On Querying Connected Components in Large Temporal Graphs

HAOXUAN XIE, The Chinese University of Hong Kong, Shenzhen, China
YIXIANG FANG*, The Chinese University of Hong Kong, Shenzhen, China
YUYANG XIA, The Chinese University of Hong Kong, Shenzhen, China
WENSHENG LUO, The Chinese University of Hong Kong, Shenzhen, China
CHENHAO MA, The Chinese University of Hong Kong, Shenzhen, China

In many real-world applications, the relationships between entities can be modeled as temporal graphs, where each edge is associated with a timestamp representing the interaction time. As a fundamental problem in network science, the connected component (CC) query has received tremendous research attention. Existing works on CC queries in the temporal graph find sets of vertices that are either connected in every timestamp of a time interval, or connected by paths with edges of increasing timestamps. However, these temporal constraints are too strict for applications without needing time-respecting paths. In this paper, we relax the above constraints by introducing a novel CC model, called window-CC, for both the undirected and directed temporal graphs in a given time window. We first propose online algorithms to query the window-CC and further develop efficient index-based query algorithms. Experimental results on real large undirected and directed temporal graphs show that our best index-based query algorithms are up to three and two orders of magnitude faster than the two online algorithms, respectively. Moreover, compared to the baseline indices, our optimized indices cost much less space in both theory and practice.

CCS Concepts: • **Mathematics of computing → Paths and connectivity problems**; **Graph algorithms**; • **Theory of computation → Design and analysis of algorithms**.

Additional Key Words and Phrases: temporal graph/network, connected component, window-CC

ACM Reference Format:
Haoxuan Xie, Yixiang Fang, Yuyang Xia, Wensheng Luo, and Chenhao Ma. 2023. On Querying Connected Components in Large Temporal Graphs. *Proc. ACM Manag. Data* 1, 2, Article 170 (June 2023), 27 pages. https://doi.org/10.1145/3589315

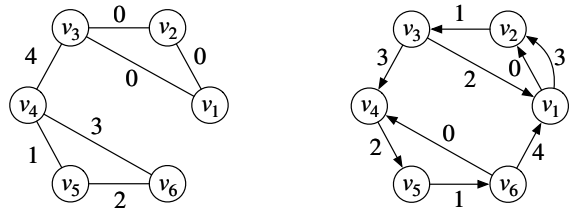
1 INTRODUCTION

In many real-world applications, the relationships between entities can be modeled as temporal graphs, where each edge is associated with a timestamp representing the interaction time. Research on temporal graphs has recently attracted much attention from both industry and research communities [29, 52, 59]. Figure 1 presents examples of undirected and directed temporal graphs, where

*Corresponding author.

Authors' addresses: Haoxuan Xie, haoxuanxie@link.cuhk.edu.cn, The Chinese University of Hong Kong, Shenzhen, Guangdong, China; Yixiang Fang, yixiangfang@cuhk.edu.cn, The Chinese University of Hong Kong, Shenzhen, Guangdong, China; Yuyang Xia, yuyangxia@link.cuhk.edu.cn, The Chinese University of Hong Kong, Shenzhen, Guangdong, China; Wensheng Luo, luowensheng@cuhk.edu.cn, The Chinese University of Hong Kong, Shenzhen, Guangdong, China; Chenhao Ma, chenhaoma@cuhk.edu.cn, The Chinese University of Hong Kong, Shenzhen, Guangdong, China.

2836-6573/2023/6-ART170 $15.00
https://doi.org/10.1145/3589315



(a) An undirected graph (b) A directed graph

Fig. 1. Two example temporal graphs.

the numbers on the edges denote the occurring time of the edges. For example, in Figure 1(a), the edge between vertices v_1 and v_2 denotes that they have an interaction (e.g., v_1 and v_2 confirms a transaction) at timestamp 0; in Figure 1(b), the edge between vertices v_2 and v_3 denotes that they have an interaction (e.g., v_2 sends a message to v_3) at timestamp 1.

In this paper, we study the problem of connected component (CC) query on temporal graphs, and the general goal is to find all the CCs in a query time window from a temporal graph. As a fundamental structure in the graph, the CC is a maximal set of vertices such that each pair of vertices is mutually reachable in the graph. It has found various real applications, such as community detection [14, 33], PPI network analysis [9, 18], and network routing protocols [2, 27]. Most of the existing works about CC focus on conventional static graphs, including distributed BFS search [4] and distributed Union-Find algorithms [11, 39], but they cannot be applied to temporal graphs due to the temporal edges. Recently, some works have attempted to study the CC query on temporal graphs. For example, Vernet et al. [51] proposed the persistent CC model for temporal graphs, which is a set of vertices that are connected in every timestamp of a time window. Bhadra and Ferreira [6, 7] introduced the temporal CC model, which is a set of vertices such that each pair of vertices is connected by a path with edges of increasing timestamps. However, these temporal constraints are too strict for applications without time-respecting paths.

To tackle the issues of existing works, we relax the temporal constraints by introducing a novel CC model, called window-CC, for both the undirected and directed temporal graphs in a time window $[t_s, t_e]$. Specifically, given a temporal graph G and a time window $[t_s, t_e]$, we first introduce the concept of the projected graph, which is a static graph formed by edges in G with timestamps in $[t_s, t_e]$. We then formulate the window-CC of the undirected temporal graph as a set of vertices that are mutually reachable by undirected paths in the projected graph. The window-CC can also be extended as window-SCC for the directed temporal graph, which is a set of vertices that are mutually reachable by directed paths in the projected graph. Based on the concepts above, we propose the window-CC and window-SCC queries, which aim to find all the window-CCs and window-SCCs from the undirected and directed temporal graphs, respectively.

Applications. The window-CC and window-SCC queries can be used in many real applications. Here we just name a few:

- **Infectious disease tracking.** To track the transmission of infectious diseases (e.g., transmission of Covid-19[1]), researchers often model the interactions between people as an undirected temporal graph, where each vertex denotes a person and each temporal edge between two vertices means they have an interaction (e.g., handshake) at a specific timestamp. The window-CCs of the graphs can be used to reveal the possible infected people. For example, if a patient is tested positive for Covid-19 at timestamp t, then all the people within the patient's window-CC of time window

[1]https://www.cdc.gov/coronavirus/2019-ncov/php/contact-tracing/contact-tracing-plan/contact-tracing.html

$[t-d, t]$ have been potentially infected, where d is the incubation period of Covid-19 (usually 2 weeks), since they have direct or indirect contacts with the patient. Note that here we mainly consider the projected graph of the incubation period, but do not need the strict time-respecting paths since the timestamps of edges may not be accurately collected in practice.

- **Active community analysis.** In social networks, users are often actively involved in different communities during different periods, where a community is often an SCC [14, 33]. For instance, the retweet relationships between Twitter users can be modeled as a directed temporal graph, where each vertex denotes a user and each temporal edge between two vertices means one retweets a message of the other at a specific timestamp. By finding the window-SCCs in different time windows (e.g., different weeks) from the graph, we can track the active communities that a user participates in. Thus, various recommendation tasks can be performed in different time periods by using the community-based A/B testing [12], which assigns different treatments (e.g., friend invitation or product recommendation) to the user, as widely investigated by LinkedIn [23] and Alipay [8].
- **Network anomaly detection.** In E-commerce, the money transfer transactions between users can be modeled as a temporal graph. Since cycles may reveal anomaly behaviors (e.g., credit card fraud) [42], the window-CCs/SCCs can be applied to detecting them. Besides, we will show later that we have used window-SCCs to identify anomaly DBLP data in Section 5.4.

	Query time	Index space	Indexing time
U-online	$O(m+n)$	$\emptyset$	$\emptyset$
U-baseline	$O(n \log\log n)$	$O(n \log n \cdot t_{max})$	$O((m+n\log n)t_{max})$
TSF-index	$O(n)$	$O(m)$	$O(mt_{max})$

Table 1. Solutions for undirected temporal graphs.

	Query time	Index space	Indexing time
D-online	$O(m+n)$	$\emptyset$	$\emptyset$
D-baseline	$O(n)$	$O(nt_{max})$	$O((m+n)t_{max}^2)$
RES-index	$O(n)$	$O(m)$	$O(mt_{max}+nt_{max}^2)$

Table 2. Solutions for directed temporal graphs.

Online solutions. The online solutions can be developed using existing algorithms of CC computation. Specifically, given an undirected temporal graph and a time window $[t_s, t_e]$, we collect all the edges with timestamps in this window. After that, we compute all the CCs from the graph projected by these edges using a CC computation algorithm (e.g., BFS or DFS). Similarly, for the directed temporal graph, we can compute all the SCCs using an SCC computation algorithm (e.g., [19, 47, 49]). However, such online solutions require accessing all the edges in the window, so the efficiency and scalability are limited, especially when the window is wide.

In this paper, we aim to design index-based solutions for efficient window-CC and window-SCC queries on large temporal graphs. A naive index is to pre-compute and store the answer for any time window. Although this approach has high query efficiency, the index itself takes $O(nt_{max}^2)$ space cost, where n denotes the number of vertices and t_{max} is the number of distinct time slots in the graph, making the space overhead render this approach impractical.

Index-based solutions. For the undirected temporal graph, we build an index by compressing all the CCs in the time windows with the same start time. Specifically, consider a fixed start time t_s. Initially, we regard each vertex v as a single CC and assign it a label $L(v)$ indicating the CC that contains it. Then, we sequentially consider the edges in $[t_s, t_{max}]$ in chronological order and merge

the disjoint CCs connected by each edge. Whenever there is a merge process between two disjoint CCs, we update the label of vertices in the smaller CC with the label of the larger CC and record the update time. After the above process, each vertex is associated with a list of $\langle label, time \rangle$ pairs, which can serve as an index, termed as U-baseline. The index costs $O(n \log n \cdot t_{max})$ space and can be built in $O((m + n \log n)t_{max})$ time, where m is the number of edges.

Although the U-baseline index is better than the naive index, it is still costly in time and space when t_{max} is large. We further propose the TSF-index, or temporal spanning forest (TSF) index, by exploiting the idea that the CCs of an undirected graph can be represented by its spanning trees. Specifically, we first introduce the concept of TSF, which preserves the spanning trees of a projected undirected graph by an edge set. We then observe that there are two kinds of overlapping relationships among the TSFs over different time windows. The first one reveals the nested relationships of TSFs sharing the same start time, while the second one shows the overlapping relationships of TSFs for multiple start times. Based on the observation, we design the TSF-index, whose time and space cost are much less than those of U-baseline index.

For the directed temporal graph, inspired by TSF-index, we develop an index called D-baseline by building a forest structure for each start time t_s, where two vertices are in the same tree iff they are in the same SCC. However, unlike the undirected graph, the SCCs of a directed graph cannot be derived by a directed spanning tree [20], so we cannot use the directed spanning tree to represent an SCC. Instead, we propose to create some undirected edges to denote the SCCs. For each start time t_s, we run SCC algorithms on the projected graph $G_{[t_s,t]}$ for each $t \in [t_s, t_{max}]$ in chronological order, and create the aforementioned forest structure. Since the forest contains at most $(n-1)$ edges, the index costs $O(nt_{max})$ space and can be built in $O((m+n)t_{max}^2)$ time.

The D-baseline index is still costly in both time and space cost when t_{max} is large. To alleviate these issues, we further develop another novel index, called RES-index or reconstruction edge set (RES) index. The key idea is that for any SCC with k vertices in the directed graph, we theoretically prove that it can be represented by a set of $2(k-1)$ edges in the original SCC, which is termed as the reconstruction edge set (RES). Inspired by the two kinds of overlapping relationships among TSFs above, we further identify two kinds of overlapping relationship among the RES's over different time windows. We further develop the RES-index which takes the same space cost as TSF-index.

We would like to remark that there is a trade-off between the online and index-based solutions. While the index-based solutions need some extra time cost for offline index construction, they outperform online solutions when the number of queries is large, because their offline processing time cost could be amortized. The experimental results on real-world temporal graphs show that our solutions are highly efficient. In particular, the queries based on the TSF-index and RES-index are up to three and two orders of magnitude faster than the two online algorithms. Besides, these two indices cost much less space than the baseline indices.

In summary, our principal contributions are as follows.

- We introduce the problems of querying window-CCs and window-SCCs on undirected and directed temporal graphs, respectively, which have not been studied in the literature yet.
- We propose several index-based solutions to support the efficient queries of window-CCs and window-SCCs. The space costs of the best indices are linear to the sizes of the graphs.
- We perform an extensive experimental evaluation on real-world datasets from SNAP and KONECT [30, 32], demonstrating the high efficiency and effectiveness of the proposed solutions.

Outline. We formulate our research problems and present online algorithms in Section 2. In Sections 3 and 4, we present the index-based algorithms for undirected and directed temporal graphs, respectively. We report experimental results in Section 5. We review the related work in Section 6 and conclude in Section 7.

2 PRELIMINARIES

We first formally present the window-CC and window-SCC queries in Section 2.1, and then show their online solutions in Section 2.2.

2.1 Problem definitions

Consider an undirected temporal graph $G = (V, E)$, where V and E denote the sets of vertices and edges, respectively. Let $|V| = n$ and $|E| = m$. Each edge $e \in E$ is a triplet (u, v, t) with $u \in V, v \in V$ and $t \in \mathbb{N}$ representing the interaction timestamp between u and v. W.l.o.g., we assume that the timestamps of edges are consecutive integer values in the range $[0, t_{max}]$, implying that $m \geq t_{max}$.

Definition 1 (projected graph). *The projected graph of an undirected temporal graph $G = (V, E)$ over a time window $[t_s, t_e]$ is $G_{[t_s,t_e]}=(V, E_{[t_s,t_e]})$, where $E_{[t_s,t_e]}=\{(u, v)|(u, v, t) \in E \wedge t \in [t_s, t_e]\}$.*

Before formulating the concept of window-CC, we present the definition of span-reachability [52].

Definition 2 (span-reachability [52]). *Given an undirected temporal graph $G = (V, E)$, two vertices u, v and a time window $[t_s, t_e]$, u span reaches v in $[t_s, t_e]$, denoted by $u \rightsquigarrow_{[t_s,t_e]} v$, if u is connected to v via a path in the projected graph $G_{[t_s,t_e]}$.*

Definition 3 (window-CC). *Given an undirected temporal graph $G = (V, E)$ and a time window $[t_s, t_e]$, the window-CC is a maximal set of vertices S, such that for any two vertices $u, v \in S$, $u \rightsquigarrow_{[t_s,t_e]} v$.*

Example 1. *In the undirected temporal graph of Figure 1(a), we have $v_1 \rightsquigarrow_{[0,4]} v_6$ but $v_1 \not\rightsquigarrow_{[0,3]} v_6$. For the time window $[0, 4]$, there is one window-CC $\{v_1, v_2, v_3, v_4, v_5, v_6\}$, while for the time window $[0, 3]$, there are two window-CCs, i.e., $\{v_1, v_2, v_3\}$ and $\{v_4, v_5, v_6\}$.*

Problem 1 (window-CC query). *Given an undirected temporal graph $G = (V, E)$ and a time window $[t_s, t_e]$, find all the window-CCs in this time window.*

For the directed temporal graph, the definitions of projected graphs and span-reachability can be similarly defined as shown in [52], so we omit the details. The concept of the window-CC can be reformulated as window-SCC, defined as follows.

Definition 4 (window-SCC). *Given a directed temporal graph $G = (V, E)$ and a time window $[t_s, t_e]$, the window-SCC is a maximal set of vertices S, such that for any two vertices $u, v \in S$, $u \rightsquigarrow_{[t_s,t_e]} v$ and $v \rightsquigarrow_{[t_s,t_e]} u$.*

Problem 2 (window-SCC query). *Given a directed temporal graph $G = (V, E)$ and a time window $[t_s, t_e]$, find all the window-SCCs in this time window.*

Example 2. *Consider the directed temporal graph in Figure 1(b). For the time window $[0, 4]$, the whole graph is a window-SCC. For the time window $[0, 2]$, there are two window-SCCs, which are $\{v_1, v_2, v_3\}$ and $\{v_4, v_5, v_6\}$.*

2.2 Online algorithms

Before presenting the online algorithms, we introduce a technique for shrinking the size of the undirected temporal graph without affecting the final query result. Since the reachability information of a static undirected graph can be maintained by a spanning forest with at most $(n - 1)$ edges, we can construct a spanning forest for each $G_{[t,t]}$ with each $t \in [0, t_{max}]$, and then replace E with the edges in the spanning forests. In practice, since the shrinking technique above can reduce the number of edges significantly, we use it to preprocess all the undirected temporal graphs by

default. Note that for the directed graph, since the directed spanning forest [20] cannot capture the reachability information, the shrinking technique above cannot be used directly.

As aforementioned, there are some existing algorithms for querying CCs or SCCs in conventional static graphs. For querying CCs in an undirected graph, the Breadth-First Search (BFS), Depth-First Search (DFS), and Union-Find data structure can give solutions in $O(m+n)$ time cost. For querying SCCs in a directed graph, various algorithms have been discovered to process in $O(m+n)$ time [19, 47, 49]. These algorithms can be used for answering our window-CC queries on temporal graphs. The general idea is that we can first build a projected graph $G_{[t_s,t_e]}$, and then run an existing CC computation algorithm above if $G_{[t_s,t_e]}$ is undirected, or an existing SCC computation algorithm above if $G_{[t_s,t_e]}$ is directed. For lack of space, we omit the details here. We denote the online algorithms for processing undirected and directed temporal graphs by U-online and D-online respectively.

Lemma 1. *The running time of the online window-CC and window-SCC query algorithms are bounded by* $O(m_{[t_s,t_e]}+n)$.

Proof. The lemma directly follows the discussions above. □

Unlike the conventional static graph, the temporal graph often has a relatively large number of edges due to the unlimited number of timestamps. Therefore, although the online query algorithms take linear time cost as stated by Lemma 1, they are still inefficient and unscalable for processing large temporal graphs. In this paper, we focus on developing efficient index-based solutions.

We would also like to note that the window-CC is defined based on span-reachability, so our solutions can also be used to answer the span-reachability queries in undirected temporal graphs [52].

3 INDEX-BASED SOLUTIONS FOR UNDIRECTED TEMPORAL GRAPHS

As mentioned in Section 1, the naive index takes $O(nt_{max}^2)$ space cost, which is impractical if t_{max} is large, so we do not further consider it in the paper. In the following, we first present two index-based solutions, and the best one only costs $O(m)$ space.

3.1 A baseline index-based solution

3.1.1 Index overview. In this section, we propose a nontrivial index named U-baseline index, and the main idea is that for each start time, we try to record a set of "labels" for each vertex, such that the window-CCs in different time windows are well-preserved.

Specifically, consider a sub-problem where the start time t_s is fixed and the goal is to answer a query with time window $[t_s, t_e]$ where $t_e \in [t_s, t_{max}]$. We observe that for the time windows $[t_s, t_s]$, $[t_s, t_s+1], \cdots, [t_s, t_{max}]$, the corresponding window-CCs would only be merged but never split, when t_e increases from t_s to t_{max}. Thus, by considering all the edges in $[t_s, t_{max}]$ in chronological order, we can label the window-CCs and keep track of label changes when they are merged. We now formally define the vertex label.

Definition 5 (Vertex label). *Given an undirected graph, the label of each vertex* v, $L(v)$, *denotes the CC containing it. Initially, each vertex* v *forms a single CC with a label* $L(v) = v$.

We denote the vertex set of the CC containing v by $S(v)$, i.e., $S(v) = \{u \in V | L(u) = L(v)\}$. By considering all the edges in chronological order, we can merge the window-CCs incrementally. Initially, each vertex v forms a single CC with $L(v) = v$. Then, for each edge (u, v, t), if u and v are not in the same window-CC, we merge the two window-CCs of u and v, by adding vertices of the small one into the large one, with the following union operation:

Definition 6 (Union). *Given an edge (u, v, t), if $|S(u)| > |S(v)|$, then we set $L(w) = L(u)$ for all $w \in S(v)$, and update $S(u) = S(u) \cup S(v)$; otherwise, we set $L(w) = L(v)$ for all $w \in S(u)$, and update $S(v) = S(u) \cup S(v)$.*

In the above process, whenever we merge two window-CCs, we can record some $\langle label, time \rangle$ pairs for vertices whose labels are changed. As a result, we obtain an index structure, which contains such pairs for each start time and each vertex, denoted by

$$B(t_s, v) = (\langle l_1, t_1 \rangle, \langle l_2, t_2 \rangle, \cdots). \tag{1}$$

Example 3. *Consider the graph in Figure 1(a). By anchoring the start time $t_s = 2$, we can build the index structure in Figure 2(a), where the process of merging CCs is depicted in Figure 2(b).*

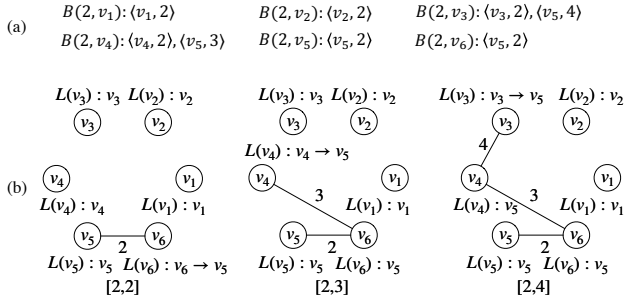


Fig. 2. The U-baseline index by anchoring $t_s = 2$.

Next, we analyze the space cost of the index by Lemmas 2 and 3.

Lemma 2. *For each anchored start time t_s, the label of each vertex is changed at most $(\log n)$ times.*

Proof. In the union operation, when we merge two window-CCs, we always add vertices in the smaller one into the larger one, by updating the labels of vertices in the smaller window-CC as the label of the larger one. Thus, for vertices in the smaller window-CC, the size of the window-CC containing them will be at least doubled after the union operation. Since the size of any window-CC is at most n, the label of each vertex is changed at most $(\log n)$ times. □

Lemma 3. *The baseline index costs $O(n \log n \cdot t_{max})$ space.*

Proof. For each start time, the space cost is $O(n \log n)$ as there are n vertices and each vertex's label is changed $O(\log n)$ times. By enumerating all the t_{max} start times, we obtain Lemma 3. □

3.1.2 Query processing. Based on the index above, to answer a query with the time window $[t_s, t_e]$, we first retrieve $B(t_s, v)$ for each vertex $v \in V$. Then, for each vertex v, we find the pair $\langle label, time \rangle$ corresponding to the query time window. Finally, we build an undirected graph and compute the CCs from it. Algorithm 1 presents the detailed steps. It initializes a graph G' with only n vertices (line 1). Then, for each vertex v, it binary searches the $\langle l_k, t_k \rangle$ satisfying $t_k \le t_e$ and $t_{k+1} > t_e$ from $B(t_s, v)$ (lines 2-3), and adds an edge (v, l_k) to G' (line 4). Note that after the process above, G' has $n - 1$ edges. Finally, it finds and returns all the CCs in G' (line 5).

Lemma 4. *Algorithm 1 answers a query in $O(n \log \log n)$ time.*

Proof. By Lemma 2, each $B(t_s, v)$ has at most $(\log n)$ $\langle label, time \rangle$ pairs. Thus, the binary search on each $B(t_s, v)$ costs $O(\log \log n)$ time, making the overall query time cost be $O(n \cdot \log \log n)$. □

Algorithm 1: U-baseline-query(B, $[t_s, t_e]$)

Input: the index B and query time window $[t_s, t_e]$;
Output: all the window-CCs in $G_{[t_s,t_e]}$;
1 $G' \leftarrow (V, E')$ where $E' = \emptyset$;
2 **for** $v \in V$ **do**
3 $\quad \langle l_k, t_k \rangle \leftarrow$ binary search a pair $\langle l_k, t_k \rangle$ such that $t_k \le t_e$ and $t_{k+1} > t_e$ from $B(t_s, v)$;
4 $\quad$ add an edge (v, l_k) between v and l_k into G';
5 **return** all the CCs computed from G';

3.1.3 Index construction. To build the index, for each start time t_s, we sequentially consider the edges in chronological order, during which we merge the CCs and record the $\langle label, time \rangle$ pairs. Algorithm 2 shows the details. For each start time t_s, we first initialize $L(v)$, $S(v)$, and $B(t_s, v)$ for each vertex v (lines 1-4). We then sequentially consider the edges in chronological order, merge window-CCs, and record the $\langle label, time \rangle$ pairs (lines 5-14). For each edge (u, v, t), if the two end vertices are not in the same CC, we merge the vertices of the small CC into the large one and change their labels, during which the $\langle label, time \rangle$ pairs are recorded.

Algorithm 2: U-baseline-construct(G)

Input: an undirected temporal graph $G = (V, E)$;
Output: the index $B(t_s, v)$ for all $0 \le t_s \le t_{max}$ and all $v \in V$;
1 **for** $t_s \in [0, t_{max}]$ **do**
2 $\quad$ **for** $v \in V$ **do**
3 $\quad\quad L(v) \leftarrow v, S(v) \leftarrow \{v\}$;
4 $\quad\quad B(t_s, v) \leftarrow (\langle v, t_s \rangle)$;
5 $\quad$ **for** $(u, v, t) \in E \wedge t \ge t_s$ **do**
6 $\quad\quad$ **if** $L(u) \neq L(v)$ **then**
7 $\quad\quad\quad$ **if** $|S(u)| \ge |S(v)|$ **then**
8 $\quad\quad\quad\quad S(u) \leftarrow S(u) \cup S(v)$;
9 $\quad\quad\quad\quad$ **for** $w \in S(v)$ **do**
10 $\quad\quad\quad\quad\quad L(w) \leftarrow L(u), S(w) \leftarrow S(u)$;
11 $\quad\quad\quad\quad\quad$ delete all $\langle l, t \rangle$ pairs in $B(t_s, w)$;
12 $\quad\quad\quad\quad\quad$ append $\langle L(w), t \rangle$ to $B(t_s, w)$;
13 $\quad\quad\quad$ **else**
14 $\quad\quad\quad\quad$ merge $S(u)$ into $S(v)$ and update labels (similar to lines 8-12);

Lemma 5. *Algorithm 2 takes $O((m + n \log n) t_{max})$ time cost.*

Proof. There are $O(t_{max})$ start times and for each t_s, we examine at most m edges, and each vertex is involved in $O(\log n)$ union operations, so the overall time cost is $O((m + n \log n) t_{max})$. □

In addition, the U-baseline index can be easily updated without re-building it from scratch, when a new edge $(u, v, t_{max} + 1)$ is added into the graph. Specifically, we can design a revised version of Algorithm 2, by ranging t_s from 0 to $(t_{max} + 1)$ in its outer loop. When $t_s \in [0, t_{max}]$, instead of the trivial initialization in lines 2-4, we re-use the sets L, S with the previously derived $B(t_s, \cdot)$, and only consider the newly added edge by running lines 5-14; when $t_s = t_{max} + 1$, we directly follow the original procedure.

3.2 An advanced index-based solution

Generally, the solution using the U-baseline index performs well on small-to-moderate-sized graphs, but the index space cost may still be too large, especially when t_{max} is large. To further reduce the space and time cost of indexing, we propose an advanced index, called temporal spanning forest (TSF) index, or TSF-index, by exploiting the idea that on an undirected graph, each CC corresponds to a spanning tree of its spanning forest.

3.2.1 The concept of TSF and its properties. We first introduce the novel concept of TSF.

Definition 7 (TSF). *Given an undirected temporal graph $G = (V, E)$ and a time window $[t_s, t_e]$, the TSF over $[t_s, t_e]$ is the minimum edge set $\Psi_{[t_s,t_e]} \subseteq E$, such that all the CCs of the graph $G' = (V, \Psi_{[t_s,t_e]})$ are exactly the window-CCs of $G_{[t_s,t_e]}$.*

By Definition 7, the TSF has the minimum number of edges, implying that the edges form the spanning trees. Next, we show that there exist two kinds of overlapping relationships among different TSFs, allowing us to design a space-efficient index.

• **Nested relationship by anchoring a start time.** This is described by Lemma 6.

Lemma 6. *Given an undirected temporal graph G and an anchored start time t_s, there exists a chain of TSFs such that*

$$\Psi_{[t_s,t_s]} \subseteq \Psi_{[t_s,t_s+1]} \subseteq \cdots \subseteq \Psi_{[t_s,t_{max}]}. \tag{2}$$

Proof. We prove the lemma by giving a method to construct such a chain. Suppose that $\Psi_{[t_s,t_s]}$ has been derived by an MST algorithm [22]. We show that $\Psi_{[t_s,t_s+1]}$ can be derived based on $\Psi_{[t_s,t_s]}$. Specifically, we first initialize $\Psi_{[t_s,t_s+1]} = \Psi_{[t_s,t_s]}$, and then update it by the edges with timestamp $t_s + 1$. For each edge $(u, v, t_s + 1)$, if vertices u and v are not connected via the edges in $\Psi_{[t_s,t_s+1]}$, we add it into $\Psi_{[t_s,t_s+1]}$. After processing all the edges with $t = t_s + 1$, we obtain $\Psi_{[t_s,t_s+1]}$. By repeating the above process, we can derive a chain of TSFs satisfying Eq. (2). Figure 3 shows this process when we are constructing $\Psi_{[2,t]}$ of Figure 1(a). □

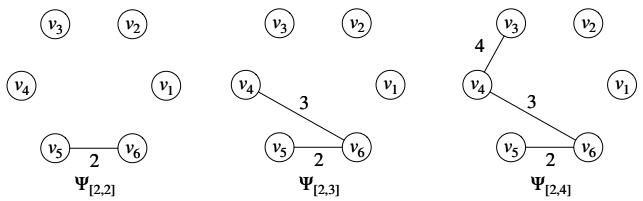


Fig. 3. The nested relationship by anchoring $t_s = 2$.

By Lemma 6, for each anchored t_s, we can compress all the TSFs $\Psi_{[t_s,t_s]}, \Psi_{[t_s,t_s+1]}, \cdots, \Psi_{[t_s,t_{max}]}$ using $O(n)$ space cost by only keeping the newly added edges for each TSF. However, if we directly compress all the TSFs for each $t_s \in [0, t_{max}]$ as an index, it will take $O(nt_{max})$ space cost, which is still costly when t_{max} is large. To overcome the enormous space cost caused by t_{max}, we observe and exploit another overlapping relationship among TSFs.

• **Overlapping relationship for multiple start times.** We begin with an important observation in Lemma 7, which allows us to compress all the TSFs by using only $O(m)$ space.

Lemma 7. *Given an undirected temporal graph G, there exist $(t_{max}+1)$ TSFs, i.e., $\Psi_{[0,t_{max}]}, \Psi_{[1,t_{max}]}, \cdots, \Psi_{[t_{max},t_{max}]}$, such that for each edge (u, v, t), there is a chain starting from some $\widehat{t} \in [0, t]$*

$$\Psi_{[\widehat{t},t_{max}]}, \Psi_{[\widehat{t}+1,t_{max}]}, \cdots, \Psi_{[t,t_{max}]}, \tag{3}$$

satisfying (1) each TSF in the chain contains the edge and (2) the edge is excluded in any TSFs that are not in the chain.

Proof. We prove the lemma by giving a method to construct a set of TSFs satisfying the condition in the lemma. Suppose that $\Psi_{[0,t_{max}]}$ has been derived by the process in Lemma 6. We then can build $\Psi_{[1,t_{max}]}$ based on $\Psi_{[0,t_{max}]}$. Specifically, we first initialize $\Psi_{[1,t_{max}]}$ as $\Psi_{[0,t_{max}]} \cap E_{[1,t_{max}]}$, i.e. delete edges with timestamp 0 in $\Psi_{[0,t_{max}]}$, and then update it by the edges in $E_{[1,t_{max}]}$. Consider the edges in chronological order. For each edge (u, v, t), if vertices u and v are not connected via the edges in $\Psi_{[1,t_{max}]}$, we add it into $\Psi_{[1,t_{max}]}$. After processing all edges in $E_{[1,t_{max}]}$, we obtain $\Psi_{[1,t_{max}]}$. By repeating the above process, we can derive a set of TSFs $\Psi_{[0,t_{max}]}, \Psi_{[1,t_{max}]}, \cdots, \Psi_{[t_{max},t_{max}]}$. Since each edge with timestamp t will be added into Ψ for some $\widehat{t}$ and deleted in the initialization of $\Psi_{[t+1,t_{max}]}$, the condition in the lemma holds. Figure 4 shows this process when we are constructing $\Psi_{[2,t_{max}]}$ from $\Psi_{[1,t_{max}]}$ of Figure 1(a). □

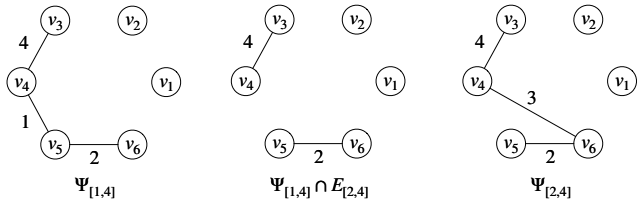


Fig. 4. The overlapping relationship between $t_s = 1, 2$

By Lemma 7, for each edge (u, v, t), we can identify an *appearing time interval* $[\widehat{t}, t]$ such that for any $t' \in [\widehat{t}, t]$, the TSF over $[t', t_{max}]$ contains it.

3.2.2 Index overview. After processing each edge and obtaining its appearing time interval, we can reversely organize the edges and their appearing time intervals, such that for each time interval $[t_i, t_j]$ where $0 \le t_i \le t_j \le t_{max}$, there is a set of edges corresponding to it, denoted by $F(t_i, t_j)$, which can serve as the TSF-index.

Example 4. *Figure 5(a) presents the appearing time interval for each edge of the graph in Figure 1(a), based on which we can design the TSF-index as depicted in Figure 5(b).*

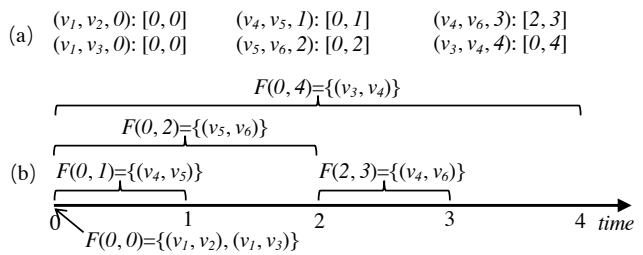


Fig. 5. The TSF-index for the graph in Figure 1(a).

Lemma 8. *The TSF-index costs $O(m)$ space.*

Proof. By Lemma 7, each edge has only one appearing time interval, so it appears only once in the index, making the overall space cost be $O(m)$. Note that $F(t_i, t_j)$ can be stored in a sparse manner, i.e., we only need to store the $F(t_i, t_j)$ if it is non-empty. □

3.2.3 Query processing. To answer a query, we first utilize the TSF-index to find all edges in the TSF over $[t_s, t_e]$, then identify all the window-CCs from G'. Algorithm 3 presents the details. It initializes a new static graph G' with the original n vertices (line 1). Then, it adds the edges of all

$F(t_i, t_j)$ with $t_i \le t_s$ and $t_s \le t_j \le t_e$ into G' (lines 2-3), which form the TSF over $[t_s, t_e]$. Finally, it finds and returns CCs on G' as the window-CCs (line 4).

Algorithm 3: TSF-query(F, t_s, t_e)

Input: the TSF-index F and query time window $[t_s, t_e]$;
Output: all the window-CCs in $G_{[t_s,t_e]}$;
1 $G' \leftarrow (V, E')$ where $E' = \emptyset$;
2 **for** $t_i \in [0, t_s], t_j \in [t_s, t_e]$ **do**
3 $\quad E' \leftarrow E' \cup F(t_i, t_j)$;
4 **return** all the CCs computed from $G' = (V, E')$;

Lemma 9. *Algorithm 3 answers a query in $O(n)$ time.*

Proof. Since the graph G' has n vertices and at most $n-1$ edges and it represents the TSF over $[t_s, t_e]$, i.e., $\Psi_{[t_s,t_e]}$, finding all the CCs from G' can be completed in $O(n)$ time. □

3.2.4 Index construction. The TSF-index can be built by computing the TSFs for all the possible time windows and compressing them compactly. We show the steps in Algorithm 4. We first anchor each start time t_i (line 1). Then for each $t_j \in [t_i, t_{max}]$, we initialize $\Psi_{[t_i,t_j]}$ by $\Psi_{[t_i,t_j-1]}$ (lines 2-4). Afterwards, we process the edges in $[t_i, t_{max}]$ in chronological order; that is, for each edge (u, v, t_j), if vertices u and v are not connected via the edges in $\Psi_{[t_i,t_j]}$, we add it to $\Psi_{[t_i,t_j]}$ (lines 5-7). In the meantime, if the edge is added for the first time, we put it into $F(t_i, t_j)$ according to Lemma 7 (lines 8-9). After processing every time interval $[t_i, t_j]$, we can derive the set of edges in each $F(t_i, t_j)$ correspondingly.

Algorithm 4: TSF-construct(G)

Input: an undirected temporal graph $G = (V, E)$;
Output: the index $F(t_i, t_j)$ for all $0 \le t_i \le t_j \le t_{max}$;
1 **for** $t_i \leftarrow 0, \cdots, t_{max}$ **do**
2 $\quad$ **for** $t_j \leftarrow t_i, t_i + 1, \cdots, t_{max}$ **do**
3 $\quad\quad \Psi_{[t_i,t_j]} \leftarrow \emptyset$;
4 $\quad\quad$ **if** $t_j > t_i$ **then** $\Psi_{[t_i,t_j]} \leftarrow \Psi_{[t_i,t_j-1]}$;
5 $\quad\quad$ **for** $(u, v, t_j) \in E$ **do**
6 $\quad\quad\quad$ **if** *u and v are not connected by $\Psi_{[t_i,t_j]}$* **then**
7 $\quad\quad\quad\quad \Psi_{[t_i,t_j]} \leftarrow \Psi_{[t_i,t_j]} \cup \{(u, v, t_j)\}$;
8 $\quad\quad\quad\quad$ **if** $(u, v, t_j) \notin \Psi_{[t_i-1,t_{max}]}$ **then**
9 $\quad\quad\quad\quad\quad F(t_i, t_j) \leftarrow F(t_i, t_j) \cup \{(u, v, t_j)\}$;

Lemma 10. *Algorithm 4 costs $O(mt_{max})$ time.*

Proof. In Algorithm 4, each edge with timestamp t_j will be iterated for $O(t_j)$ times, since it would be considered for each $t_i \in [0, t_j]$. Hence, the total time cost is bounded by $O(mt_{max})$. □

The TSF-index can also be easily maintained when a new edge $(u, v, t_{max} + 1)$ is included in the graph. Specifically, we can revise Algorithm 4, such that t_i is ranged from 0 to $(t_{max} + 1)$ in its outer loop. When $t_i \in [0, t_{max}]$, the variable t_j in the inner loop is fixed as $(t_{max} + 1)$, and instead of initializing the TSF by lines 3-4, we reconstruct $\Psi_{[t_i,t_{max}]}$ from the current TSF-index and then run the original steps in lines 5-9; when $t_i = t_{max} + 1$, we run the original procedure of the inner loop.

4 INDEX-BASED SOLUTIONS FOR DIRECTED TEMPORAL GRAPHS

Inspired by the indices for undirected graphs, we first give a spanning forest-based index for window-SCC queries on directed temporal graphs. To further reduce the indexing time and space cost, we propose an advanced index with $O(m)$ space.

4.1 A baseline index-based solution

4.1.1 Index overview. In this section, we propose an index named D-baseline index. Unlike undirected graphs, an SCC in the directed graph cannot be represented by a directed spanning tree [20], so we cannot build a directed version of the TSF-index for directed temporal graphs. Instead, we propose another forest structure of undirected edges, which connects the vertices in each SCC. Specifically, we maintain an undirected edge set $D(t_s)$ of the forest for each start time t_s, where edges are associated with timestamps. If u and v are in the same SCC in $G_{[t_s,t_e]}$, then u and v are connected via the edges in $D(t_s)$ with timestamps $\leq t_e$. Figure 6 depicts the constructed forests for the graph in Figure 1(b).

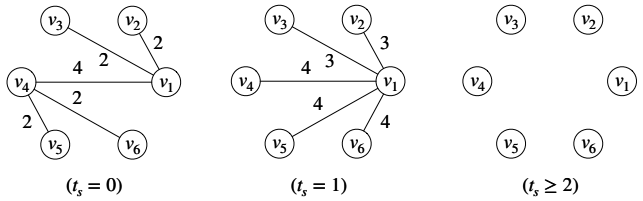


Fig. 6. Constructed forests with different $t_s \in [0, 4]$.

Lemma 11. *The D-baseline index costs $O(nt_{max})$ space.*

Proof. For each start time t_s, the space cost is $O(n)$ as $|D(t_s)| \leq n$. By enumerating all the t_{max} start times, the lemma holds. □

4.1.2 Query processing. To solve the query over a time window $[t_s, t_e]$, we first retrieve all the edges in $D(t_s)$ with timestamps $\leq t_e$. Then, we construct a new static graph $G' = (V, E')$ where V contains all the vertices of the graph G and E' represents the retrieved edges. Finally, we detect all the CCs from graph G'. Algorithm 5 shows the steps above. It firstly initializes a graph G' without edges (line 1), then adds all the edges in $D(t_s)$ with timestamps in $[t_s, t_e]$ into G' (line 2), and finally returns CCs on G' as the window-SCCs (line 3), which actually are the same as the SCCs in $G_{[t_s,t_e]}$.

Algorithm 5: D-baseline-query(D, $[t_s, t_e]$)

Input: the index D and the query time window $[t_s, t_e]$;
Output: all window-SCCs in $G_{[t_s,t_e]}$;
1 $G' \leftarrow (V, E')$ where $E' = \emptyset$;
2 **for** $e = (u, v, t) \in D(t_s) \wedge t \leq t_e$ **do** $E' \leftarrow E' \cup \{(u, v)\}$;
3 **return** all the CCs computed from G';

Lemma 12. *Algorithm 5 answers a query in $O(n)$ time.*

Proof. Since $D(t_s)$ is a forest structure with n vertices, its number of edges is at most $n - 1$. Hence, the total time cost is $O(n)$. □

4.1.3 Index construction. To build the index, for each start time t_s, we iteratively find all SCCs in $G_{[t_s,t]}$ for each $t \in [t_s, t_{max}]$, and create the minimal undirected edges to connect the vertices newly involved in the same SCC at time t. Algorithm 6 gives the details. We first anchor each start

time t_s and initialize $D(t_s)$ (lines 1-2). As the time window expands to $[t_s, t]$, we run algorithms to find all SCCs in $G_{[t_s,t]}$ (lines 3-4). Then for each SCC, we add undirected edges into D until the vertices in the SCC are connected by the edges in D (lines 5-9).

Algorithm 6: D-baseline-construct(G)

```
   Input: a directed temporal graph G = (V, E);
   Output: the index D(t_s) for all 0 ≤ t_s ≤ t_max;
1  for t_s = 0, 1, ··· , t_max do
2      D(t_s) ← ∅;
3      for t ← t_s, t_s + 1, ··· , t_max do
4          run algorithms to find SCCs on G_[t_s,t];
5          for each SCC in G_[t_s,t] do
6              choose a vertex u in the SCC;
7              for all other vertices v in the SCC do
8                  if u and v are not connected by D(t_s) then
9                      D(t_s) ← D(t_s) ∪ {(u, v, t)};
```

Lemma 13. *Algorithm 6 costs $O((m+n)t_{max}^2)$ time.*

Proof. For each time window $[t_s, t_e]$, the time complexity of finding all SCCs in $G_{[t_s,t_e]}$ is $O(m_{[t_s,t_e]} + n)$ [19, 47, 49], so the overall time complexity is $O\left(\sum_{t_s,t_e} m_{[t_s,t_e]} + nt_{max}^2\right)$. Since $m_{[t_s,t_e]} \leq m$, the complexity is also bounded by $O((m+n)t_{max}^2)$. □

In addition, when a new edge $(u, v, t_{max} + 1)$ arrives, we can update the D-baseline index without re-building it from scratch. Specifically, we can design a revised version of Algorithm 6, by ranging t_s from 0 to $(t_{max} + 1)$ in its outer loop. When $t_s \in [0, t_{max}]$, the variable t in the inner loop (line 3) is fixed as $t = t_{max} + 1$; when $t_s = t_{max} + 1$, we run the original procedure of the inner loop.

4.2 An advanced index-based solution

The solution based on the D-baseline index handles small graphs smoothly but is costly when t_{max} is large. To reduce the enormous indexing cost caused by t_{max}, we propose an advanced index, called reconstruction edge set (RES) index, or RES-index, by maintaining small sets of edges to reconstruct SCCs, whose space cost is $O(m)$.

4.2.1 The concept of RES and its properties. The D-baseline index does not have the nested relationships w.r.t. different anchoring start times, as depicted in Figure 6. Hence, it is hard to design the space-efficient index by compressing the D-baseline indexes. Inspired by TSF-index (Definition 7), we can also extract the directed edge subset of E to reconstruct the SCCs concerning different time windows. We conjecture that such reconstruction edge sets can have nested relationships that allow compression. To verify it, we determine the edges contributing to SCCs by defining the reconstruction edge set (RES) in Definition 8.

Definition 8 (RES). *Given a directed temporal graph $G = (V, E)$ and a time window $[t_s, t_e]$, the RES over $[t_s, t_e]$ is an edge set $\Phi_{[t_s,t_e]} \subseteq E$, such that all the SCCs of the graph $G' = (V, \Phi_{[t_s,t_e]})$ are exactly the window-SCCs of $G_{[t_s,t_e]}$.*

As the RES's are designed for indexing, it is desired that the size of $\Phi_{[t_s,t_e]}$ can be bounded by $O(n)$. To fulfill such a requirement, we give an approach to find $\Phi_{[t_s,t_e]}$ by carefully choosing edges that strongly connect an SCC. Inspired by the ideas of Kosaraju algorithm [47], we can perform two BFS procedures on an SCC with k vertices and its reverse graph, respectively, and use the edges

gone through by the procedures to represent the SCC, where the number of such edges is at most $2k-2$. Iterating over all SCC's, we can obtain $\Phi_{[t_s,t_e]}$, which has most $2n-2$ edges. The details to process an SCC are presented in Algorithm 7. We first obtain the reverse graph of G' and then select a vertex in S randomly (lines 1-2). Next, we use BFS to find the edges in G' and the reverse of G' (lines 3-6).

Algorithm 7: Find-RES(G', S)

Input: an SCC S and the subgraph G' induced by S;
Output: a set of no more than $2|S|-2$ edges representing S;
1 $\overline{G}' \leftarrow$ the reverse graph of G';
2 select a vertex $u \in S$ randomly;
3 perform BFS on S in G' starting from u;
4 $E_1 \leftarrow$ the edge set which BFS went through;
5 perform BFS on S in $\overline{G}'$ starting from u;
6 $E_2 \leftarrow$ the edge set which BFS went through;
7 $\overline{E}_2 \leftarrow$ the reverse edges in E_2;
8 **return** $E_1 \cup \overline{E}_2$;

Similar to TSFs, there exist two kinds of overlapping relationships among different RES's, discussed as follows.

• **Nested relationship by anchoring a start time.** This is illustrated by Lemma 14.

Lemma 14. *Given a directed temporal graph G and an anchored start time t_s, there exists a chain of RES's such that*

$$\Phi_{[t_s,t_s]} \subseteq \Phi_{[t_s,t_s+1]} \subseteq \cdots \subseteq \Phi_{[t_s,t_{max}]}. \tag{4}$$

Proof. We prove the lemma by giving a method to construct such a chain. Suppose that $\Phi_{[t_s,t_s]}$ has been derived by invoking Algorithm 7 on each SCC of $G_{[t_s,t_s]}$. We show that $\Phi_{[t_s,t_s+1]}$ can be derived based on $\Phi_{[t_s,t_s]}$. We shrink each SCC of $G_{[t_s,t_s]}$ into a vertex and update $G_{[t_s,t_s+1]}$ with the shrinked vertices accordingly. For each SCC in the updated $G_{[t_s,t_s+1]}$, we use Algorithm 7 to find edges that strongly connect the SCC. By combining those edges with $\Phi_{[t_s,t_s]}$, we obtain $\Phi_{[t_s,t_s+1]}$. By repeating the above process, we can derive a chain of RES's satisfying Eq. (4). Figure 7 shows this process by constructing $\Phi_{[0,t_{max}]}$ for the graph in Figure 1(b). □

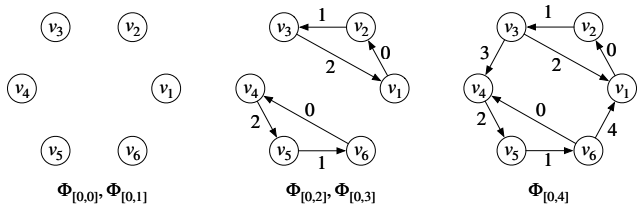


Fig. 7. The nested relationship by anchoring $t_s = 0$

By Lemma 14, for each anchored start time t_s, we can compress all RES's $\Phi_{[t_s,t_s]}, \Phi_{[t_s,t_s+1]}, \cdots, \Phi_{[t_s,t_{max}]}$ using $O(n)$ space cost, by only keeping the newly added edges for each RES with the help of Algorithm 7. For example, compressing the RES's for $t_s = 0$ will give us $\Phi_{[0,t_{max}]}$ in the leftmost subfigure of Figure 8. Taking a closer look at Figure 8, we can find that filtering the edges appearing in $[0,1]$ in the RES $\Phi_{[0,t_{max}]}$ gives us more edges than $\Phi_{[0,1]}$, which is different from TSFs. A natural question is whether those edges $\Phi_{[0,t_{max}]} \cap E_{[0,1]}$ connect the same SCCs with $\Phi_{[0,1]}$. We prove that $\Phi_{[t_s,t_{max}]} \cap E_{[t_s,t_e]}$ is indeed a valid RES to recover the corresponding SCCs by Lemmas 15 and 16.

Lemma 15. *If $E_1 \subseteq E_{[t_s,t_e]}$ is a valid RES for some $\Phi_{[t_s,t_e]}$, then $E_1 \cup E_2$ is also a valid RES for $\Phi_{[t_s,t_e]}$, where $E_2 \subseteq E_{[t_s,t_e]}$.*

Proof. We prove the lemma by contradiction. Assume $G'' = (V, E_1 \cup E_2)$ has a different set of SCCs as that of $G' = (V, E_1)$, then let $E_2 = E_{[t_s,t_e]}$, $G'' = G_{[t_s,t_e]}$ has inconsistent SCCs with $G' = (V, E_1)$. This contradicts Definition 8. □

Lemma 16. *$\Phi_{[t_s,t_{max}]} \cap E_{[t_s,t_e]}$ is a valid RES for $\Phi_{[t_s,t_e]}$.*

Proof. Since $\Phi_{[t_s,t_e]} \subseteq \Phi_{[t_s,t_{max}]}$ and $\Phi_{[t_s,t_e]} \subseteq E_{[t_s,t_e]}$, there exists a valid RES satisfying $\Phi_{[t_s,t_e]} \subseteq \Phi_{[t_s,t_{max}]} \cap E_{[t_s,t_e]}$. By Lemma 15, $\Phi_{[t_s,t_{max}]} \cap E_{[t_s,t_e]}$ is also a valid RES. □

By Lemma 16, it is safe to only keep track of $\Phi_{[t_s,t_{max}]}$ for each $t_s \in [0, t_{max}]$. However, if we only compress RES's by each anchored start time, the index covering all start times will still take $O(n \cdot t_{max})$ space cost, which is the same as that of D-baseline index. To alleviate the enormous space cost caused by t_{max}, we exploit another overlapping relationship among RES's.

• **Overlapping relationship for multiple start times.** We present an important observation in Lemma 17, which allows us to compress all the RES's by using only $O(m)$ space.

Lemma 17. *Given an directed temporal graph G, there exist $(t_{max}+1)$ RES's, i.e., $\Phi_{[0,t_{max}]}$, $\Phi_{[1,t_{max}]}$, $\cdots$, $\Phi_{[t_{max},t_{max}]}$, such that for each edge (u, v, t), there is a chain and some $t_l, t_r \in [0, t]$*

$$\Phi_{[t_l,t_{max}]}, \Phi_{[t_l+1,t_{max}]}, \cdots, \Phi_{[t_r,t_{max}]}, \tag{5}$$

satisfying (1) each RES in the chain contains the edge and (2) the edge is excluded in any RES's that are not in the chain.

Proof. We prove the lemma by giving a method to construct a set of RES's satisfying the condition in the lemma. Suppose that $\Phi_{[0,t_{max}]}$ has been derived by the process in Lemma 14. We then can build $\Phi_{[1,t_{max}]}$ based on $\Phi_{[0,t_{max}]}$. Specifically, when calling Algorithm 7 to collect edges for $\Phi_{[1,t_{max}]}$, the edges in $\Phi_{[0,t_{max}]}$ have the privilege to be traversed first when performing BFS. By repeating the above process, we can derive a set of RES's $\Phi_{[0,t_{max}]}$, $\Phi_{[1,t_{max}]}$, $\cdots$, $\Phi_{[t_{max},t_{max}]}$. Since each edge with timestamp t will be added into Φ for some t_l and deleted for some $t_r + 1$, the condition in the lemma holds. □

By Lemma 17, for each edge (u, v, t), we can identify an *appearing time interval* $[t_l, t_r]$ such that for any $t' \in [t_l, t_r]$, the RES over $[t', t_{max}]$ contains it. For example, the edge $(v_5, v_6, 1)$ appears in the RES's starting from $t_s \in [0, 1]$, as shown in Figure 8.

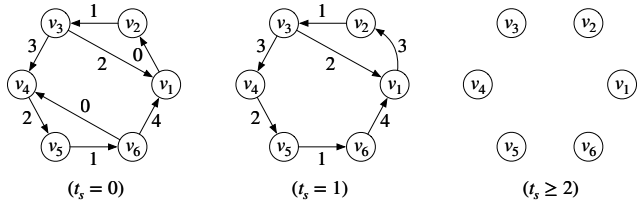


Fig. 8. The structure of RES's.

4.2.2 Index overview. After obtaining the appearing time intervals for all edges, we can reversely organize the edges and their appearing time intervals, such that for each time interval $[t_i, t_j]$ where $0 \leq t_i \leq t_j \leq t_{max}$, there is a set of edges corresponding to it, denoted by $R(t_i, t_j)$, which can serve as the RES-index.

Example 5. *Figure 9(a) presents the appearing time interval for each edge of the graph in Figure 1(b), based on which we can design the RES-index as depicted in Figure 9(b).*

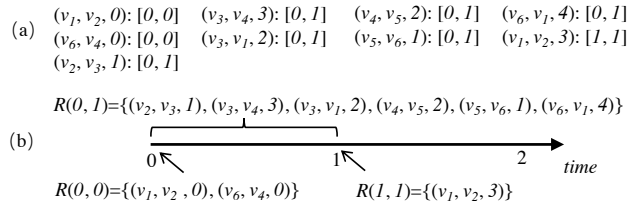


Fig. 9. The RES-index for the graph in Figure 1(b).

Lemma 18. *The RES-index costs $O(m)$ space.*

Proof. The lemma can be proved similarly to Lemma 8. □

4.2.3 Query processing. To answer the query over a time window $[t_s, t_e]$, we first utilize the RES-index to find all the edges in the RES over $[t_s, t_e]$. Then we can construct a new static graph $G' = (V, E')$ where V remains the same as the original graph G and E' represents the retrieved edges. Finally, we run SCC algorithms on the graph G' to detect all the SCCs. Algorithm 8 presents the details. The algorithm first constructs an empty graph G' (line 1). Then, it adds all the edges in $R(t_i, t_j)$ where $t_i \le t_s$ and $t_s \le t_j \le t_e$ into G' (line 2). Finally, it computes and returns SCCs on G' (line 3).

Algorithm 8: RES-query(Φ, t_s, t_e)

Input: the RES-index R and query time window $[t_s, t_e]$;
Output: all window-SCCs in $G_{[t_s,t_e]}$;
1 $G' \leftarrow (V, E'), E' \leftarrow \emptyset$;
2 **for** $t_i \in [0, t_s], t_j \in [t_s, t_e]$ **do** $E' \leftarrow E' \cap R(t_i, t_j)$;
3 **return** all the SCCs computed from G';

Lemma 19. *Algorithm 8 answers a query in $O(n)$ time.*

Proof. In Algorithm 8, the constructed edge set E' corresponds to a valid $\Phi_{[t_s,t_e]}$. Since $|\Phi_{[t_s,t_e]}|$ is bounded by $O(n)$, the query time complexity is bounded by $O(|\Phi_{[t_s,t_e]}| + n) = O(n)$. □

4.2.4 A two-pointer optimization. As mentioned above, we would need to compute $\Phi_{[t_s,t_{max}]}$ for each $t_s \in [0, t_{max}]$ before obtaining RES-index. A straightforward idea is to run SCC algorithms on $G_{[t_s,t_e]}$ for all time windows. This is unrealistic since it costs $O((m+n)t_{max}^2)$ time. Fortunately, we observe that for each edge with the start time $t_s \in [t_l, t_r]$ in Lemma 17, the smallest t when it is involved in an SCC of $G_{[t_s,t]}$ is increasing as t_s increases. We define such t as the **effective time** in Definition 9 and we formally prove the observation above in Lemma 20.

Definition 9 (effective time). *Given an edge e and a start time t_s, if both vertices of e are in the same SCC of $G_{[t_s,t]}$, but they are not in the same SCC of $G_{[t_s,t-1]}$, then the effective time of e at t_s is defined as $T(t_s, e) = t$. If e is never involved in an SCC, $T(t_s, e) = t_{max} + 1$.*

Lemma 20 (monotonicity of effective time). *For an edge e in G, it holds for all $t_s \ge 1$ that $T(t_s, e) \ge T(t_s - 1, e)$.*

Proof. We prove the lemma by contradiction. Assume $T(t_s, e) < T(t_s - 1, e)$. Then $T(t_s, e) \le t_{max}$ and e is involved in an SCC of $G_{[t_s,T(t_s,e)]}$. Therefore, it should be also involved in an SCC of $G_{[t_s-1,T(t_s,e)]}$, which contradicts our assumption. □

By Lemma 20, the effective times of an edge e at different t_s from its appearing time interval $[t_l, t_r]$ are monotonically increasing.

Example 6. *As shown in Figure 10, the edge $e = (v_1, v_2, 3)$ has effective times 2, 3, 5. When $t_s = 0$, v_1, v_2 starts to form an SCC at time $T(0, e) = 2$. Similarly $T(1, e) = 3$. When $t_s \geq 2$, v_1, v_2 can never form an SCC. Then $T(t_s, e) = t_{max} + 1 = 5$ for $t_s \geq 2$. Hence, this edge e only needs to be considered when running SCC algorithms for $G_{[t_s, t']}$, where $T(t_s - 1, e) \leq t' \leq T(t_s, e)$.*

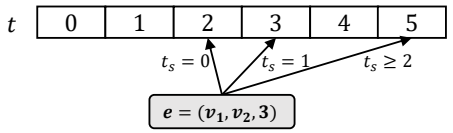


Fig. 10. Effective times of $e = (v_1, v_2, 3)$.

The number of different time windows where an edge is considered is $O(t_{max})$ by Lemma 21. Therefore, the time complexity is reduced to $O(mt_{max} + nt_{max}^2)$.

Lemma 21. *For an edge e in G, it can be considered in $2t_{max} + 1 = O(t_{max})$ different time windows.*

Proof. For the start time t_s, the number of different time windows is $T(t_s, e) - T(t_s - 1, e) + 1$. Therefore, the total number of different time windows is:

$$\sum_{t_s=0}^{t_{max}} (T(t_s, e) - T(t_s - 1, e) + 1) = t_{max} + 1 + t_{max} = 2 \cdot t_{max} + 1,$$

where $T(-1, e) = 0$. The equation is asymptotic to $O(t_{max})$. □

4.2.5 Index construction. Based on the above two-pointer optimization, the RES-index can be built by computing the RES's for all possible time windows efficiently and compressing them compactly.

Algorithm 9 shows the detailed steps. We first anchor each start time t_i and initialize G' (lines 1-2). The vertex set V' represents the set of current SCCs in G. Since the initial graph is empty, the initial value of V' is V. As the time window expands to $[t_i, t_j]$, we initialize $\Phi_{[t_i,t_j]}$ by $\Phi_{[t_i,t_j-1]}$ based on Lemma 14 (lines 3-5). For each original edge $e = (u, v, t_j) \in E_{t_j}$, we map it into G' by adding (u', v') into E' (lines 6-7). We clear E_{t_j} and run algorithms to find SCCs on G' (line 8). For each SCC found (line 9), we iterate the internal edges and put the corresponding original edges into E_{t_j} (lines 10-11). Hence, for the next anchored start time $t_i + 1$, those edges will only be accessed at timestamp t_j, i.e., the effective time at t_i. Afterward, we find the RES edges in E' which can connect all pairs of vertices in the SCC with bidirectional paths via Algorithm 7 (line 12). We add the corresponding original edges of them into $\Phi_{[t_i,t_j]}$ and shrink the SCC in G' into a new vertex (line 13). After processing every $t_j \in [t_i, t_{max}]$, we can derive $\Phi_{[t_i,t_{max}]}$ and the corresponding edges in R (lines 14-15).

Lemma 22. *Algorithm 9 costs $O(mt_{max} + nt_{max}^2)$ time.*

Proof. According to Lemma 21, each edge is processed $O(t_{max})$ times. Meanwhile, each vertex is processed $O(t_{max}^2)$ times. Hence, the overall time cost is bounded by $O(mt_{max} + nt_{max}^2)$. □

Additionally, when a new edge $(u, v, t_{max} + 1)$ appears, the RES-index can be maintained without recomputing from scratch. Specifically, we can revise Algorithm 9. In the outer loop (line 1), we range variable t_i from 0 to $(t_{max} + 1)$. In the inner loop, when $t_i \in [0, t_{max}]$, the variable t_j is fixed as $(t_{max} + 1)$, and instead of initializing the RES by lines 4-5, we reconstruct $\Phi_{[t_i,t_{max}]}$ from the current RES-index, initialize E' as $E_{[t_i,t_{max}]}$, and run the steps in lines 8-15; when $t_i = t_{max} + 1$, we follow the original procedure.

Algorithm 9: RES-construct(G)

Input: a directed temporal graph $G = (V, E)$;
Output: the index $R(t_i, t_j)$ for all $0 \le t_i \le t_j \le t_{max}$;

```
for t_i ← 0, ··· , t_max do
    G' ← (V', E') where V' ← V, E' ← ∅;
    for t_j ← t_i, ··· , t_max do
        Φ_[t_i,t_j] ← ∅;
        if t_j > t_i then Φ_[t_i,t_j] ← Φ_[t_i,t_j−1];
        for e = (u, v, t_j) ∈ E_{t_j} do
            Map u, v to u', v'; E' ← E' ∪ {e' = (u', v')};
        Clear E_{t_j} and run algorithms to find SCCs on G';
        for each SCC in G' do
            for each internal edge e' in the SCC do
                e ← the original edge of e'; Add e into E_{t_j};
            Find RES edges E_S for the SCC by calling Algorithm 7;
            Add E_S to Φ_[t_i,t_j] and shrink the SCC into a vertex;
    for e ∈ Φ_[t_i,t_max] \ Φ_[t_i−1,t_max] do t_l(e) ← t_i;
    for e ∈ Φ_[t_i−1,t_max] \ Φ_[t_i,t_max] do add e into R(t_l(e), t_i − 1);
```

5 EXPERIMENTS

We now present the experimental results. We first discusses the setup in Section 5.1. Then we report the efficiency results in Sections 5.2 and 5.3. We further motivate the window-CC/SCC by a case study of anomaly DBLP data detection in Section 5.4. Our codes are available in two GitHub repositories including window-CC[2] and window-SCC[3] solutions, respectively.

5.1 Setup

Datasets	Undirected temporal graphs						Directed temporal graphs					
	contact (CT)	facebook (FB)	mit (MIT)	youtube (YT)	dblp_coauthor (DA)	wikipedia (WK)	bitcoin (BTC)	dblp-cite (DC)	CollegeMsg (CM)	email-eu (EE)	flickr (FK)	amazon (AM)
n	275	63,732	97	3,223,590	1,824,702	1,870,710	5,881	12,590	1,899	986	2,302,926	19,546,448
m	28,244	817,035	1,086,404	9,375,374	29,487,744	39,953,145	35,592	49,759	59,835	332,334	33,140,017	52,885,992
t_{max}	15,661	333,923	33,451	202	76	2,197	35,591	29	58,910	207,879	133	272
size (bytes)	225,952	6,536,280	8,691,232	75,002,992	235,901,952	319,625,160	284,736	396,632	478,680	2,658,672	265,120,136	423,087,936

Table 3. Datasets used in our experiments.

In our experiments, we use 12 real-world temporal graphs from SNAP [32] and KONECT project [30]. Table 3 provides the statistics of each graph, where n and m are the numbers of vertices and edges respectively, t_{max} is the maximum timestamp (counting from 0), and *size (bytes)* is the memory cost (in bytes) of the graph.

To measure the query efficiency of each dataset, for each dataset, we generate 1000 queries with the size of query window (i.e., $t_e - t_s$) as $0.8t_{max}$ where t_s is selected randomly. Then we execute the 1000 queries sequentially and compute the average time cost of the queries. Besides, the indexing time and the average query time of each dataset is measured, respectively. All algorithms

[2]https://github.com/ForwardStar/spannedCC
[3]https://github.com/ForwardStar/spannedSCC

are implemented in C++, compiled with the g++ compiler at -O3 optimization level, and run on a Linux machine with an Intel Xeon 2.40GHz CPU and 128GB RAM.

5.2 Efficiency on undirected temporal graphs

5.2.1 Query processing. Figure 11 shows the average query time of three query algorithms on all undirected datasets. Note that the results of the U-baseline-query algorithm are not reported for FB and WK datasets since it raises an OOM (out-of-memory) exception for the huge space cost. In the figures and tables, we use "N/A" to denote that the index construction could not be finished within 72 hours or raised an OOM exception. Clearly, our index-based query algorithms are faster than the U-online solution. Besides, we see that the larger ratio of m over n results in a larger speedup, which is in line with the time complexities of these query algorithms as listed in Table 1.

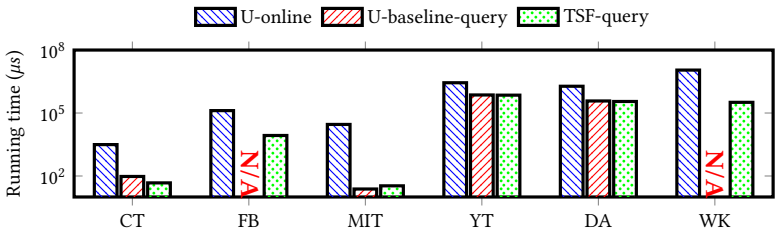


Fig. 11. Average time cost of window-CC queries.

We also evaluate the effect of the query time window size. For each graph, we fix the start time of the query window as $t = 0$ and consider five query time window sizes, i.e., 20%, 40%, 60%, 80% and 100% of t_{max} respectively. We then record the average running time of 1,000 queries with each window size. Figure 12 reports the efficiency results on undirected datasets. When the time window is small, all algorithms cost a similar amount of time. However, when the time window is larger, the U-online algorithm takes more time, while our index-based solutions' time costs do not change much.

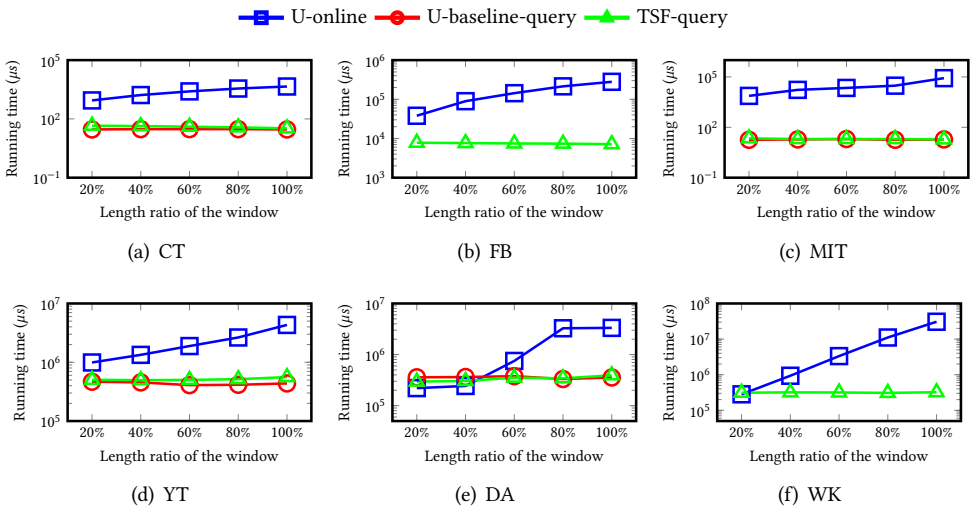
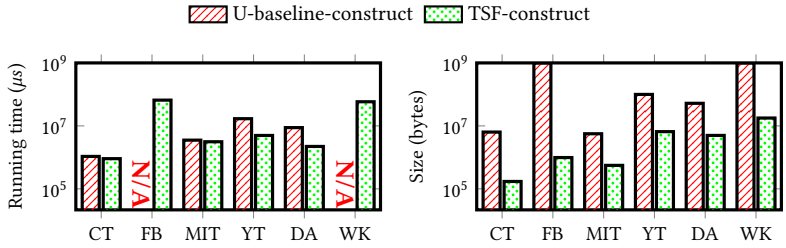


Fig. 12. Effect of the size of the query window on undirected temporal graphs.

5.2.2 Index construction. We compare the efficiency of index construction algorithms on all graphs in Figure 13(a). Clearly, the TSF-index can be built faster than the U-baseline index, since U-baseline-construct costs $O((m + n \log n)t_{max})$ time while TSF-construct needs $O(mt_{max})$ time. Besides, we report the space cost of each index on all graphs in Figure 13(b). We see that the space cost of TSF-index is much less than that of U-baseline-index, because their space costs are bounded by $O(m)$ and $O(n \log n \cdot t_{max})$, respectively.



(a) Construction time (b) Index space cost

Fig. 13. Index construction time and index space cost.

Besides, we test the scalability of the index construction algorithms. Specifically, for each graph, we build four subgraphs using edges with timestamps in $[0, f \cdot t_{max}]$ with f=25%, 50%, 75%, and 100%, respectively. Table 4 depicts the indexing time and space costs on these subgraphs. We can see that TSF-index scales better than the U-baseline index for its optimized time and space complexities.

Metrics	Construction time (μs)								Index space cost (bytes)							
Indices	U-baseline-construct				TSF-construct				U-baseline-construct				TSF-construct			
f	25%	50%	75%	100%	25%	50%	75%	100%	25%	50%	75%	100%	25%	50%	75%	100%
CT	349K	1.1M	2.1M	3.6M	197K	709K	1.6M	2.8M	5.3M	12M	26M	51M	70K	127K	177K	226K
FB	N/A	N/A	N/A	N/A	117M	432M	9.9B	17B	N/A	N/A	N/A	N/A	1.1M	1.8M	2.5M	3.2M
MIT	2.9M	7.6M	15M	22M	1.9M	5.7M	12M	18M	12M	22M	35M	43M	489K	859K	1.2M	1.4M
YT	19M	44M	96M	221M	1.4M	4.8M	15M	36M	244M	626M	1.5B	3.1B	16M	23M	33M	54M
DA	1.3M	4.0M	15M	83M	7.8K	47K	1.1M	11M	187K	10M	195M	1.2B	6.6K	220K	4.3M	36M
WK	57M	N/A	N/A	N/A	2.9M	36M	232M	15B	429M	N/A	N/A	N/A	2.5M	19M	78M	237M

Table 4. Scalability test of indexing time and size.

We also evaluate the efficiency of index maintenance. Specifically, we first build the U-baseline index and TSF-index with edges in $[0, 0.8t_{max}]$. Then, we update the indices with the remaining edges in $(0.8t_{max}, t_{max}]$. Table 5 shows the average update time of each new edge on each dataset. Clearly, the maintenance of the indices is much faster than rebuilding the indices from scratch.

Algorithms	U-baseline-index						TSF-index					
Datasets	CT	FB	MIT	YT	DA	WK	CT	FB	MIT	YT	DA	WK
Time (μs)	234	N/A	257	59	2	N/A	253	2K	381	12	1	40

Table 5. The average update time of each new edge.

In addition, the above index-based query algorithms' efficiency is measured without considering the index construction time, so they may not be reasonable for scenarios that need to consider the indexing time and query time in a collective manner. Thus, we also evaluate the number of TSF-index-based queries required to amortize the index construction time cost, and report the minimum numbers of such queries in Table 6. For each dataset, when the total number of queries exceeds the number in Table 6, the TSF-query algorithm will be faster than the U-online algorithm. Note that

the results of the U-baseline-index are skipped since it fails to build the index on some datasets. We can observe that the numbers of such queries are not very large, and thus the TSF-query algorithm performs better than the U-online algorithm for applications which may have large numbers of queries.

	Undirected graph						Directed graph					
Datasets	CT	FB	MIT	YT	DA	WK	CT	FB	MIT	YT	DA	WK
No.	874	9.1K	479	15	7	69	1.9M	10	1.1M	5.9M	437	102

Table 6. The minimum number of queries required to amortize the index construction time cost.

5.3 Efficiency on directed temporal graphs

5.3.1 Query processing. Figure 14 shows the average query time of three query algorithms on all directed datasets. The results of the D-baseline-query algorithm are not reported for EE dataset since it fails to build the index within 72 hours. Our RES-query algorithm is always faster than the D-online algorithm. The speedup mainly depends on the ratio of m over n as the D-online algorithm and index-based algorithms cost $O(m)$ and $O(n)$ time, respectively. Besides, the D-baseline-query algorithm is slightly faster than the RES-query algorithm because the retrieved edge set of RES-index is twice the number of D-baseline-index, and the RES-query algorithm needs to run an SCC algorithm while the D-baseline-query algorithm does not, although they have the same theoretical time complexities.

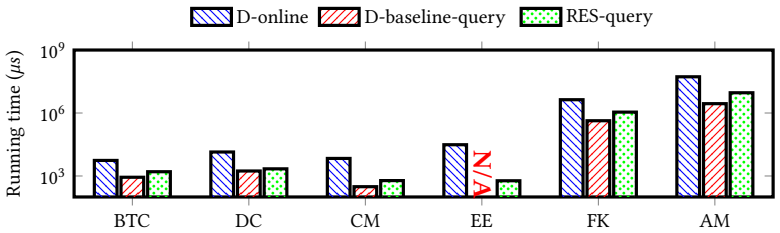


Fig. 14. Average time cost of window-SCC queries.

We also evaluate the effect of the query time window size by setting experiments similar to those of undirected temporal graphs in Section 5.2.1. Figure 15 depicts the results on directed datasets. Clearly, for any size of the query windows, our index-based solutions always perform better than the D-online solution.

5.3.2 Index construction. We report the efficiency of the index construction algorithms on all datasets in Figure 16(a). Notice that for EE dataset with a large t_{max} value, the D-baseline-construct algorithm fails to build the index within 72 hours while the RES-construct algorithm succeeds. The reason is that the construction of the D-baseline-index takes $O((m+n)t_{max}^2)$ time, while the time complexity of the RES-index is smaller than it. Besides, we compare the space cost of indices in Figure 16(b). Clearly, the RES-index costs much less space than the D-baseline-index, since its space cost $O(m)$ is linear to the size of the graph, while the D-baseline-index needs $O(nt_{max})$ space.

Besides, we test the scalability of index construction algorithms by setting experiments similar to those for undirected temporal graphs in Section 5.3.2. Table 7 reports the time and space costs for different f values. We can observe that the RES-index scales better than D-baseline-index because of its optimized time and space complexities.

We also evaluate the maintenance efficiency using similar settings for the efficiency evaluation of index maintenance on the undirected graph. Table 8 shows the average update time of each new

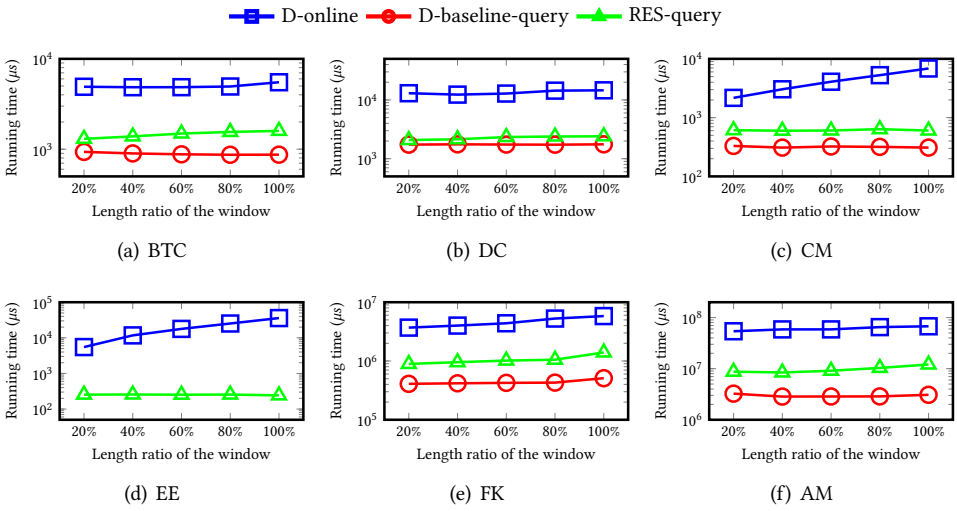


Fig. 15. Effect of the size of the query window on directed temporal graphs.

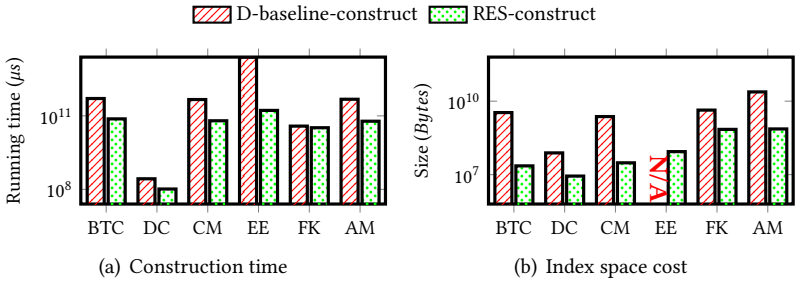


Fig. 16. Index construction time and size.

Metrics	Construction time (μs)								Index space cost (bytes)							
Indices	D-baseline-construct				RES-construct				D-baseline-construct				RES-construct			
f	25%	50%	75%	100%	25%	50%	75%	100%	25%	50%	75%	100%	25%	50%	75%	100%
BTC	8.6B	34B	72B	152B	95M	786M	3.2B	6.3B	419M	838M	1.3B	1.7B	103K	200K	303K	396K
DC	23K	101K	291K	523K	2.8K	14K	54K	106K	806K	1.6M	2.3M	3.1M	2.3K	13K	42K	80.1K
CM	8.3B	31B	65B	130B	149M	895M	2.7B	4.7B	224M	448M	672M	896M	132K	302K	461K	638K
EE	40B	137B	N/A	N/A	1.8B	6.6B	14B	24B	418M	8.4B	N/A	N/A	891K	1.9M	2.8M	3.7M
FK	144M	466M	1.2B	2.1B	54K	454M	814M	1.6B	627M	1.3B	1.9B	2.5B	42K	79M	100M	121M
AM	2.1B	9.1B	28B	136B	16M	146M	603M	4.4B	11B	22B	33B	43B	1.9M	7.7M	25M	129M

Table 7. Scalability test of indexing time and size.

edge on each dataset. We see that the maintenance of the indices is much faster than rebuilding the indices from scratch.

In addition, we also evaluate the number of index-based queries that are required to amortize the index construction time cost. Table 6 presents the minimum numbers of such queries by using the RES-index on the directed datasets. Note that the results of the D-baseline index are skipped

Algorithms	D-baseline-index						RES-index					
Datasets	BTC	DC	CM	EE	FK	AM	BTC	DC	CM	EE	FK	AM
Time (μs)	5.6M	73	2.7M	N/A	40	2.8K	3.7M	108	654K	973K	33	3.5K

Table 8. The average update time of each new edge.

since it fails to build the index on some datasets. We see that if t_{max} is not too large, the RES-query algorithm is more efficient when the number of queries is large.

5.4 A case study of anomaly DBLP data detection

Consider the DBLP-citation (DC) graph, where each vertex is a paper and each edge indicates one paper cites the other, and the timestamps are publishing time. Strictly speaking, there should be no SCC in the graph since it implies wrong citations, i.e., a paper published in the year t may cite another paper published in year t' with $t' > t$, but in practice some anomaly cases may exist somehow.

By varying the query time windows of window-SCC queries, we identify two groups of window-SCCs, which reveal the existence of some "wrong" citation data. The first group of window-SCCs has time windows of one or two years, while the second group has time windows of three or more years. With a careful investigation, we find that for the first group, the paper citations are actually correct, because some papers may cite other preprint papers which are published in the following one or two years. However, for the second group, we do find some erroneous data. For example, the webpage of DBLP website[4] shows that the SIGMOD paper [21] published in 1989 cites another paper [38] published in 2011. We download the SIGMOD paper and find that it does cite that paper [38], which was actually published in 1986. Hence, the DBLP website does not present the year of reference correctly.

6 RELATED WORKS

6.1 CC queries on static graphs

As a fundamental problem in network science, the computation of CC has received plenty of attention, and most of the existing works focus on static undirected graphs. Computing CCs in a static graph can be achieved in linear time regarding the input graph size by BFS or DFS [31]. Recently, to process large graphs, some parallel approaches have been proposed. Earlier solutions considered the PRAM model [3], but their implementations are often complex and inefficiently matching the programming models of the current distributed frameworks. To solve this problem, distributed CCs computation algorithms based on MapReduce [41, 44] and Pergel [16, 46] have been proposed, and they mainly focus on reducing the total data communication cost and the number of iterations.

For the directed graph, computing the SCCs is known to take $O(m + n)$ time using Tarjan algorithm [47], Kosaraju-Sharir algorithm [49], or Path-based algorithm [19], which are based on DFS. However, these serial algorithms are inefficient for large graphs since the computation of DFS is known P-complete [45]. As a result, many parallel algorithms have been proposed to speed up the computation of SCCs, such as Forward-Backward-based algorithms [17, 26, 48], GPU-based algorithms [34, 54], and parallel randomized DFS algorithm [36]. In addition, distributed algorithms have been studied for large directed graphs [28, 57].

Nevertheless, the existing works above mainly focus on conventional static graphs, so their solutions cannot be applied to processing large temporal graphs due to the temporal edges.

[4]https://dblp.org/rec/conf/sigmod/GraefeW89.html (click "load references" checkbox)

6.2 CC queries on temporal graphs

Recently, a few CC queries have been studied in temporal graphs, where each edge has a timestamp denoting the interaction time. For example, Akrida and Spirakis [1], Vernet et al. [51] proposed the persistent CC model for temporal graphs, which is a set of vertices that are connected in each timestamp of a time interval. Bhadra and Ferreira [6, 7] introduced the temporal CC model, which is a set of vertices such that each pair of vertices is connected by a path with edges of increasing timestamps. Nevertheless, these works fail to capture the relationships between entities involving the same group or activity with no time-respecting path connecting them. To the best of our knowledge, our work is the first one that studies window-CC and window-SCC queries in temporal graphs.

Another kind of related but different graph model is the dynamic graph, where an edge in it may appear in a large time interval while an edge of a temporal graph only appears at a single timestamp. Some works have studied CC queries on dynamic graphs [5, 25, 43, 50]. A dynamic graph can be transformed into a temporal graph by introducing artificial timestamps, but it may lead to a huge number of new edges and high time cost if using our algorithms.

In the literature, to process temporal and dynamic graphs, many different types of algorithms have been developed, including streaming algorithms [40], semi-streaming algorithms [15], sliding window-based algorithms [10], and incremental graph algorithms [13]. The high-level ideas of building our indices are somewhat similar to the ideas in the above works. That is, when constructing the indices, we anchor each start time and then process the temporal edges incrementally, during which time windows with the same start time have been considered. However, these works do not build indices to compactly organize all the window-CCs of a temporal graph, for supporting CC/SCC queries with arbitrary time windows.

In addition, recently some other queries have many studied on temporal graphs. Wu et al. [55] studied the shortest path problem in temporal graphs. Wen et al. [53] defined a new reachability concept called span-reachability in temporal graphs and proposed index-based solutions. Gurukar et al. [24] computed the communication motifs in temporal graphs. Some cohesive subgraph queries have been studied in temporal graphs, such as k-core queries [35, 56, 59], quasi-clique query [58], and dense subgraph query [37].

7 CONCLUSION

In this paper, for the first time, we introduce the concepts of window-CCs and window-SCCs on undirected and directed temporal graphs, respectively. We then study the queries of window-CC and window-SCC by developing several efficient index-based query solutions. The space costs of the best indices are linear to the sizes of the temporal graphs. The extensive experimental evaluation on 12 real-world datasets demonstrates the high efficiency and effectiveness of the proposed solutions. In the future, we will develop distributed index construction algorithms, which would be useful for very large temporal graphs containing billions of edges. In the future, we will implement our algorithms by using a distributed computing platform (e.g., Pregel), which would be very useful when the temporal graph is too large to be kept by a single machine.

Acknowledgements. This work was supported in part by NSFC under Grants 62102341 and 62202412, Basic and Applied Basic Research Fund in Guangdong Province under Grants 2022A1515010166 and 2023A1515011280, Guangdong Talent Program under Grant 2021QN02X826, and Shenzhen Science and Technology Program under Grants JCYJ20220530143602006 and ZDSYS 20211021111415025.

REFERENCES

[1] Eleni C Akrida and Paul G Spirakis. 2019. On verifying and maintaining connectivity of interval temporal networks. *Parallel Processing Letters* 29, 02 (2019), 142–154.

[2] Géraud Allard, Philippe Jacquet, and Bernard Mans. 2006. Routing in Extremely Mobile Networks. In *Challenges in Ad Hoc Networking*, K. Al Agha, I. Guérin Lassous, and G. Pujolle (Eds.). 129–138.

[3] Baruch Awerbuch and Yossi Shiloach. 1987. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Trans. Comput.* 36, 10 (1987), 1258–1263.

[4] Scott Beamer, Aydin Buluç, Krste Asanovic, and David Patterson. 2013. Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search. In *2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum*. 1618–1627.

[5] Aaron Bernstein, Maximilian Probst, and Christian Wulff-Nilsen. 2019. Decremental strongly-connected components and single-source reachability in near-linear time. In *ACM SIGACT*. 365–376.

[6] Sandeep Bhadra and Afonso Ferreira. 2003. Complexity of Connected Components in Evolving Graphs and the Computation of Multicast Trees in Dynamic Networks. In *Ad-Hoc, Mobile, and Wireless Networks*, Samuel Pierre, Michel Barbeau, and Evangelos Kranakis (Eds.). 259–270.

[7] Sandeep Bhadra and Afonso Ferreira. 2012. Computing multicast trees in dynamic networks and the complexity of connected components in evolving graphs. *Journal of Internet Services and Applications* 3, 3 (2012), 269–275.

[8] Tianchi Cai, Daxi Cheng, Chen Liang, Ziqi Liu, Lihong Gu, Huizhi Xie, Zhiqiang Zhang, Xiaodong Zeng, and Jinjie Gu. 2021. LinkLouvain: Link-Aware A/B Testing and Its Application on Online Marketing Campaign. In *Database Systems for Advanced Applications*, Christian S. Jensen, Ee-Peng Lim, De-Nian Yang, Wang-Chien Lee, Vincent S. Tseng, Vana Kalogeraki, Jen-Wei Huang, and Chih-Ya Shen (Eds.). Springer International Publishing, Cham, 499–510.

[9] Ananta Chowdhury, Jessica Sharmin Rahman, and Md. Shiplu Hawlader. 2016. Well-connectedness - a novel measure for improving protein complex detection from PPI network. In *2016 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*. 1–6.

[10] Michael S. Crouch, Andrew McGregor, and Daniel Stubbs. 2013. Dynamic Graphs in the Sliding-Window Model. In *Algorithms – ESA 2013*, Hans L. Bodlaender and Giuseppe F. Italiano (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–348.

[11] G. Cybenko, T. G. Allen, and J. E. Polito. 1989. Practical Parallel Union-Find Algorithms for Transitive Closure and Clustering. *Int. J. Parallel Program.* 17, 5 (oct 1989), 403–423.

[12] Aleksander Fabijan, Pavel Dmitriev, Helena Holmstrom Olsson, and Jan Bosch. 2018. Online Controlled Experimentation at Scale: An Empirical Survey on the Current State of A/B Testing. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 68–72. https://doi.org/10.1109/SEAA.2018.00021

[13] Wenfei Fan and Chao Tian. 2022. Incremental Graph Computations: Doable and Undoable. *ACM Trans. Database Syst.* 47, 2, Article 6 (may 2022), 44 pages. https://doi.org/10.1145/3500930

[14] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A survey of community search over big graphs. *The VLDB Journal* 29, 1 (2020), 353–392.

[15] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. 2004. On Graph Problems in a Semi-streaming Model. In *Automata, Languages and Programming*, Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 531–543.

[16] Xing Feng, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Computing connected components with linear communication cost in pregel-like systems. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 85–96.

[17] Lisa K Fleischer, Bruce Hendrickson, and Ali Pınar. 2000. On identifying strongly connected components in parallel. In *International Parallel and Distributed Processing Symposium*. Springer, 505–511.

[18] Andrew D. Fox, Benjamin J. Hescott, Anselm C. Blumer, and Donna K. Slonim. 2011. Connectedness of PPI network neighborhoods identifies regulatory hub proteins. *Bioinformatics* 27, 8 (2011), 1135–1142.

[19] Harold N. Gabow. 2000. Path-based depth-first search for strong and biconnected components. *Inform. Process. Lett.* 74, 3 (2000), 107–114.

[20] Harold N. Gabow and Eugene W. Myers. 1978. Finding All Spanning Trees of Directed and Undirected Graphs. *SIAM J. Comput.* 7, 3 (1978), 280–287.

[21] Goetz Graefe and Karen Ward. 1989. Dynamic query evaluation plans. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*. 358–366.

[22] R.L. Graham and Pavol Hell. 1985. On the History of the Minimum Spanning Tree Problem. *Annals of the History of Computing* 7, 1 (1985), 43–57.

[23] Huan Gui, Ya Xu, Anmol Bhasin, and Jiawei Han. 2015. Network A/B Testing: From Sampling to Estimation. In *Proceedings of the 24th International Conference on World Wide Web* (Florence, Italy) *(WWW '15)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 399–409. https://doi.org/10.1145/

2736277.2741081
[24] Saket Gurukar, Sayan Ranu, and Balaraman Ravindran. 2015. Commit: A scalable approach to mining communication motifs from dynamic networks. In *SIGMOD*. 475–489.
[25] Chirayu Anant Haryan, G Ramakrishna, Kishore Kothapalli, and Dip Sankar Banerjee. 2022. Shared-Memory Parallel Algorithms for Fully Dynamic Maintenance of 2-Connected Components. In *IPDPS*. IEEE, 1195–1205.
[26] Sungpack Hong, Nicole C Rodia, and Kunle Olukotun. 2013. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11.
[27] Philippe Jacquet and Bernard Mans. 2007. Routing in Intermittently Connected Networks: Age Rumors in Connected Components. In *Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PerComW'07)*. 53–58.
[28] Yuede Ji, Hang Liu, and H Howie Huang. 2018. ispan: Parallel identification of strongly connected components with spanning trees. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 731–742.
[29] Udayan Khurana and Amol Deshpande. 2013. Efficient snapshot retrieval over historical graph data. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 997–1008.
[30] Jérôme Kunegis. 2013. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion*. 1343–1350.
[31] Charles Eric Leiserson, Ronald L Rivest, Thomas H Cormen, and Clifford Stein. 1994. *Introduction to algorithms*. Vol. 3. MIT press.
[32] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1.
[33] Vincent Levorato and Coralie Petermann. 2011. Detection of Communities in Directed Networks based on Strongly p-Connected Components. *Proceedings of the 2011 International Conference on Computational Aspects of Social Networks, CASoN'11*.
[34] Guohui Li, Zhe Zhu, Zhang Cong, and Fumin Yang. 2014. Efficient decomposition of strongly connected components on GPUs. *Journal of Systems Architecture* 60, 1 (2014), 1–10.
[35] Rong-Hua Li, Jiao Su, Lu Qin, Jeffrey Xu Yu, and Qiangqiang Dai. 2018. Persistent community search in temporal networks. In *ICDE*. IEEE, 797–808.
[36] Gavin Lowe. 2016. Concurrent depth-first search algorithms based on Tarjan's algorithm. *International Journal on Software Tools for Technology Transfer* 18, 2 (2016), 129–147.
[37] Shuai Ma, Renjun Hu, Luoshu Wang, Xuelian Lin, and Jinpeng Huai. 2017. Fast computation of dense temporal subgraphs. In *ICDE*. IEEE, 361–372.
[38] Stuart E Madnick and Meichun Hsu. 1986. INFOPLEX, research in a high-performance database computer. (1986).
[39] Fredrik Manne and Md. Mostofa Ali Patwary. 2009. A Scalable Parallel Union-Find Algorithm for Distributed Memory Computers. 186–195.
[40] Andrew McGregor. 2014. Graph Stream Algorithms: A Survey. *SIGMOD Rec.* 43, 1 (may 2014), 9–20. https://doi.org/10.1145/2627692.2627694
[41] Lu Qin, Jeffrey Xu Yu, Lijun Chang, Hong Cheng, Chengqi Zhang, and Xuemin Lin. 2014. Scalable big graph processing in mapreduce. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 827–838.
[42] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.
[43] Léo Rannou, Clémence Magnien, and Matthieu Latapy. 2020. Strongly connected components in stream graphs: Computation and experimentations. In *International Conference on Complex Networks and Their Applications*. Springer, 568–580.
[44] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. 2013. Finding connected components in map-reduce in logarithmic rounds. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 50–61.
[45] John H Reif. 1985. Depth-first search is inherently sequential. *Inform. Process. Lett.* 20, 5 (1985), 229–234.
[46] Semih Salihoglu and Jennifer Widom. 2014. Optimizing Graph Algorithms on Pregel-like Systems. *Proceedings of the VLDB Endowment* 7, 7 (2014).
[47] M. Sharir. 1981. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications* 7, 1 (1981), 67–72.
[48] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 550–559.
[49] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

[50] David Tench, Evan West, Victor Zhang, Michael A. Bender, Abiyaz Chowdhury, J. Ahmed Dellas, Martin Farach-Colton, Tyler Seip, and Kenny Zhang. 2022. GraphZeppelin: Storage-Friendly Sketching for Connected Components on Dynamic Graph Streams. In *SIGMOD*. ACM, 325–339.
[51] Mathilde Vernet, Yoann Pigné, and Eric Sanlaville. 2022. A study of connectivity on dynamic graphs: computing persistent connected components. *4OR* (04 2022).
[52] Dong Wen, Yilun Huang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2020. Efficiently Answering Span-Reachability Queries in Large Temporal Graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1153–1164.
[53] Dong Wen, Bohua Yang, Ying Zhang, Lu Qin, Dawei Cheng, and Wenjie Zhang. 2022. Span-reachability querying in large temporal graphs. *The VLDB Journal* 31, 4 (2022), 629–647.
[54] Anton Wijs, Joost-Pieter Katoen, and Dragan Bošnački. 2016. Efficient GPU algorithms for parallel decomposition of graphs into strongly connected and maximal end components. *Formal Methods in System Design* 48, 3 (2016), 274–300.
[55] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path problems in temporal graphs. *Proceedings of the VLDB Endowment* 7, 9 (2014), 721–732.
[56] Huanhuan Wu, James Cheng, Yi Lu, Yiping Ke, Yuzhen Huang, Da Yan, and Hejun Wu. 2015. Core decomposition in large temporal graphs. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 649–658.
[57] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. 2014. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1821–1832.
[58] Yi Yang, Da Yan, Huanhuan Wu, James Cheng, Shuigeng Zhou, and John CS Lui. 2016. Diversified temporal subgraph pattern mining. In *SIGKDD*. 1965–1974.
[59] Michael Yu, Dong Wen, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2021. On Querying Historical K-Cores. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2033–2045.

Received October 2022; revised January 2023; accepted February 2023