively. Lines 9–22 correspond to condition 3. In lines 10 and 11, we use a binary search to locate the first interval falling in $[t_1, t_2]$. The condition of line 15 holds if all intervals of $\mathcal{L}_{out}(u)_w$ (or $\mathcal{L}_{in}(v)_w$) in $[t_1, t_2]$ are scanned, and we break the loop. Line 17 holds if we find a pair of intervals falling in the same sliding window. In lines 19 and 21, we move the sliding window with a new start time of $\min(t_s, t'_s)$.

Theorem 5 *Given a pair of vertices u and v, the running time of Algorithm 5 is bounded by* $O(|\mathcal{L}_{out}(u)| + |\mathcal{L}_{in}(v)|)$.

Example 9 Given a query interval $[1, 8]$ and $\theta = 3$, assume that we aim to answer 3-reachability from v_6 to v_4. The out-label and in-label of v_6 and v_4 are given in Fig. 3, respectively. In line 9 of Algorithm 5, we find a common vertex v_1 in

$\mathcal{V}(\mathcal{L}_{out}(v_6))$ and $\mathcal{V}(\mathcal{L}_{in}(v_4))$. We have $[t_s, t_e] = [5, 6]$ in line 13 and $[t'_s, t'_e] = [1, 4]$ in line 14. The conditions in lines 15, 17 and 19 do not hold. As a result, line 21 is executed. In the next iteration, we have $[t'_s, t'_e] = [4, 5]$ and $[t_s, t_e]$ is kept constant. The condition in line 17 holds, and *true* is returned.

6 TILL-Index maintenance

Many real-world temporal graphs incrementally and continuously update as edge streams. In this section, we extend the priority queue-based search technique in Sect. 4.3.1 to maintain TILL-Index in dynamic temporal streams. Section 6.1 investigates the problem of incremental TILL-Index maintenance given a set of new edges. Section 6.2 provides a method to prune TILL-Index for expiring edges.

The problem of maintaining hop-labeling-based index for shortest distance queries in unlabeled simple graphs has been studied in an existing work [4]. Unlike [4], our techniques for TILL-Index maintenance are around relationships between time intervals in the index and are extended from the priority-queue-based search proposed in Sect. 4.3.1. In addition, [4] gives up the support of deleting outdated label entries due to the poor efficiency. However, in the context of temporal graphs, we will show in Sect. 6.2 that the outdated labels can be pruned efficiently and the updated index is guaranteed to be minimal. Following [4], we assume the vertex order is fixed when edges update. Note that we only consider edge updates in the paper. This is because insertions or deletions of vertices can be expressed using a set of edge updates.

6.1 Incremental index maintenance

Let t_{max} be the latest time in the current temporal graph $\mathcal{G}$. Given a set of new edges with incurring times later than t_{max} inserted to $\mathcal{G}$, we aim to update the TILL-Index to support the queries for the latest time. The main technical challenges in designing algorithms for maintaining TILL-Index are to guarantee its completeness and minimality. For ease of presentation, we first assume that the time of each edge is unique unless otherwise stated. The assumption is crucial to guarantee the minimality. We will lift the restriction later and discuss the case that multiple new edges come associated with the same time.

Assume that an edge $\langle u_t, v_t, t\rangle$ is inserted with $t > t_{max}$. Due to the equivalence of CRTs and TILL-Index, the basic idea of TILL-Index maintenance is to monitor the changes of CRTs after inserting $\langle u_t, v_t, t\rangle$.

Lemma 11 *Let $\mathcal{T}$ and $\mathcal{T}^+$ be the set of all CRTs before and after the insertion of $\langle u_t, v_t, t\rangle$, respectively. We have $\mathcal{T} \subset \mathcal{T}^+$.*

Proof Given that the times of all edges are earlier than t, the insertion of $\langle u_t, v_t, t\rangle$ must generate at least one new CRT ending at t. Therefore, we have $\mathcal{T} \neq \mathcal{T}^+$. Next, we prove $\mathcal{T} \subseteq \mathcal{T}^+$. Assume that there exists a CRT $\langle u, v, t_s, t_e\rangle$ in $\mathcal{T}$ and not in $\mathcal{T}^+$. $\langle u, v, t_s, t_e\rangle$ must be dominated by a new CRT in $\mathcal{T}^+ \setminus \mathcal{T}$. This contradicts that any new CRT must end at t with $t > t_e$. □

Based on Lemma 11, all existing CRTs are still in the updated index, and we only need to find all new CRTs produced by the insertion of $\langle u_t, v_t, t\rangle$. Then, we update the index accordingly. Recall that given a new CRT $\langle u, v, t_s, t_e\rangle$, $\langle u, t_s, t_e\rangle$ is added to the in-label of v if $\mathcal{O}(u) < \mathcal{O}(v)$. Otherwise, $\langle v, t_s, t_e\rangle$ is added to the out-label of u. For simplicity, we mainly discuss computing all new CRTs $\langle u, v, t_s, t_e\rangle$ with $\mathcal{O}(u) < \mathcal{O}(v)$ and completing in-labels of each vertex. The idea for updating out-labels is similar.

Let $\langle u, v, t_s, t_e\rangle$ be an arbitrary new CRT generated by the insertion of $\langle u_t, v_t, t\rangle$. We immediately have $t_e = t$, which can be easily proved based on the definition of CRT. It is also straightforward to derive that $u \rightsquigarrow_{[t_s,t_e]} u_t$ and $v_t \rightsquigarrow_{[t_s,t_e]} v$. Intuitively, the search space to find all new CRTs can be very large since there may exist many vertices reaching u_t and reached from v_t. We refine it by making use the existing TILL-Index, which is shown in the following lemma.

Lemma 12 *Given a new edge $\langle u_t, v_t, t\rangle$ and an arbitrary new CRT $\langle u, v, t_s, t_e\rangle$ with $\mathcal{O}(u) < \mathcal{O}(v)$, we have $\mathcal{O}(u) < \mathcal{O}(v_t)$ and $u \in \{u_t\} \cup \mathcal{L}_{in}(u_t)$.*

Proof Given the new CRT $\langle u, v, t_s, t_e\rangle$, there exists a path from u to v over the interval $[t_s, t_e]$. The path is via $\langle u_t, v_t, t\rangle$, and $t_e = t$. Given $\mathcal{O}(u) < \mathcal{O}(v)$, the rank of u is the highest in the path. The path from u to u_t corresponds to a CRT. Given the equivalence of CRTs and the labeling index, we have $u \in \{u_t\} \cup \mathcal{L}_{in}(u_t)$. □

Based on Lemma 12, we compute all new CRTs starting from every vertex in $\{u_t\} \cup \mathcal{L}_{in}(u_t)$. Recall that in Algorithm 3, we compute CRTs by performing a priority queue-based search from each root vertex. Given a root vertex $u \in \{u_t\} \cup \mathcal{L}_{in}(u_t)$, instead of searching from scratch, we can reuse the intermediate searching result from u to u_t and resume the search from u_t to all new vertices reached by u. This is because any valid path of the new CRT $\langle u, v, t_s, t_e\rangle$ must pass (u_t, v_t). We show the following two lemmas to support our detailed algorithms.

Lemma 13 *Let $\mathcal{T}_{u,*}$ and $\mathcal{T}_{*,u}$ be the sets of CRTs reached from u and reaching u, respectively. $\mathcal{T}^+_{u,*}$ and $\mathcal{T}^+_{*,u}$ be their counterparts after inserting $\langle u_t, v_t, t\rangle$. We have $\mathcal{T}_{*,u_t} = \mathcal{T}^+_{*,u_t}$ and $\mathcal{T}_{v_t,*} = \mathcal{T}^+_{v_t,*}$.*

Lemma 14 *Given a new edge $\langle u_t, v_t, t\rangle$ and an arbitrary new CRT $\langle u, v, t_s, t_e\rangle$ with $\mathcal{O}(u) < \mathcal{O}(v)$ and $u \neq u_t$, let $[t'_s, t'_e]$*

be the last (latest) interval in $\mathcal{L}_{in}(u_t)_u$. *We have* $[t'_s, t'_e] \subseteq [t_s, t_e]$.

Based on Lemma 13, we can safely use the existing CRTs from a root vertex u to u_t. Based on Lemma 14, we resume the search of u from only one tuple, which is $\langle u_t, t'_s, t'_e \rangle$. Searching from other tuples from u to u_t maintained in $\mathcal{L}_{in}(u_t)_u$ would produce results dominated by those from $\langle u_t, t'_s, t'_e \rangle$.

Algorithm 6: TILL-Insert()

```
Input: TILL-Index of $\mathcal{G}$, a parameter $\theta$, a vertex order $\mathcal{O}$ and a
       new edge $\langle u_t, v_t, t \rangle$
Output: the updated index
1  $T_{in}, T_{out} \leftarrow \emptyset$;
2  foreach $u \in \mathcal{V}(\mathcal{L}_{in}(u_t))$ do
3      if $\mathcal{O}(u) \geq \mathcal{O}(v_t)$ then break;
4      $[t_s, t_e] \leftarrow$ the last interval in $\mathcal{L}_{in}(u_t)_u$;
5      if $t - t_s + 1 > \vartheta$ then continue;
6      $T_{in} \leftarrow T_{in} \cup \{\langle u, v_t, t_s, t \rangle\}$;
7  foreach $v \in \mathcal{V}(\mathcal{L}_{out}(v_t))$ do
8      if $\mathcal{O}(v) \geq \mathcal{O}(u_t)$ then break;
9      $[t_s, t_e] \leftarrow$ the last interval in $\mathcal{L}_{out}(v_t)_v$;
10     if $t - t_s + 1 > \vartheta$ then continue;
11     $T_{out} \leftarrow T_{out} \cup \{\langle u_t, v, t_s, t \rangle\}$;
12 $T \leftarrow$ perform a binary merge sort on $T_{in}$ and $T_{out}$;
13 add $\langle u_t, v_t, t, t \rangle$ to the end of $T$;
14 foreach $\langle u, v, t_s, t_e \rangle \in T$ do
15     $\mathcal{Q} \leftarrow$ an empty priority queue;
16     if $\mathcal{O}(u) < \mathcal{O}(v)$ then
           // search following edge directions
              and complete in-labels
17         $\mathcal{Q}.push(\langle v, t_s, t_e \rangle)$;
18         perform lines 7–16 in Algorithm 3 by replacing $u_i$ with $u$;
19     else
           // search following reverse edge
              directions and add out-labels
20         $\mathcal{Q}.push(\langle u, t_s, t_e \rangle)$;
21         perform line 17 in Algorithm 3 by replacing $u_i$ with $v$;
```

The Algorithm. We now present the algorithm to incrementally maintain TILL-Index in Algorithm 6. Lines 1–13 prepare all CRTs as initial states of the priority-queue based search (lines 14–21). Lines 2–6 prepare CRTs to complete in-labels of the index. Based on Lemma 12, we only consider the vertices that can reach u_t and have lower ranking values (line 2). Note that vertices in $\mathcal{V}(\mathcal{L}_{in}(u_t))$ have been arranged following the total order. As a result, we terminate the iteration once finding a vertex ranking lower than v_t in line 3. Based on Lemma 14, we derive the last time interval from u to u_t. We explore the interval via the edge $\langle u_t, v_t, t \rangle$ and generate the CRT $\langle u, v_t, t_s, t \rangle$ since $t > t_e$. Similarly, lines 7–11 prepare CRTs for out-labels.

Lines 12–13 organize all reachability tuples in non-decreasing order of their smallest vertex ranking values, which is crucial to guarantee the index minimality and improve the updating efficiency. We simply perform a merge sort in line 12 since tuples in T_{in} and T_{out} are sorted, respectively, during the construction. In addition, ranks of u in line 6 and v in line 11 are higher than that any of u_t and v_t, which supports us to simply add $\langle u_t, v_t, t, t \rangle$ to the end of T in line 13. Lines 14–21 resume the priority queue-based search from each given starting reachability tuple, which is self-explanatory. The completeness and minimality of the index can be easily derived based on the lemmas in Sect. 6.1, and we omit the detailed proofs.

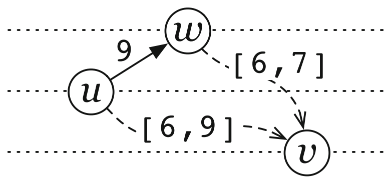

Fig. 4 An example of single edge insertion

We analyze the running time of Algorithm 6 below where l^{new} is the number of all new labels. The definitions of l_q and d are the same as those in Theorem 3.

Theorem 6 *The running time of Algorithm 6 is bounded by* $O(l^{new} d(\log l^{new} d + l_q))$.

Proof The proof is similar to that of Theorem 3, and we omit the details. □

Note that Algorithm 6 can be extended to handle out-of-order edge insertions (i.e., the time of the new edge is not the largest). Let t be the time of the new edge. Instead of picking the last interval in line 4, we derive the union of $[t, t]$ and each interval in $\mathcal{L}_{in}(u_t)_u$ and add them to T_{in} (line 6). We revise lines 9–11 similarly. The revised algorithm computes the complete index to answer any possible query, but the index may not be minimal for out-of-order insertions.

Handling simultaneous edges. When multiple edges come and are assigned by the same new time, iteratively processing each edge by invoking Algorithm 6 still works but probably cannot guarantee the index minimality. We consider an example shown in Fig. 4. The ranks of three vertices are $\mathcal{O}(w) < \mathcal{O}(u) < \mathcal{O}(v)$. Before the insertion of $\langle u, w, 9 \rangle$, we have $\langle w, 6, 7 \rangle \in \mathcal{L}_{in}(v)$, $\langle u, 6, 9 \rangle \in \mathcal{L}_{in}(v)$, and u cannot reach w. Note that $\langle u, 6, 9 \rangle$ is added to the in-label of v due to the insertion of another edge at the time 9. When $\langle u, w, 9 \rangle$ is inserted, we add $\langle w, 9, 9 \rangle$ to the out-label of u by Algorithm 6. As a result, w covers the reachability tuple $\langle u, v, 6, 9 \rangle$, and $\langle u, 6, 9 \rangle$ in $\mathcal{L}_{in}(v)$ is redundant.

Even though such a case is not common in most datasets according to our performance studies, we extend Algorithm 6 to guarantee the index minimality theoretically. We make the following observation based on the example above.

Lemma 15 *Given a new CRT $\langle u, v, t_s, t_e\rangle$ by the insertion of an edge $\langle u_t, v_t, t\rangle$, $\langle u, v, t_s, t_e\rangle$ becomes redundant when inserting an edge $\langle u'_t, v'_t, t'\rangle$ with $t' \geq t$ only if $t' = t$.*

Lemma 15 reveals that the index redundancy only happens for the new edges coming at the same time. Therefore, instead of processing each single edge, we process all new edges coming at the same time together. Specifically, we first perform lines 1–13 of Algorithm 6 for every edge and merge reachability tuples of all edges as one sorted list T. Assume that u is the vertex with the highest rank in all tuples of T. Unlike single edge insertion, there may exist several tuples starting (or reaching) from u in T. Accordingly, we modify the phase of priority-queue based search (lines 14–21). Instead of only pushing one tuple to the priority queue $\mathcal{Q}$ (line 17 and line 20), we push all tuples starting from u to $\mathcal{Q}$ and perform line 18. We push all tuples ending to v to $\mathcal{Q}$ and perform line 21. In this way, each derived CRT can never be dominated in future iterations.

6.2 Edge deletion

In certain applications, the index may never need to support the query intervals starting earlier than a given time. We propose an algorithm to dynamically prune the index in this subsection. Let t_{min} be the earliest time of all edges in the temporal graph. We start by considering a case that removing all edges at t_{min}. The updating method is simple and efficient based on the following lemma.

Lemma 16 *Let $\mathcal{T}^-$ be the set of all CRTs not starting from t_{min}. $\mathcal{T}^-$ is exactly the set of all CRTs in the graph after deleting all edges at t_{min}.*

Proof Let $\mathcal{G}^-$ be the temporal graph after deleting all edges at t_{min}. Based on Definition 4, deleting an edge at t_{min} would not break any CRT starting after t_{min}. Therefore, every CRT in $\mathcal{T}^-$ is still valid in $\mathcal{G}^-$. Next we show the completeness of $\mathcal{T}^-$. Assume that there exists a CRT c of $\mathcal{G}^-$ not in $\mathcal{T}^-$. Given that $\mathcal{T}^-$ is the set of all CRTs not starting from t_{min}, c must be dominated by a tuple starting from t_{min}, which contradicts that c starts after t_{min}. The proof is finished. □

Based on Lemma 16, we only need to remove all labels containing t_{min} and finish updating the index. We give the pseudocode for the edge deletion in Algorithm 7. The parameter t represents the earliest supported time of the updated index. For the case of removing all edges at t_{min}, we set $t = t_{min} + 1$ in Algorithm 7. Note that in lines 3 and 7, intervals have been sorted chronologically. It is clear to see that the index returned by Algorithm 7 is still minimal. The time complexities for deleting edges are given as follows.

Lemma 17 *The running time to delete all edges at the earliest time t_{min} is bounded by $O(\sum_{u\in\mathcal{V}} |\mathcal{V}(\mathcal{L}_{in}(u))| + |\mathcal{V}(\mathcal{L}_{out}(u))|)$.*

Algorithm 7: TILL-Delete()

```
Input: TILL-Index of $\mathcal{G}$ and the earliest supported time $t$ of the
       index
Output: the updated index
1 foreach $u \in \mathcal{V}$ do
2     foreach $v \in \mathcal{V}(\mathcal{L}_{in}(u))$ do
3         foreach $[t_s, t_e] \in \mathcal{L}_{in}(u)_v$ do
4             if $t_s \geq t$ then break;
5             remove $[t_s, t_e]$;
6     foreach $v \in \mathcal{V}(\mathcal{L}_{out}(u))$ do
7         foreach $[t_s, t_e] \in \mathcal{L}_{out}(u)_v$ do
8             if $t_s \geq t$ then break;
9             remove $[t_s, t_e]$;
```

Table 2 Network statistics

Dataset	$\mathcal{M}$	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\vartheta_{\mathcal{G}}$
CollegeMsg	D	1899	59,835	16,736,181
Chess	D	7301	65,053	99
Slashdot	D	51,083	140,778	1,157,361,660
MathOverflow	D	24,818	506,500	203,068,736
Facebook_f	U	63,731	817,035	1,232,231,923
Epinions	D	131,828	841,372	944
Facebook_wp	D	46,952	876,993	134,873,285
AskUbuntu	D	159,316	964,437	225,834,442
Enron	D	87,273	1,148,072	1,401,187,797
SuperUser	D	194,085	1,443,339	239,614,928
Digg	D	279,630	1,731,653	1,247,032,805
Wiki	U	118,100	2,917,785	239,001,193
Prosper	D	89,269	3,394,979	2142
Arxiv	U	28,093	4,596,803	3649
Youtube	U	3,223,589	9,375,374	225
DBLP	U	1,314,050	18,986,618	76
Flickr	D	2,302,925	33,140,017	197
DBLP_p	U	2,828,689	156,773,140	10

Lemma 18 *The running time of Algorithm 7 for an arbitrary parameter t is bounded by $O(l)$, where l is the number of all labels.*

7 Experiments

We conducted extensive experiments to evaluate the performance of our proposed algorithms, summarized as follows:

– Online-Reach: Algorithm 1.
– Span-Reach: Algorithm 4.

– ES-Reach: a naive method to answer θ-reachability by invoking several runs of Span-Reach(). More details can be found in Sect. 5.2.
– ES-Reach*: Algorithm 5.
– TILL-Construct: A basic implementation of Algorithm 2. We use a queue to compute all SRTs and get CRTs by checking whether every SRT can be covered by existing labels. More details can be found in Sect. 4.1.
– TILL-Construct*: Algorithm 3.
– TILL-Insert: The algorithm for edge insertion.
– TILL-Delete: The algorithm for edge deletion.

All algorithms were implemented in C++ and compiled using a g++ compiler at a -O3 optimization level. All the experiments were conducted on a Linux Server with an Intel Xeon 2.7GHz CPU and 180GB RAM.

Datasets. We conducted experiments on eighteen real-world graphs. The detailed statistics of these datasets are summarized in Table 2. $\mathcal{M}$ demonstrates the types of datasets, where D represents the directed graph and U represents the undirected graph. $\vartheta_{\mathcal{G}}$ demonstrates the number of atomic units between the smallest timestamp and the largest timestamp. DBLP_p is a graph generated from the data of DBLP from 2011 to 2020. Each vertex is a publication. Two vertexes are connected by an edge if they have a common author. The time of the edge is the later publication year of two vertices. All other networks and corresponding detailed descriptions can be found in SNAP[1] and KONECT[2].

The rest of this section is organized as follows. Section 7.1 provides the performance of answering span-reachability queries. Section 7.2 evaluates the index construction algorithms. Section 7.3 reports the performance of answering θ-reachability queries. Section 7.4 reports the performance of index maintenance. Section 7.5 reports the performance for continuous query processing.

7.1 Span-reachability query processing

We evaluate the performance of span-reachability query processing. To generate input queries, we randomly pick 100 vertex pairs in each graph $\mathcal{G}$. For each vertex pair, we randomly generate subintervals of $[1, \vartheta_{\mathcal{G}}]$ and only keep intervals if the conditions in Lemmas 9 and 10 are satisfied. We repeat this step until 10 intervals are found. This strategy works because the query algorithm is only invoked if the conditions in Lemmas 9 and 10 hold. As a result, we fully prepare 1000 span-reachability queries. We report the running time of Span-Reach for such 1000 queries with Online-Reach as a comparison in Fig. 5. In addition to the overall query time, we report the average time of all cases

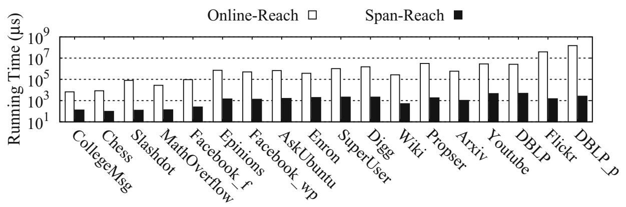

Fig. 5 Time of span-reachability query processing

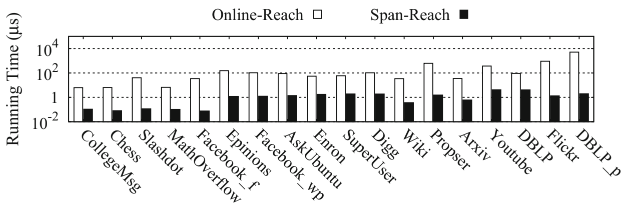

Fig. 6 Average time of span-reachability query processing (true cases)

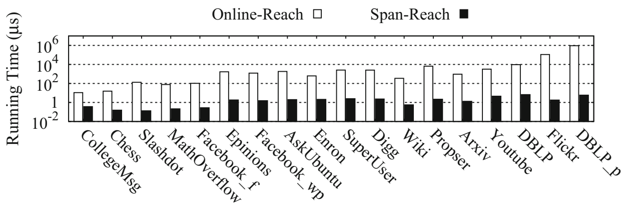

Fig. 7 Average time of span-reachability query processing (false cases)

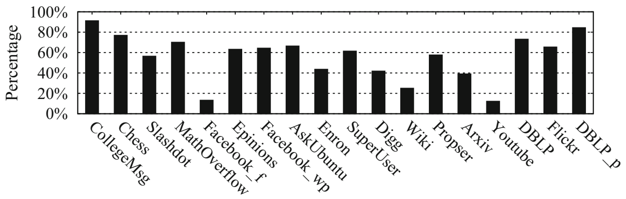

Fig. 8 Percentage of true cases in 1000 queries

that the result is true in Fig. 6 and report the average time for all false cases in Fig. 7. The percentage of true cases in 1000 queries for each dataset is reported in Fig. 8.

We can see that the running time of Span-Reach is at least two orders of magnitude smaller than that of Online-Reach in all datasets in the experiment. For example, in the largest dataset Flickr, Online-Reach takes over 30 seconds, while our Span-Reach algorithm takes only about 1.4 ms ($1\text{s} = 10^3\text{ms} = 10^6\mu\text{s}$). For each dataset, the average running time of true cases is smaller than that of false cases since we need to scan all labels if the result is false.

7.2 Index Construction

This section is devoted to evaluating the performance of index construction algorithms.

[1] http://snap.stanford.edu/data/index.html.
[2] http://konect.cc/.

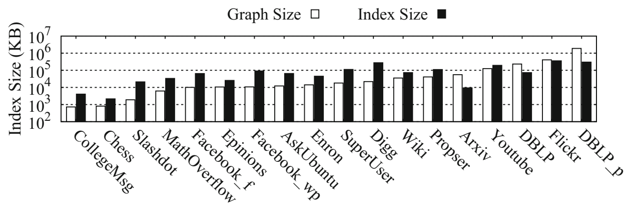

Fig. 9 Index size

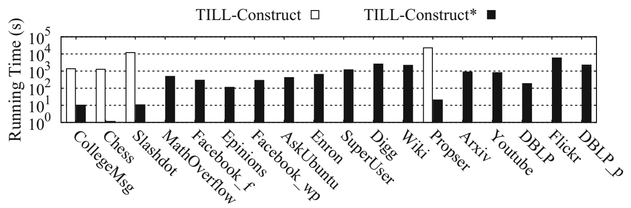

Fig. 10 Indexing Time

7.2.1 Index size

We report the index size of all datasets in Fig. 9 and also add the size of datasets as a comparison. We can find that in several large datasets, the index size is smaller than the graph size. For example, in Flickr, the dataset takes about 400 MB while the index takes only about 350 MB.

7.2.2 Indexing Time

The running time of TILL-Construct* for all datasets is reported with TILL-Construct as a comparison in Fig. 10.

Note that the running time of TILL-Construct on several datasets are not given as the algorithm cannot finish in twenty-four hours. It is clear that in comparing all reported times of TILL-Construct, TILL-Construct* is at least two orders of magnitude faster. For example in Flickr, TILL-Construct* takes about 1.5 hours to compute TILL-Index. TILL-Construct* takes about 1 second on Chess, which is the shortest on all reported times. By contrast, the running time of TILL-Construct on Chess is about 20 minutes.

7.2.3 Varying ϑ

The running times and index sizes of TILL-Construct* are presented in Fig. 11 by varying the input parameter ϑ from 20% to 100% of $\vartheta_{\mathcal{G}}$ for each dataset $\mathcal{G}$. Note that $\vartheta = \vartheta_{\mathcal{G}}$ is equivalent to the default setting ϑ as $+\infty$. Due to limited space here, Fig. 11 shows only the results of four datasets — Enron, Youtube, DBLP and Flickr. The results for other datasets display similar trends.

We can see from the figures(a)–(d) that the increasing speed of running time becomes small when both vertex and edge sampling ratio increases. For example, the running time of TILL-Construct* on Flickr is about 14 minutes when the

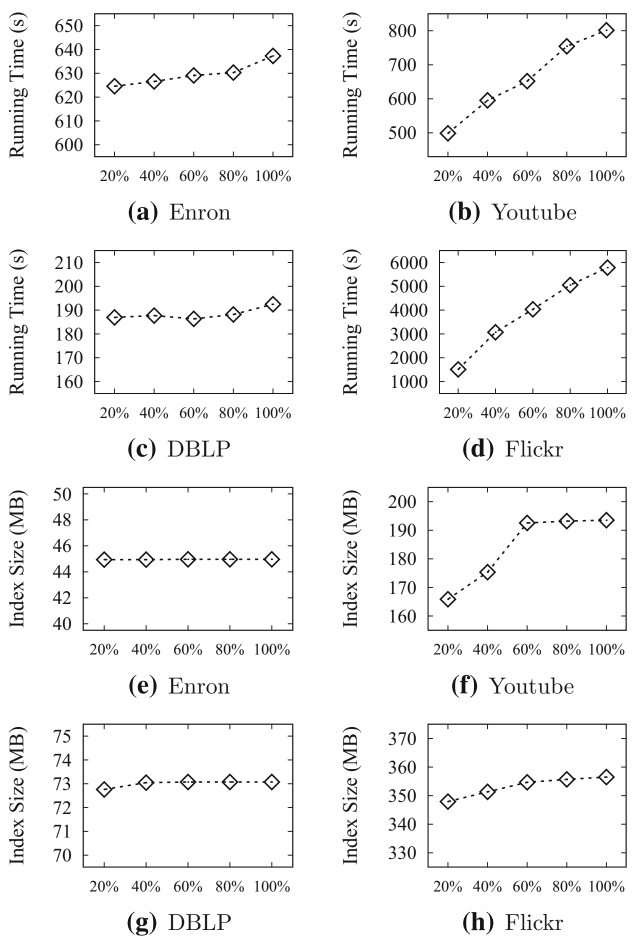

Fig. 11 Varying ϑ of TILL-Construct*

edge sampling ratio is 20%. It reaches to 22 minutes, 35 minutes and 73 minutes when the edge sampling ratio is 40%, 60% and 80%, respectively. Finally, on the ratio of 100%, the time reaches about 90 minutes. The increasing trends for the index size in figures(e)–(h) are similar and even more gentle.

Fig. 11(a)–(d) reports the running times. We can see that the increases on both Enron and DBLP are not obvious (does not exceed 20 seconds) from 20% to 100%. The lines are almost linear in Youtube and Flickr, which start from about 500 seconds and 25 minutes, ending at about 750 seconds and 1.5 hours, respectively. Fig. 11(e)–(h) reports the index size. The change on all reported datasets is very small. The group of figures shows that the index size and indexing time are confined even though we do not set any interval length limitation ($\vartheta = +\infty$) in TILL-Construct*.

7.2.4 Scalability

This experiment tests the scalability of our index construction algorithm, which is shown in Fig. 12. We only report results for four real-world graph datasets as representatives—Enron,

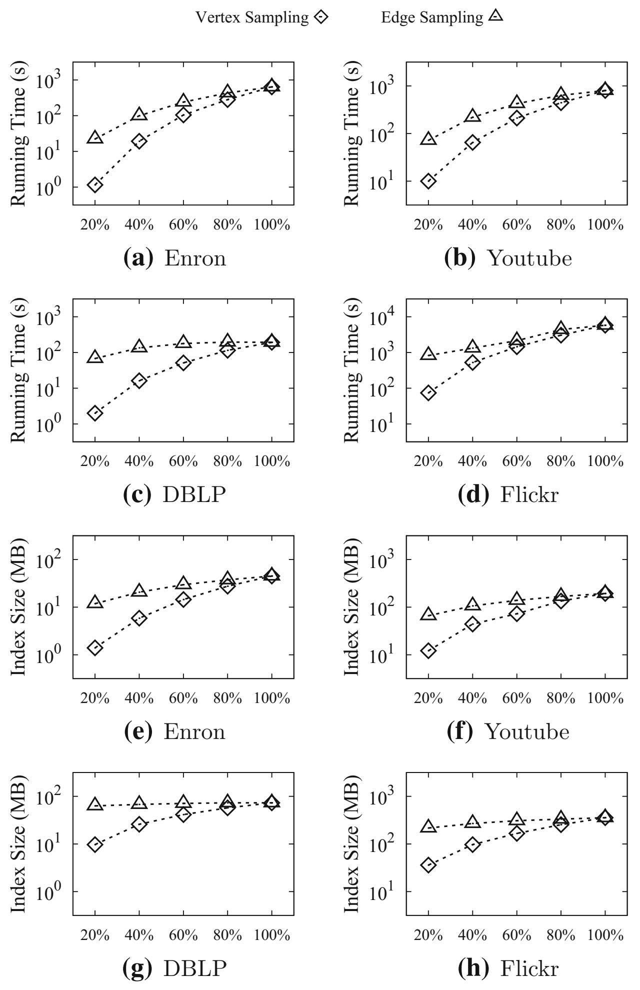

Fig. 12 Scalability of index construction

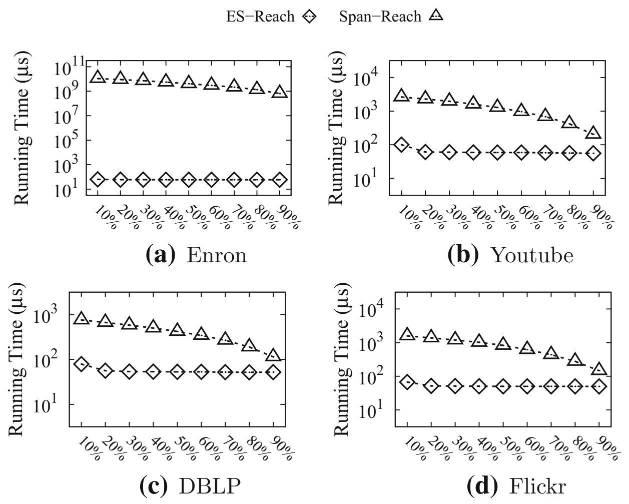

Fig. 13 Performance of θ-reachability query processing

Table 3 Performance of index maintenance (μs)

Dataset	Insertion	Deletion	Q_{online}	Q_{index}
CollegeMsg	126	0.14	6	0.12
Chess	10	0.08	8	0.09
Slashdot	67	0.58	80	0.12
MathOverflow	577	0.24	27	0.13
Facebook_f	6511	0.61	93	0.25
Epinions	604	0.02	715	1.42
Facebook_wp	288	0.41	505	1.33
AskUbuntu	306	0.32	675	1.60
Enron	384	0.16	374	1.94
SuperUser	417	0.39	1027	2.13
Digg	1701	0.70	1525	2.12
Wiki	3394	1.37	268	0.51
Prosper	9	0.10	3140	1.81
Arxiv	159	0.27	588	1.04
Youtube	2345	0.99	2849	4.54
DBLP	12	0.05	2613	4.81
Flickr	302	0.01	38,879	1.48
DBLP_p	10	0.01	150,414	2.59

Youtube, DBLP and Flickr. The results on other datasets show similar trends. For each dataset, we vary the graph size and graph density by randomly sampling vertices and edges from 20% to 100%. When sampling vertices, we derive the induced subgraph of the sampled vertices, and when sampling edges, we select the incident vertices of the edges as the vertex set.

7.3 θ-Reachability query processing

We evaluate the performance of θ-reachability query processing in this subsection. To prepare the input queries, we adopt the same strategy described in Sect. 7.1 and randomly pick 100 vertex pairs and 10 intervals for each vertex pair. For each interval, we set θ as a fraction of its length and adjust the fraction from 10% to 90%. The running time of ES-Reach* on four representative datasets is given in Fig. 13, with ES-Reach as a comparison.

We can see from Fig. 13 that ES-Reach* is faster than ES-Reach on all parameter settings. Their times trend towards equal when θ increases, since two algorithms are equivalent when θ is the length of the query interval. For the performance of ES-Reach*, it is clear that all lines present roughly downward trends.

7.4 Index maintenance

We report the practical performance of index maintenance algorithms. For the edge insertion, we pick the latest ten percent of all edges and insert them into the temporal graph of the front ninety percent. We record the average processing time for each edge. For the edge deletion, we pick the earliest

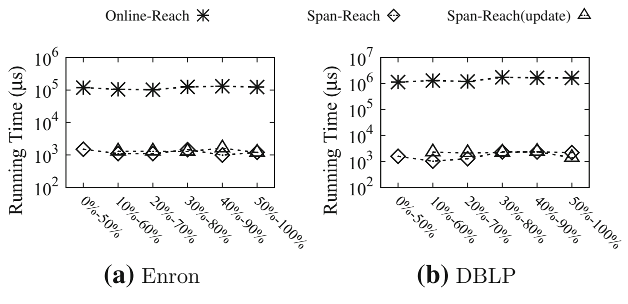

Fig. 14 Query time by sliding time window

ten percent of edges and delete them from the original temporal graph. Similarly, we record the average processing time. The results are shown in Table 3. As for comparisons, we also report the average times of an online span-reachability query and an index-based span-reachability query, respectively. We can see that the edge deletion is extremely fast due to its lightweight processing strategy. In most datasets, the processing time of an edge insertion is smaller than that of an online query. Given that the index-based query time is almost negligible compared with the online query time, the results support that our index-based solution still works well for dynamic temporal graphs.

7.5 Continuous query processing

We simulate a real scenario by continuously maintaining a time window for two representative datasets Enron and DBLP. For each dataset, we initially pick the first 50% edges and construct a temporal graph. Then, we slide the time window by adding 10% new edges and removing the oldest 10% edges each time.

Figure 14 reports the performance of query processing in different time windows. Span-Reach represents the query algorithm where the index is constructed from scratch for the current time window. Span-Reach (update) represents the query algorithm where the index is updated from previous windows. Note that vertices always follow the degree order of the initial window (0% – 50%) when updating the index. We can see that the query times of Span-Reach and Span-Reach (update) are almost the same in all windows.

Figures 15 and 16 report the indexing time and the index size, respectively, when sliding the time window. The index size is almost the same even though we use the same degree order of the original temporal graph. Note that the indexing time for the initial window (0% – 50%) is relatively large since more times are included in the first 50% edges.

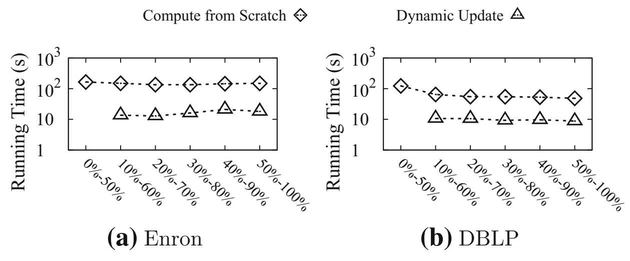

Fig. 15 Indexing time by sliding time window

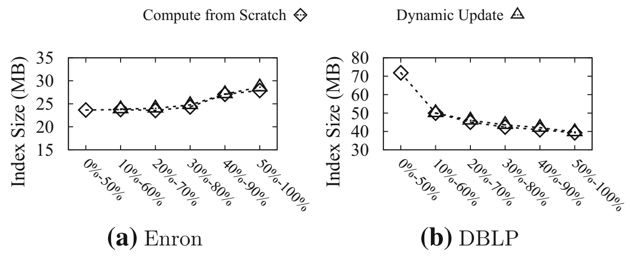

Fig. 16 Index size by sliding time window

8 Related works

Reachability in temporal graphs. The time-respecting path is defined in [21] to model the reachability problem in temporal graphs. The similar concept is also studied using the terms journey [14,34] or non-decreasing path [10]. Based on the time-respecting path, an index-based algorithm to efficiently answer the reachability problem in temporal graphs is studied in [33] and is improved in [39] for the distributed environment. The time-respecting model only requires a non-decreasing order of edge times in each valid path. Accordingly, each temporal graph can be transformed to a unlabeled directed graph without breaking the correctness of any time-respecting reachability query. However, the definition of the span-reachability model is totally different, and existing indexing techniques cannot be applied. The historical reachability problem is studied in [25]. Given an interval $[t_1, t_2]$ and a pair of vertices u, v, the conjunctive historical reachability of u, v is true if for each possible $t \in [t_1, t_2]$, there exists a path connecting u, v and all timestamps in the path are t. The disjunctive historical reachability of u, v is true if there exists a timestamp $t \in [t_1, t_2]$ and a path connecting u, v in which all timestamps in the path are t [25]. Other mining problems in temporal graphs can be found in surveys [8,18,23].

Reachability in static graphs and dynamic graphs. A large number of works have been done to design an index for answering the reachability query in static graphs [2,9,11,13,15,19,24,27,29,31,35,36]. These works only focus on the topological structure of graphs and ignore the temporal information. Distributed algorithms for reachability testing are also studied in [40]. Interested readers can find more details in surveys [6,38]. Several works study the index maintenance in dynamic graphs [7,24,37,41]. Estimating reachability based on random walks is studied in [26].

9 Conclusion

In this paper, we define a span-reachability model to capture entity relationships in a specific period of temporal graphs. We propose an index-based method based on the concept of two-hop cover to answer the span-reachability query for any pair of vertices and time intervals. Several optimizations are given to improve the efficiency of index construction. We also study the problem of θ-reachability, which is a generalized version of span-reachability. Index maintenance algorithms are proposed for dynamic temporal graphs. We conduct extensive experiments on eighteen real-world datasets to show the efficiency of our proposed algorithms.

Acknowledgements Ying Zhang is supported by ARC FT170100128 and ARC DP210101393. Lu Qin is supported by ARC FT200100787 and DP210101347. Dawei Cheng is supported by the National Science Foundation of China under grant no 62102287. Wenjie Zhang is supported by ARC DP180103096 and ARC DP200101116.

References

1. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: Hierarchical hub labelings for shortest paths. In: ESA, pages 24–35, (2012)
2. Agrawal, R., Borgida, A., Jagadish, H.V.: Efficient management of transitive relationships in large data and knowledge bases. SIGMOD **18**, 253–262 (1989)
3. Akiba, T., Iwata, Y., Yoshida, Y.: Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In: SIGMOD, pages 349–360, (2013)
4. Akiba, T., Iwata, Y., Yoshida, Y.: Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In: Chung, C., Broder, A.Z., Shim, K., Suel, T. (eds.) WWW, pp. 237–248. ACM (2014)
5. Anyanwu, K., Sheth, A.: ρ-queries: enabling querying for semantic associations on the semantic web. In: WWW, pages 690–699, (2003)
6. Bonifati, A., Fletcher, G., Voigt, H., Yakovets, N.: Querying graphs. Synth. Lect. Data Manag. **10**(3), 1–184 (2018)
7. Bramandia, R., Choi, B., Ng, W.K.: On incremental maintenance of 2-hop labeling of graphs. In: WWW, pages 845–854, (2008)
8. Casteigts, A., Flocchini, P., Quattrociocchi, W., Santoro, N.: Time-varying graphs and dynamic networks. Int. J. Parallel Emerg. Distrib. Syst. **27**(5), 387–408 (2012)
9. Chen, Y., Chen, Y.: An efficient algorithm for answering graph reachability queries. In: ICDE, pages 893–902, (2008)
10. Cheng, E., Grossman, J.W., Lipman, M.J.: Time-stamped graphs and their associated influence digraphs. Discrete Appl. Math. **128**(2–3), 317–335 (2003)
11. Cheng, J., Huang, S., Wu, H., Fu, A.W.-C.: Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In: SIGMOD, pages 193–204, (2013)
12. Chvatal, V.: A greedy heuristic for the set-covering problem. Math. Operations Res. **4**(3), 233–235 (1979)
13. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. SIAM J. Comput. **32**(5), 1338–1355 (2003)
14. Ferreira, A.: On models and algorithms for dynamic communication networks: The case for evolving graphs. In: In Proc. ALGOTEL, (2002)
15. Gao, Y., Zhang, T., Qiu, L., Linghu, Q., Chen, G.: Time-respecting flow graph pattern matching on temporal graphs. IEEE Trans. Knowl. Data Eng. **33**, 3453–3467 (2020)
16. Gurukar, S., Ranu, S., Ravindran, B.: Commit: A scalable approach to mining communication motifs from dynamic networks. In: SIGMOD, pages 475–489, (2015)
17. Holme, P., Edling, C.R., Liljeros, F.: Structure and time evolution of an internet dating community. Soc. Netw. **26**(2), 155–174 (2004)
18. Holme, P., Saramäki, J.: Temporal networks. Phys. Rep. **519**(3), 97–125 (2012)
19. Jin, R., Xiang, Y., Ruan, N., Fuhry, D.: 3-hop: a high-compression indexing scheme for reachability query. In: SIGMOD, pages 813–826, (2009)
20. Jin, R., Xiang, Y., Ruan, N., Wang, H.: Efficiently answering reachability queries on very large directed graphs. In: SIGMOD, pages 595–608, (2008)
21. Kempe, D., Kleinberg, J., Kumar, A.: Connectivity and inference problems for temporal networks. J. Comput. Syst. Sci. **64**(4), 820–842 (2002)
22. Li, R.-H., Su, J., Qin, L., Yu, J.X., Dai, Q.: Persistent community search in temporal networks. In: ICDE, pages 797–808 (2018)
23. Michail, O.: An introduction to temporal graphs: an algorithmic perspective. Internet Math. **12**(4), 239–280 (2016)
24. Schenkel, R., Theobald, A., Weikum, G.: Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In: ICDE, pages 360–371, (2005)
25. Semertzidis, K., Pitoura, E., Lillis, K.: Timereach: Historical reachability queries on evolving graphs. In: EDBT, pages 121–132, (2015)
26. Sengupta, N., Bagchi, A., Ramanath, M., Bedathur, S.: Arrow: Approximating reachability using random walks over web-scale graphs. In: ICDE, pages 470–481, (2019)
27. Su, J., Zhu, Q., Wei, H., Yu, J.X.: Reachability querying: can it be even faster? TKDE **29**(3), 683–697 (2016)
28. Viard, T., Latapy, M., Magnien, C.: Computing maximal cliques in link streams. Theor. Comput. Sci. **609**, 245–252 (2016)
29. Wang, H., He, H., Yang, J., Yu, P.S., Yu, J.X.: Dual labeling: Answering graph reachability queries in constant time. In: ICDE, page 75 (2006)
30. Wang, S., Lin, W., Yang, Y., Xiao, X., Zhou, S.: Efficient route planning on public transportation networks: A labelling approach. In: SIGMOD, pages 967–982, (2015)
31. Wei, H., Yu, J.X., Lu, C., Jin, R.: Reachability querying: An independent permutation labeling approach. PVLDB **7**(12), 1191–1202 (2014)
32. Wen, D., Huang, Y., Zhang, Y., Qin, L., Zhang, W., Lin, X.: Efficiently answering span-reachability queries in large temporal graphs. In: ICDE, pages 1153–1164. IEEE, (2020)
33. Wu, H., Huang, Y., Cheng, J., Li, J., Ke, Y.: Reachability and time-based path queries in temporal graphs. In: ICDE, pages 145–156, (2016)
34. Xuan, B.B., Ferreira, A., Jarry, A.: Computing shortest, fastest, and foremost journeys in dynamic networks. Int. J. Found. Comput. Sci. **14**(02), 267–285 (2003)
35. Yano, Y., Akiba, T., Iwata, Y., Yoshida, Y.: Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In: CIKM, pages 1601–1606, (2013)

36. Yıldırım, H., Chaoji, V., Zaki, M.J.: Grail: a scalable index for reachability queries in very large graphs. VLDBJ **21**(4), 509–534 (2012)
37. Yildirim, H., Chaoji, V., Zaki, M.J.: Dagger: A scalable index for reachability queries in large dynamic graphs. arXiv preprint arXiv:1301.0977, (2013)
38. Yu, J.X., Cheng, J.: Graph reachability queries: a survey. In: Managing and Mining Graph Data, pages 181–215. (2010)
39. Zhang, T., Gao, Y., Chen, L., Guo, W., Pu, S., Zheng, B., Jensen, C.S.: Efficient distributed reachability querying of massive temporal graphs. VLDBJ, pages 1–26, (2019)
40. Zhang, T., Gao, Y., Li, C., Ge, C., Guo, W., Zhou, Q.: Distributed reachability queries on massive graphs. DASFAA **11448**, 406–410 (2019)
41. Zhu, A.D., Lin, W., Wang, S., Xiao, X.: Reachability queries on large dynamic graphs: a total order approach. In: SIGMOD, pages 1323–1334, (2014)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.