Querying Cohesive Subgraphs in Temporal Graphs

YINYU LIU, Harbin Institute of Technology, Shenzhen, China
KAIQIANG YU*, State Key Laboratory of Novel Software Technology, Nanjing University, China
SHENGXIN LIU*, Harbin Institute of Technology, Shenzhen, China
CHENG LONG, Nanyang Technological University, Singapore
XUN ZHOU, Harbin Institute of Technology, Shenzhen, China

Temporal graphs are widely used to model dynamic interactions across diverse domains, including social networks, biology, and e-commerce. Querying historical cohesive subgraphs, defined as extracting cohesive subgraphs from the graph snapshot induced by a given time interval, is a fundamental task in temporal graph analytics. Yet, current index-based methods are typically designed for a single model. This specialization yields high performance for a specific target but cannot be reused for others, which limits extensibility. In this paper, we propose a *unified* index-based framework to address this limitation. We focus on a generalized class of *Historical CSMs (His-CSMs)*, which is characterized by two structural properties: *non-overlapping* (resultant subgraphs are vertex-disjoint) and *monotonic* (validity in a time interval implies validity in any super interval). This class includes prominent models such as historical connected components (His-CC) and historical k-cores (His-Core). To address the *His-CSM* query problem, we reduce queries for different CSMs to a single canonical query, the *spanning connected component query*, and design an index-based solution to answer this query efficiently. Building on this framework, we provide concrete instantiations for both His-CC and His-Core queries. Extensive experiments on real-world temporal graphs show that our method outperforms state-of-the-art baselines, achieving up to 60× speedups for His-CC and up to 100× speedups for His-Core.

CCS Concepts: • **Theory of computation** → **Dynamic graph algorithms**; • **Mathematics of computing** → **Graph algorithms**; • **Information systems** → *Temporal data.*

Additional Key Words and Phrases: Temporal graphs, Cohesive subgraph, Query processing, Indexing

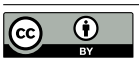
ACM Reference Format:
Yinyu Liu, Kaiqiang Yu, Shengxin Liu, Cheng Long, and Xun Zhou. 2026. Querying Cohesive Subgraphs in Temporal Graphs. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 226 (June 2026), 27 pages. https://doi.org/10.1145/3802103

1 Introduction

Temporal graphs capture evolving interactions between entities, where each edge is annotated with a timestamp indicating when the interaction occurs [5, 34, 36, 51]. Such graphs arise in social networks, financial transactions, biological systems, and many other domains. Querying historical cohesive subgraphs, which extracts cohesive subgraphs from the graph snapshot induced by a given time interval, is a core problem in temporal graph analytics and has been widely

*Corresponding authors.

Authors' Contact Information: Yinyu Liu, Harbin Institute of Technology, Shenzhen, China, 23s151053@stu.hit.edu.cn; Kaiqiang Yu, State Key Laboratory of Novel Software Technology, Nanjing University, China, kaiqiang002@e.ntu.edu.sg; Shengxin Liu, Harbin Institute of Technology, Shenzhen, China, sxliu@hit.edu.cn; Cheng Long, Nanyang Technological University, Singapore, c.long@ntu.edu.sg; Xun Zhou, Harbin Institute of Technology, Shenzhen, China, zhouxun2023@hit.edu.cn.



ACM 2836-6573/2026/6-ART226
https://doi.org/10.1145/3802103

studied [10, 25, 27, 33, 41–43, 52]. Cohesive subgraphs capture dense regions of a graph and often provide an important abstraction for characterizing interaction patterns in real applications [12, 14]. In response to such needs, numerous *cohesive subgraph models (CSMs)* have been proposed in recent years [8, 17, 23, 62], including models tailored specifically to temporal graphs [22, 29, 39, 57, 63]. Consequently, querying historical CSMs across various graph snapshots enables tracking the evolution of dense regions and uncovering long-term behaviors and anomalies that are of practical significance.

Building on this practical significance, the task of querying historical CSMs finds broad applications across multiple disciplines, including financial anomaly detection [4, 12, 37, 61], temporal-aware recommendation systems [38], and infectious disease surveillance [14]. In financial anomaly detection, abrupt changes in tightly interconnected trading groups often correspond to fraudulent activities, market manipulation, or other forms of misconduct; thus, tracking cohesive subgraphs over time can reveal such unusual patterns [4, 37]. In infectious disease surveillance, temporal graphs can model transmission via contact networks between individuals [14]; cohesive subgraphs within specific time windows may correspond to clusters with elevated transmission risk, thereby providing practical insights for outbreak control. Furthermore, prior work [57, 63, 64] has conducted case studies by querying historical CSMs, such as historical k-cores and historical connected components, to analyze the evolution of research groups in coauthorship networks derived from DBLP.

Existing methods and limitations. A straightforward online approach to historical CSMs queries is to collect all edges whose timestamps fall in the query interval and then run a corresponding mining algorithm on the resulting graph. This approach is often inefficient for large graphs or long intervals because it requires scanning and processing all relevant edges. As a result, most prior work targets a single cohesive subgraph model at a time and designs a specialized index that exploits properties of that cohesive subgraph model, including historical connected component (His-CC) [57] and historical k-core (His-Core) queries [48, 50, 63]. These indexes are model specific and do not transfer across models. This lack of generality leads to duplicated engineering effort and complicates system support. Each new historical CSM requires designing a new index from scratch, and graph systems must maintain multiple index schemes to support diverse historical CSM queries.

Our unified framework. To address the aforementioned limitations, it is necessary to answer the following research question:

Can we design a unified index-based framework to support historical CSM queries for various CSMs?

In this paper, we answer this question in the affirmative for a broad class of CSMs that are *non-overlapping* and *monotonic*, which includes connected component, k-core, and k-edge-connected component. We formalize the resulting general problem as the *His-CSM query*. Formally, given a temporal graph $\mathcal{G}$, a cohesive subgraph model $\mathcal{P}$, and a query time interval $[t_s, t_e]$, the His-CSM query seeks to identify all subgraphs within the graph snapshot induced by $[t_s, t_e]$ that satisfy $\mathcal{P}$. This formulation generalizes prior studies [48, 50, 57, 63, 64] and provides a common interface for querying historical cohesive subgraph models.

To address this problem, we develop a *unified* approach that converts various historical cohesive-subgraph queries into *a single canonical form and answers it efficiently via tailored index structures*. Specifically, our framework contains two components:

- *Transformer*. We first reduce His-CSM queries for different CSMs into a single canonical query, the *spanning connected component (Spanning-CC) query*, defined over an *affiliated (spanning) graph*. Solving the Spanning-CC query is equivalent to answering the original His-CSM query, which provides a unifying abstraction.

- *Indexing based on affiliated forests.* We introduce *affiliated forests (AF)* and leverage them to design two unified index structures, **AF-3DT** and **AF-BTAS** for efficient Spanning-CC queries.

This design delivers both generality and efficiency across multiple CSMs. Among our AF-based methods, **AF-BTAS** attains the strongest theoretical and empirical performance.

Concretely, when instantiated for historical connected component (His-CC), **AF-BTAS-CC** builds its index in $O(m \log m)$ time and answers each query in $O(n + \log^2 m)$ time, significantly improving over the state-of-the-art TSF [57] (which requires $O(mt_{\max})$ construction and $O(n + t_s)$ query time), while maintaining the same linear space complexity $O(m)$. Here, m and n denote the numbers of temporal edges and vertices; $t_{\max}$ is the maximum timestamp with $t_{\max} \leq m$; and t_s is the start time of the query interval $[t_s, t_e]$ (bounded by $t_{\max}$). Furthermore, when instantiated for historical k-core (His-Core), **AF-BTAS-Core** also shows improved performance over the state-of-the-art methods PHC and MCTS [50, 63, 64] with comparable index sizes, as summarized in Table 1 and validated by our experiments in Section 4.

Our contributions. Our main contributions are as follows:

- We formulate the problem of querying historical cohesive subgraph models that are non-overlapping and monotonic. (Section 2)
- We propose two unified index-based solutions called **AF-3DT** and **AF-BTAS** for answering His-CSM queries, and we further instantiate **AF-BTAS** for His-CC and His-Core queries. (Section 3)
- We conduct extensive experiments on real-world temporal graphs and show that our methods outperform state-of-the-art baselines for His-CC and His-Core queries. (Section 4)

2 Preliminaries

We focus on an undirected temporal graph $\mathcal{G} = (V, \mathcal{E})$, where V denotes the set of n vertices and $\mathcal{E}$ denotes the set of m *temporal* edges. Here, each temporal edge is represented by a triple (u, v, t), where $u \in V$, $v \in V$, and $t \in \mathbb{N}$ indicates the timestamp at which u interacts with v. Following existing studies [22, 29, 39, 57, 63], we assume that all timestamps are consecutive integers ranging from 1 to $t_{\max}$, which implies $t_{\max} \leq |\mathcal{E}|$.

Given a time window $[t_s, t_e]$, the *projected graph* of $\mathcal{G}$ over $[t_s, t_e]$, denoted by $G_{[t_s,t_e]} = (V, E_{[t_s,t_e]})$, is an undirected graph that includes the set of vertices V and the set of edges $E_{[t_s,t_e]} = \{(u, v) \mid (u, v, t) \in \mathcal{E}, t \in [t_s, t_e]\}$. We remark that projected graphs discard the timestamps and retain only the edges whose timestamps lie in the window. Given a graph $G = (V, E)$ and a vertex set $V_{sub} \subseteq V$, we use $G[V_{sub}]$ to denote the subgraph induced by V_{sub}, i.e., $G[V_{sub}]$ includes the set of vertices V_{sub} and the set of edges $\{(u, v) \in E \mid u, v \in V_{sub}\}$. All subgraphs considered in this paper are by default vertex-induced subgraphs.

A cohesive subgraph is typically considered as a subgraph whose vertices are more densely connected to each other than to the rest of the graph. Let $\mathcal{P}$ be a *cohesive (vertex-induced) subgraph model* (CSM) defined on an undirected graph, e.g., connected component, k-core, clique, k-plex, etc. Given an undirected graph $G = (V, E)$, we write $\mathcal{P}(G)$ for the set of (vertex-induced) subgraphs of G that satisfies $\mathcal{P}$. For a subgraph g of G, if $g \in \mathcal{P}(G)$ (resp. $g \notin \mathcal{P}(G)$), we say that g satisfies (resp. violates) CSM $\mathcal{P}(G)$. In this paper, we focus on CSMs that are both *non-overlapping* and *monotonic*.

Definition 2.1 (Non-overlapping). A CSM $\mathcal{P}$ is said to be *non-overlapping* if and only if for any graph $G = (V, E)$ and any two subgraphs $G[V_1] \in \mathcal{P}(G), G[V_2] \in \mathcal{P}(G)$, the vertex sets V_1 and V_2 are disjoint, i.e., $V_1 \cap V_2 = \emptyset$.

Clearly, for a non-overlapping CSM $\mathcal{P}$, $\mathcal{P}(G)$ consists of at most $|V|$ different subgraphs.

Definition 2.2 (monotonic). A CSM $\mathcal{P}$ is said to be *monotonic* if and only if for any graph $G = (V, E)$, any subgraph $G[V_{sub}] \in \mathcal{P}(G)$ (with $V_{sub} \subseteq V$), and any supergraph $G' = (V', E')$ of G (with $V \subseteq V'$ and $E \subseteq E'$), there exists a subgraph $G'[V'_{sub}]$ in $\mathcal{P}(G')$ such that $V_{sub} \subseteq V'_{sub}$.

Intuitively, for a monotonic CSM $\mathcal{P}$, all vertices in a subgraph satisfying $\mathcal{P}(G)$ remain in the same subgraph satisfying $\mathcal{P}(G)$ after adding additional vertices and/or edges to G. Besides, we note that many CSMs satisfy the non-overlapping and monotonic properties, including connected component and k-core.

Lemma 2.3. *The connected component, k-core, and connected k-core are non-overlapping and monotonic.*

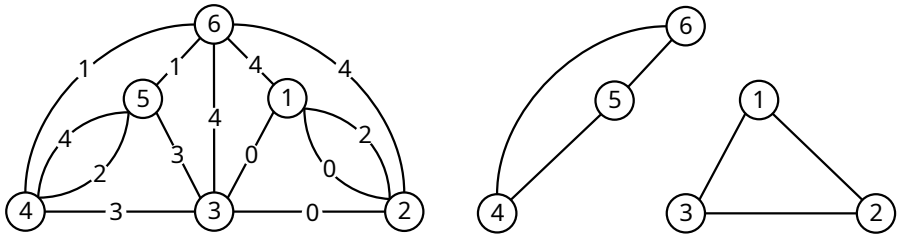
Proof. **k-core.** We show that the k-core is a non-overlapping and monotonic CSM. The k-core of a graph G is defined as a maximal subgraph where every vertex has degree at least k within the subgraph. To show that the k-core is non-overlapping, we prove that the k-core is unique in a given graph G. Assume to the contrary that g_1 and g_2 are two distinct k-cores in a graph G. By the definition of k-core, the union of g_1 and g_2, i.e., $G[V(g_1) \cup V(g_2)]$, where $V(g)$ denotes the vertex set of subgraph g, is clearly k-core, which implies that there exists a larger k-core, contradicting the maximality of the k-core. Thus, k-core is a non-overlapping CSM. We next prove the monotonic property. For any supergraph G' of G, we observe that for any vertex u, the degree of u in G' is at least the degree in G. Then, any vertex that satisfies the k-core condition continues to satisfy the condition in G'. Thus, k-core is monotonic.
Connected component. We then consider the connected component. Recall that a connected component of a graph G is a connected subgraph that is not part of any larger connected subgraph. Assume to the contrary that there are two connected components g_1 and g_2 that are overlapped, i.e., there exists a vertex $v \in V(g_1) \cap V(g_2)$. This implies that $v \in V(g_1)$ and $v \in V(g_2)$. Since $v \in V(g_1)$, by the definition of connected components, v is connected to every other vertex in $V(g_1)$ via a path. Similarly, since $v \in V(g_2)$, v is connected to every other vertex in $V(g_2)$ via a path. This implies that $G[V(g_1) \cup V(g_2)]$ forms a single connected component, contradicting the maximality of g_1 and g_2. Therefore, the connected component is non-overlapping. We next consider the monotonic property. For any supergraph G' of G, a connected component C in G, C is still a connected subgraph in G', which indicates that C belongs to a larger connected component in G'. Thus, the connected component is a monotonic CSM.
Connected k-core. The connected k-core is a maximal connected subgraph where every vertex has a degree of at least k [64]. The case for the connected k-core can be derived in a manner similar to the proof above, and the detail is shown in Appendix [1].

□

For brevity, when we refer to CSMs in this paper we mean CSMs that satisfy both the non-overlapping and monotonic properties. We note that the mentioned CSMs are originally defined on undirected and unweighted graphs. To extend them to temporal graphs, there are several studies in the literature, leading to various temporal versions of the same non-temporal CSMs. Following prior studies [57, 63], we focus on one such extension called *historical-CSMs (His-CSMs)*. Intuitively, a His-CSM is defined based on a CSM $\mathcal{P}$ and a specified time window $[t_s, t_e]$. Formally, we have the following.

Definition 2.4 (Historical CSMs). Given a temporal graph $\mathcal{G}$, a time window $[t_s, t_e]$, and a CSM $\mathcal{P}$, a subgraph $\mathcal{G}[V_{sub}]$ of $\mathcal{G}$ satisfies the $\mathcal{P}$-based His-CSM over $[t_s, t_e]$ if and only if the projected graph of $\mathcal{G}[V_{sub}]$ over $[t_s, t_e]$ satisfies $\mathcal{P}$.

Now, we are ready to formulate the problem studied in this paper.



(a) A temporal graph $\mathcal{G}$ (b) The projected graph $G_{[0,2]}$

Fig. 1. Illustration the basic concepts of temporal graph and projected graph

Problem Definition (Historical Non-overlapping and Monotonic CSM Indexing). *Given a temporal graph $\mathcal{G}$ and a non-overlapping and monotonic CSM $\mathcal{P}$, the goal is to construct an index structure that, for any query time window $[t_s, t_e]$, returns all vertex sets $S \subseteq V$ with $|S| \geq 2$ such that the induced subgraph $G_{[t_s,t_e]}[S]$ satisfies the model $\mathcal{P}$ in the projected graph $G_{[t_s,t_e]}$.*

To illustrate, we take the historical connected 2-core as an example of non-overlapping and monotonic CSMs in Figure 1. Given the temporal graph in Figure 1(a), its historical connected 2-cores across different time intervals are shown in Figure 2. Specifically, each grid corresponds to a time interval and each historical connected 2-core is colored with the same color, e.g., there are two connected 2-cores in the projected graph $G_{[0,2]}$, i.e., $G[\{1, 2, 3\}]$ and $G[\{4, 5, 6\}]$, marked red and blue respectively.

We remark that the Historical-CSM queries have been widely studied recently. However, all focus on one specific type of CSM, i.e., Historical-Connected-Component (or, His-CC) [57] and Historical-k-Core (or, His-Core) [63]. Thus, previously proposed methods cannot be generalized to query other CSMs. In the following, we will propose a *unified* and efficient framework for the Historical-CSM query and instantiate it for His-Core and His-CC queries.

3 Our Unified Index-based Framework

We introduce our *unified* index-based framework, which constructs novel indexes to support the His-CSM query. In general, it consists of two key components. The first is a *transformer*, which transforms the His-CSM query w.r.t. various CSMs in the temporal graph to a unified query formulation, namely *spanning connected component (Spanning-CC) query* in the *affiliated (spanning) graph* (Sections 3.1). Solving the latter query is equivalent to solving the former problem. This guarantees the generality of our framework to answer different His-CSM queries. The second corresponds to two index-based solutions, namely `AF-3DT` and `AF-BTAS`, which are based on a novel concept of *affiliated forests (AF)* and are designed to answer the newly formulated Spanning-CC query (Section 3.2 and Section 3.3). Finally, we instantiate the framework for two models, His-CC and His-Core, and summarize our findings in Section 3.4.

3.1 A Unified Query Formulation

In this section, we show how different His-CSM queries, each defined with respect to a specific CSM, can be transformed into a single and unified query formulation, namely, the spanning-CC query. The key idea is to identify the smallest time intervals during which a pair of vertices belong to the same CSM subgraph, leading to the notion of a *minimal u-v-TW*. In other words, due to the *monotonic* property, any query interval that fully contains a u-v-TW guarantees that u and v remain in the same CSM subgraph. Based on this concept, we introduce an affiliated graph, in which a

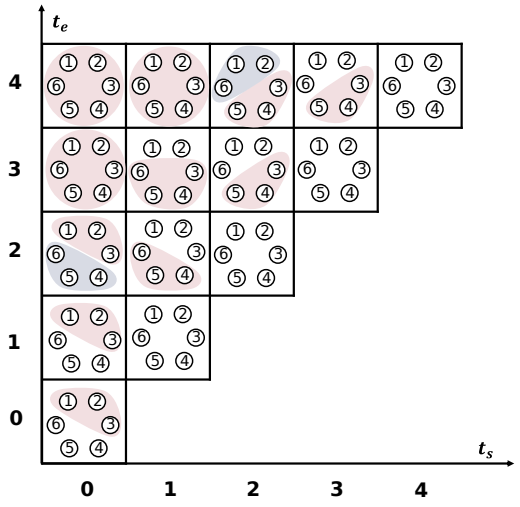


Fig. 2. Illustrating historical connected 2-cores

spanning-CC query corresponds exactly to a His-CSM query in the original temporal graph. We elaborate on the details below.

Minimal u-v-TW. We start with the following definition.

Definition 3.1 (u-v-TW). Given a temporal graph $\mathcal{G} = (V, \mathcal{E})$ and a CSM $\mathcal{P}$. Let u and v be two vertices in V, and $[t_1, t_2]$ be a time window. $[t_1, t_2]$ is a u-v time window (or simply, u-v-TW) w.r.t. $\mathcal{P}$ iff u and v belong to the same subgraph in $\mathcal{P}(G_{[t_1,t_2]})$.

Besides, a u-v-TW $[t_1, t_2]$ is said to be *minimal* iff for any sub-interval $[t'_1, t'_2] \subset [t_1, t_2]$, u and v belong to the different subgraphs in $\mathcal{P}(G_{[t'_1,t'_2]})$. Clearly, there could be multiple minimal u-v-TW between $[0, t_{\max}]$. To illustrate, consider an example in Figure 2 based on a temporal graph in Figure 1(a). The time interval $[1, 2]$ qualifies as a minimal 5-6-TW according to the definition, as these two vertices belong to the same connected 2-core in $[1, 2]$ but not within any of its sub-intervals. Similarly, the minimal 3-4-TWs are $[2, 3]$ and $[3, 4]$. Then, we observe the following lemma.

Lemma 3.2. *Let $[t_1, t_2]$ be a (minimal) u-v-TW w.r.t. $\mathcal{P}$. Any time window $[t'_1, t'_2]$ with $[t_1, t_2] \subseteq [t'_1, t'_2]$ is a u-v-TW w.r.t. $\mathcal{P}$.*

Proof. This can be verified by the monotonic property of $\mathcal{P}$ (Definition 2.2). Assume that u and v belong to a subgraph g in $\mathcal{P}(G_{[t_1,t_2]})$. We note that there exists a subgraph g' of $\mathcal{P}(G_{[t'_1,t'_2]})$ such that g is contained in g' (i.e., $V(g) \subseteq V(g')$ and $E(g) \subseteq E(g')$) since $G_{[t'_1,t'_2]}$ is a supergraph of $G_{[t_1,t_2]}$ and $\mathcal{P}$ satisfies the monotonic property. Therefore, u and v belong to subgraph g' in $\mathcal{P}(G_{[t'_1,t'_2]})$ and $[t'_1, t'_2]$ is a u-v-TW. □

Affiliated spanning graph. We note that due to the *monotonicity* property, any query interval that fully contains a u-v-TW guarantees that u and v remain in the same CSM subgraph. Based on the minimal u-v-TWs, we subsequently define the *affiliated spanning graph* of $\mathcal{G}$, denoted by $\overline{\mathcal{G}}$, as below.

Definition 3.3 (Affiliated spanning graph). Given a temporal graph $\mathcal{G} = (V, \mathcal{E})$ and a CSM $\mathcal{P}$, the *affiliated spanning graph (or affiliated graph)* of $\mathcal{G}$ w.r.t. $\mathcal{P}$ is a "weighted" graph $\overline{\mathcal{G}} = (V, \overline{\mathcal{E}})$, which

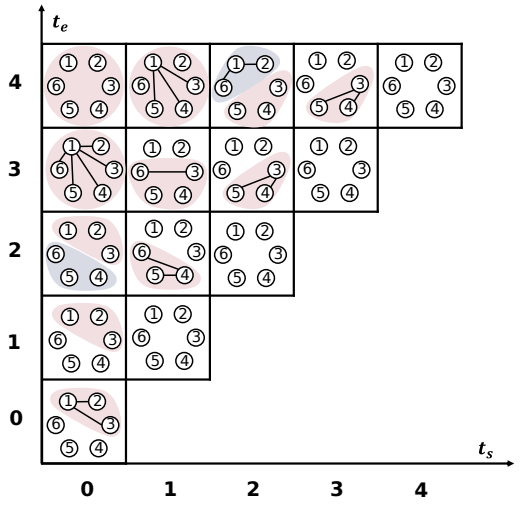


Fig. 3. Illustrating the affiliated graph

includes the set of vertices V and the set of *spanning edges* each of which corresponds to a minimal u-v-TW, formally,

$$\overline{\mathcal{E}} = \{(u, v, t_1, t_2) \mid u, v \in V, [t_1, t_2] \text{ is a minimal } u\text{-}v\text{-TW}\}, \tag{1}$$

and satisfies the following constraint.

Constraint: For any time window $[t_s, t_e]$, two vertices u and v belong to the same subgraph in $\mathcal{P}(G_{[t_s,t_e]})$ if and only if u is reachable to v in $\overline{G}_{[t_s,t_e]}$.

Here, $\overline{G}_{[t_s,t_e]} = (V, \overline{E}_{[t_s,t_e]})$ is the *projected graph* of $\overline{\mathcal{G}}$ over $[t_s, t_e]$, which includes the set of vertices V and the set of edges, i.e.,

$$\overline{E}_{[t_s,t_e]} = \{(u, v) \mid (u, v, t_1, t_2) \in \overline{\mathcal{E}}, [t_1, t_2] \subseteq [t_s, t_e]\}. \tag{2}$$

We note that an affiliated spanning graph $\overline{\mathcal{G}}$ of $\mathcal{G}$ can be built easily by adding a spanning edge for each minimal u-v-TW in $\mathcal{G}$. We can verify this in two directions. First, if u and v belong to the same subgraph in $\mathcal{P}(G_{[t_s,t_e]})$, $[t_s, t_e]$ is a u-v-TW by definition and thus there exists at least one minimal u-v-TW $[t_1, t_2]$ such that $[t_1, t_2] \subseteq [t_s, t_e]$. In this case, $\overline{\mathcal{G}}$ has a spanning edge (u, v, t_1, t_2), and thus u and v are adjacent in $\overline{G}_{[t_s,t_e]}$. Second, u and v are reachable in $\overline{G}_{[t_s,t_e]}$ via a path $\langle u_0 = u, u_1, u_2, ..., u_t = v\rangle$ with $t \geq 1$. For a pair of vertices u_i and u_{i+1} with $0 \leq i \leq t-1$, we note that $\overline{\mathcal{G}}$ has a spanning edge (u_i, u_{i+1}, t_1, t_2) such that $[t_1, t_2] \subseteq [t_s, t_e]$, which indicates a minimal u-v-TW $[t_1, t_2]$. By Lemma 3.2, $[t_s, t_e]$ is a u-v-TW, and thus u_i and u_{i+1} belong to the same subgraph in $\mathcal{P}(G_{[t_s,t_e]})$ by definition, i.e., the *monotonic property*. Finally, vertices in $\{u_0, u_1, ..., u_t\}$ (including u and v) belong to the same subgraph in $\mathcal{P}(G_{[t_s,t_e]})$ due to the *non-overlapping property*.

However, the resulting affiliated graph could be very dense. As a result, queries over a dense affiliated graph may suffer from efficiency issues. To address this, an affiliated graph $\overline{\mathcal{G}}$ can be efficiently refined by removing some redundant spanning edges. (u, v, t_1, t_2). We will discuss more details about the construction and refinement of affiliated graphs in the following sections.

To illustrate, see an affiliated graph in Figure 3, which is constructed based on a temporal graph in Figure 1(a). Each grid in the figure corresponds to a time interval, and an edge within a grid

represents the spanning edge associated with that interval. For example, the edges $(4, 5)$ and $(4, 6)$ appear in the grid with $t_s = 1$ and $t_e = 2$, indicating that spanning edges $(4, 5, 1, 2)$ and $(4, 6, 1, 2)$ are included in the affiliated graph. There are various methods to construct such an affiliated graph, and the construction process for some specific CSM is detailed in Section 3.4.

A unified query formulation: Spanning-CC query. We observe that all vertices belonging to the same subgraph in $\mathcal{P}(G_{[t_s,t_e]})$ will induce a connected component (CC) of $\overline{G}_{[t_s,t_e]}$. Here, a CC of a graph G is a connected (vertex-induced) subgraph that is not contained in any larger connected subgraph. Further, those vertices that do not belong to any subgraph in $\mathcal{P}(G_{[t_s,t_e]})$ have no neighbors in $\overline{G}_{[t_s,t_e]}$, and thus each of these vertices corresponds to a CC of size 1. We summarize the above observation as follows, which can be easily verified based on the definition of affiliated graph.

Lemma 3.4. *For a vertex subset S of V with $|S| \geq 2$, S induces a subgraph $G_{[t_s,t_e]}[S]$ in $\mathcal{P}(G_{[t_s,t_e]})$ if and only if S induces a CC $\overline{G}_{[t_s,t_e]}[S]$ of $\overline{G}_{[t_s,t_e]}$.*

Based on the above lemma, we can answer a Historical-CSM query by solving a unified query formulation called *Spanning-CC* query, formally defined as follows.

Spanning-CC query: Given an affiliated graph $\overline{\mathcal{G}}$ and a time window $[t_s, t_e]$, the spanning-CC query over $\overline{\mathcal{G}}$ aims to find all vertex sets, called spanning-CCs, where each vertex set S includes at least two vertices and induces a CC of $\overline{G}_{[t_s,t_e]}$.

We remark that the formulated spanning-CC query is *CSM-free*, as solving it does not rely on any specific input CSM. In other words, the spanning-CC query algorithm operates purely on the affiliated graph, abstracting away the input CSM. However, the construction of affiliated graphs is still CSM-specific which relies on the properties of the input CSM. Nonetheless, this step can be regarded as a one-time pre-processing for all subsequent queries.

Discussion. We note that the Historical-CC query can be regarded as a special case of the Spanning-CC query (since the former performs on a graph with temporal edges while the latter performs on a graph with spanning edges). As a result, existing methods for answering Historical-CC queries cannot be adopted for solving the newly formulated Spanning-CC query. We will present an index-based method for the latter in the following sections.

3.2 Our Index-based Solution: AF-3DT

Consider an affiliated graph $\overline{\mathcal{G}}$ and a time window $[t_s, t_e]$. To answer the spanning-CC query, a simple *online* approach is to find all CCs in the projected graph $\overline{G}_{[t_s,t_e]}$ using the Depth-First Search (DFS) and the Union-Find data structure. This online method runs in $O(n + \overline{m})$, where $\overline{m} = |\overline{\mathcal{E}}|$, due to the DFS procedure. To improve efficiency, we propose an index-based solution, called AF-3DT (Affiliated-Forest-based 3-D Range Tree Index), which utilizes the novel concept of affiliated forests introduced in Section 3.2.1 together with the classic 3-D range tree. Section 3.2.2 explains the rationale behind AF-3DT and how it answers queries efficiently, while Section 3.2.3 describes the construction of the index structure. Using AF-3DT, the query can be answered in $O(n + \log^2 \overline{m})$. We elaborate on the details below.

We first observe that all CCs in a projected graph form its spanning forest, where each CC corresponds to a spanning tree. Retrieving all CCs from a spanning forest with n vertices can be done optimally in $O(n)$ via DFS (note that a spanning forest consists of at most $n - 1$ edges). Hence, our idea is to *build indices for some spanning edges* in the affiliated graph so as to *efficiently identify the spanning forest of a projected graph $\overline{\mathcal{G}}_{[t_s,t_e]}$ and retrieve all CCs* when answering the spanning-CC query with a time window $[t_s, t_e]$.

3.2.1 Affiliated forests. We start with the following definition.

Definition 3.5 (Affiliated forest). Given an affiliated graph $\overline{\mathcal{G}}$ and a time window $[t_s, t_e]$, an affiliated forest $(V, \mathcal{F}_{t_s,t_e})$ over $[t_s, t_e]$ includes the edge set $\mathcal{F}_{t_s,t_e} \subseteq \{(u, v, t_1, t_2) \in \overline{\mathcal{E}} \mid [t_1, t_2] \subseteq [t_s, t_e]\}$ such that (1) the projected graph of $(V, \mathcal{F}_{t_s,t_e})$ over $[t_s, t_e]$ is a spanning forest of $\overline{G}_{[t_s,t_e]}$, and (2) there exists at most one spanning edge between any two vertices in $(V, \mathcal{F}_{t_s,t_e})$.

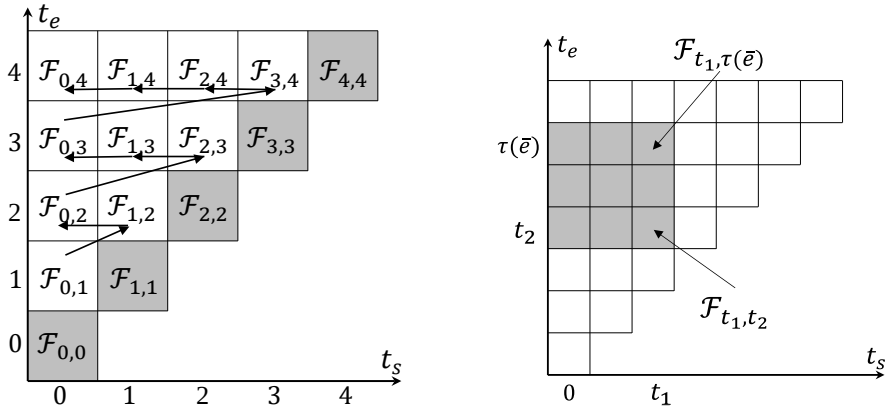
Clearly, an affiliated forest consists of at most $n - 1$ edges. Thus, retrieving edges in $\mathcal{F}_{t_s,t_e}$ will help to quickly find the spanning forest of $\overline{G}_{[t_s,t_e]}$. We then show how to construct an affiliated forest for any possible time window, followed by building indices on them for efficient retrieval. Our method `BuildAF` has two phases, as visualized in Figure 4(a). We give the details of `BuildAF` below.

- **Phase-I: Initialization.** For $0 \le t \le t_{\max}$, let $(V, E_{t,t})$ be an arbitrary spanning forest of $\overline{G}_{[t,t]}$. We add to $\mathcal{F}_{t,t}$ a spanning edge (u, v, t, t) for each edge (u, v) in $E_{t,t}$, formally,
$$\mathcal{F}_{t,t} = \{(u, v, t, t) \mid (u, v) \in E_{t,t}\}. \tag{3}$$
- **Phase-II: Iterative computations.** For $0 \le t_s < t_e \le t_{\max}$, we iteratively compute $\mathcal{F}_{t_s,t_e}$ based on a certain ordering. Specifically, we have $t_{\max}$ rounds with t_e varying from 1 to $t_{\max}$, and in the i-th round (i.e., $t_e = i$), we iteratively compute $\mathcal{F}_{t_s,i}$ for t_s from $i - 1$ to 0. In particular, we compute $\mathcal{F}_{t_s,t_e}$ based on the edges $(u, v, t_s, t_e) \in \overline{\mathcal{E}}$ and two sets, namely $\mathcal{F}_{t_s+1,t_e}$ and $\mathcal{F}_{t_s,t_e-1}$, previously obtained in the ordering or Phase-I, as below.
  - **Step 1.** Initialize $\mathcal{F}_{t_s,t_e}$ by $\mathcal{F}_{t_s+1,t_e}$.
  - **Step 2.** For each edge (u, v, t_1, t_2) in $\mathcal{F}_{t_s,t_e-1}$, if u is not connected to v via a path in $(V, \mathcal{F}_{t_s,t_e})$, we add (u, v, t_1, t_2) to $\mathcal{F}_{t_s,t_e}$.
  - **Step 3.** For each edge (u, v, t_s, t_e) in $\overline{\mathcal{G}}$, if u is not connected to v via a path in $(V, \mathcal{F}_{t_s,t_e})$, we add (u, v, t_1, t_2) to $\mathcal{F}_{t_s,t_e}$.

The refined version of the affiliated graph from Figure 3, computed by `BuildAF`, is presented in Figure 5. In this figure, the grid (x, y) corresponds to an affiliated forest $\mathcal{F}_{x,y}$. For example, $\mathcal{F}_{0,0}$ contains two edges $(1, 2, 0, 0)$ and $(1, 3, 0, 0)$. To obtain $\mathcal{F}_{0,1}$ in `BuildAF`, we first initialize it as $\mathcal{F}_{1,1}$, which contains no edges. Then, for each edge in $\mathcal{F}_{0,0}$, we add (u, v, t_1, t_2) to $\mathcal{F}_{0,1}$ only if u and v are not currently connected in $\mathcal{F}_{0,1}$. In this case, vertices 1 and 2 are not connected, so $(1, 2, 0, 0)$ is added to $F_{0,1}$.

We then prove the correctness of the above construction by induction. For the base cases $\mathcal{F}_{t,t}$ with $0 \le t \le t_{\max}$, $(V, \mathcal{F}_{t,t})$ is clearly an affiliated forest based the construction in Phase-I. We assume that $\mathcal{F}_{i,j}$ with $j \le t_e - 1$ or $t_s + 1 \le i < t_e = j$ obtained in Phase-II (i.e., those preceding $\mathcal{F}_{t_s,t_e}$ in the ordering), including $\mathcal{F}_{t_s,t_e-1}$ and $\mathcal{F}_{t_s+1,t_e}$, are affiliated forests.

For the induction step, we show that $(V, \mathcal{F}_{t_s,t_e})$ is an affiliated forest by contradiction. Suppose that $(V, \mathcal{F}_{t_s,t_e})$ is not an affiliated forest, and let (V, T) be a spanning forest of $\overline{G}_{[t_s,t_e]}$. We can easily derive that $\mathcal{F}_{t_s,t_e}$ does not have multiple spanning edges between any pair of two vertices and its projected graph over $[t_s, t_e]$ is a forest since (1) edges from $\mathcal{F}_{t_s+1,t_e}$ does not form any multiple edge or any cycle, and (2) an edge (u, v, t_1, t_2) or (u, v, t_s, t_e) in Steps 2-3 of Phase-II can be added to $\mathcal{F}_{t_s,t_e}$ only when u is not connected to v in $\mathcal{F}_{t_s,t_e}$. Hence, by assumption, we have $|\mathcal{F}_{t_s,t_e}| < |T|$ and thus there exists at least one pair of vertices u' and v' that are connected in T via an edge (u', v') but remain disconnected in $\mathcal{F}_{t_s,t_e}$. We derive the contradiction by showing that $\overline{G}_{[t_s,t_e]}$ does not have the edge (u', v') since (1) $\overline{G}_{[t_s+1,t_e]}$ (resp. $\overline{G}_{[t_s,t_e-1]}$) does not have the edge (u', v') since otherwise u' is connected to v' in $\mathcal{F}_{t_s+1,t_e}$ (resp. $\mathcal{F}_{t_s,t_e-1}$) and thus $\mathcal{F}_{t_s,t_e}$ based on the construction, and (2) there does not exist spanning edge (u', v', t_s, t_e) in $\overline{\mathcal{G}}$ since otherwise u' is connected to v' in $\mathcal{F}_{t_s,t_e}$.



(a) Illustrating BuildAF (b) Illustrating Lemma 3.6

Fig. 4. Illustrating the affiliated forests where each block corresponds to one affiliated forest. In Figure 4(a), the gray blocks are built in Phase-I while others are built in Phase-II following the ordering indicated by the arrows. In Figure 4(b), $\bar{e}$ is a spanning edge in $\mathcal{F}_{t_1,t_2}$ but not in $\mathcal{F}_{t_1,\tau(\bar{e})+1}$, and $\bar{e}$ appears in those gray blocks.

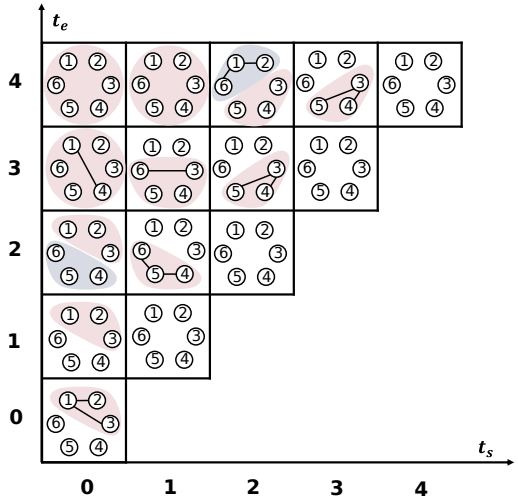


Fig. 5. Illustrating the refined affiliated graph. The entire grid collectively represents the final refined affiliated graph. The edges shown in cell (x, y) are spanning edges (u, v, t_s, t_e) such that $t_s = x$ and $t_e = y$.

We remark that (1) the set of affiliated forests is not unique and can be an arbitrary one, and (2) for those spanning edges that do not appear in the affiliated forests, they are redundant and can be pruned from the affiliated graph, thereby yielding a refined affiliated graph. For example, the refined affiliated graph in Figure 5 consists of a total of 12 spanning edges while the original affiliated graph in Figure 3 has 19 spanning edges. We then show how to build indices over those spanning edges in the affiliated forests.

3.2.2 AF-3DT *index structure and querying process.* Consider the affiliated forests built by BuildAF and a spanning edge $\overline{e} = (u, v, t_1, t_2)$ in the forests. We observe that $\overline{e}$ appears only in the following range of affiliated graphs, as visualized in Figure 4(b).

Lemma 3.6. *Given a spanning edge $\overline{e} = (u, v, t_1, t_2)$. Let $\tau(\overline{e})$ be the timestamp such that $\overline{e} \in \mathcal{F}_{t_1,\tau(\overline{e})}$ and $\overline{e} \notin \mathcal{F}_{t_1,\tau(\overline{e})+1}$. An affiliated forest $(V, \mathcal{F}_{x,y})$ built by* BuildAF *contains $\overline{e}$ if and only if*

$$0 \le x \le t_1,\ t_2 \le y \le \tau(\overline{e}). \tag{4}$$

Proof. We prove this in two dimensions. For the x-dimension, based on Step 1 of Phase-II in BuildAF, we observe that an affiliated forest $(V, \mathcal{F}_{x_1,t})$ is a superset of the following affiliated forest $(V, \mathcal{F}_{x_2,t})$, i.e.,

$$\forall 0 \le x_1 \le x_2 \le t \le t_{\max}, \mathcal{F}_{x_1,t} \supseteq \mathcal{F}_{x_2,t}. \tag{5}$$

For the y-dimension, based on Step 2 of Phase-II in BuildAF, we observe that if an affiliated forest $(V, \mathcal{F}_{t,y_1})$ does not contain $\overline{e}$, none of the following affiliated forests $\mathcal{F}_{t,y_2}$ will contain it, i.e.,

$$\overline{e} \notin \mathcal{F}_{t,y_1} \implies \forall y_1 \le y_2 \le t_{\max},\ \overline{e} \notin \mathcal{F}_{t,y_2} \tag{6}$$

Besides, we note that (u, v, t_1, t_2) is first added to an affiliated forest $\mathcal{F}_{t_1,t_2}$ by BuildAF either in Phase-I or in Step 3 of Phase-II (note that it cannot appear in any affiliated forest $(V, \mathcal{F}_{x,y})$ with $x > t_1$ or $y < t_2$ since $\mathcal{F}_{x,y}$ is a subset of $\{(u', v', t_1', t_2') \in \overline{\mathcal{E}} \mid [t_1', t_2'] \subseteq [x, y]\}$). Finally, based on Equation (5) and Equation (6), we can easily prove the correctness of the lemma. □

Based on the above lemma, each spanning edge $\overline{e} = (u, v, t_1, t_2)$ in the affiliated forests corresponds to a 3-D vector $\langle t_1, t_2, \tau(\overline{e})\rangle$. Clearly, an affiliate forest $(V, \mathcal{F}_{t_s,t_e})$ can be retrieved by finding all those edges satisfying Equation (4), i.e., $t_s \le t_1 \le t_2 \le t_e \le \tau(\overline{e})$. To achieve this efficiently, after obtaining all 3-D vectors, we organize them via a 3-D range tree, which takes $O(\overline{m}(\frac{\log \overline{m}}{\log\log \overline{m}})^2)$ space [9]. We note that each 3-D vector corresponds to a data point (x, y, z) in the 3-D range tree, and the 3-D orthogonal query with $(q_{x_1}, q_{x_2}, q_{y_1}, q_{y_2}, q_{z_1}, q_{z_2})$, which aims to find all data points satisfying $q_{x1} \le x \le q_{x2} \wedge q_{y1} \le y \le q_{y2} \wedge q_{z1} \le z \le q_{z2}$, can be answered in $O(k + \log^2 \overline{m})$ where k is the number of output data points [9]. Finally, to answer a Spanning-CC query with $[t_s, t_e]$, we first retrieve the affiliate forest $\mathcal{F}_{t_s,t_e}$ by conducting a 3-D orthogonal query with $(t_s, t_e, t_s, t_e, t_e, t_{\max})$ on the 3-D range tree, and then find all CCs with at least two vertices in its projected graph. We refer to the above process as AF-3DT-Query.

Lemma 3.7. AF-3DT-Query *answers a query in $O(n + \log^2 \overline{m})$.*

Proof. This can be easily verified based on the fact that an affiliated forest contains at most $n - 1$ edges. □

3.2.3 Construction of AF-3DT. Given all 3-D vectors, a 3-D range tree can be constructed in $O(\overline{m} \log \overline{m})$ [9]. One remaining question is how to compute $\tau(\overline{e})$ for each edge in the affiliated forests (while filtering out other redundant edges in $\overline{\mathcal{G}}$). This step can be done by constructing all affiliated forests using BuildAF, where the time complexity is proven below.

Lemma 3.8. BuildAF *runs in $O(\alpha(\overline{m}) + \alpha(nt_{\max}^2))$, where $\alpha(\cdot)$ is the inverse Ackerman function.*

Proof. To perform the two phases of BuildAF, we make use of a classical disjoint-set (union-find) data structure [16]. The disjoint-set data structure with n elements is typically implemented using a forest of trees where each tree represents a set, and the root of each tree serves as the representative of its set, allowing for efficient operations to check connectivity in $\alpha(n)$ amortized time and merge sets in $O(1)$ time, where $\alpha(\cdot)$ is the inverse Ackerman function. Specifically, during Phase-I and Step 3 of Phase-II, we perform connectivity checks for each spanning edges, resulting in

Algorithm 1: AFDecomp

Input: An affiliated graph $\overline{\mathcal{G}} = (V, \overline{\mathcal{E}})$
Output: The set of 3-D vectors $\langle t_1, t_2, \tau(\overline{e})\rangle$

```
1  Obtain the edge ordering ⟨ē_1, ē_2, ..., ē_m̄⟩ based on ω_1(·);
2  Initialize S^(0) ← ∅, D ← ∅, i ← 1;
3  foreach edge ē_i = (u, v, t_1, t_2) in the ordering do
4      if u and v are not connected in (V, S^(i−1)) then
5          S^(i) ← S^(i−1) ∪ {ē_i};
6      else
7          ē_min ← the edge with the smallest value of ω_2(·) among those in the path
             connecting u and v;
8          if ω_2(ē_i) > ω_2(ē_min) then
9              S^(i) ← S^(i−1) ∪ {ē_i} \ {ē_min};
10             τ(ē_min) ← t_2 − 1, D ← D ∪ {ē_min};
11         else S^(i) ← S^(i−1) ;
12 foreach edge ē in S^(m̄) do τ(ē) ← t_max;
13 return {⟨t_1, t_2, τ(ē)⟩ | ē ∈ S^(m̄) ∪ D};
```

an amortized time of $O(\alpha(\overline{m}))$ in total. Moreover, Step 1 of Phase-II creates a copy of the disjoint-set structure, requiring $O(n)$ time. Then, Step 2 has at most n times of connectivity checks, incurring $O(\alpha(n))$ time. These two steps are executed for each $\mathcal{F}_{t_s,t_e}$, of which there are $O(t_{max}^2)$ in total. Consequently, the overall time complexity of the algorithm BuildAF is $O(\alpha(\overline{m}) + \alpha(nt_{\max}^2))$. □

However, this approach is time-consuming due to the need to construct $O(t_{\max}^2)$ affiliated forests. To address this, we instead introduce an efficient *affiliated forest decomposition* method called AFDecomp. The idea is to utilize the overlaps between different affiliated forests to avoid building each one from scratch.

For each spanning edge $\overline{e}$ in the affiliated forests, we have

$$\overline{e} \in \mathcal{F}_{0,\tau(\overline{e})} \text{ and } \overline{e} \notin \mathcal{F}_{0,\tau(\overline{e})+1}, \tag{7}$$

according to Equation (5). Therefore, to compute all τ values, we only need to construct $t_{\max}$ affiliated forests, i.e., $(V, \mathcal{F}_{0,0})$, $(V, \mathcal{F}_{0,1})$, ..., $(V, \mathcal{F}_{0,t_{\max}})$. Besides, these $t_{\max}$ affiliated forests also have some overlapping structures and thus can be built in an efficient *bottom-up* manner. Below, we elaborate on the details.

We first define two types of weights for each spanning edge $\overline{e} = (u, v, t_1, t_2)$ in $\overline{\mathcal{G}}$ as below.

$$\omega_1(\overline{e}) = (t_2 + 1) \times t_{\max} - t_1,\ \omega_2(\overline{e}) = (t_1 + 1) \times t_{\max} - t_2. \tag{8}$$

Let $\langle \overline{e}_1, \overline{e}_2, ..., \overline{e}_{\overline{m}}\rangle$ be a sequence of edges in $\overline{\mathcal{E}}$ called ω_1-edge ordering, which is sorted in a non-decreasing order w.r.t. $\omega_1(\cdot)$, i.e.,

$$\omega_1(\overline{e}_i) \le \omega_1(\overline{e}_j),\ 1 \le i \le j \le \overline{m}. \tag{9}$$

We note that one spanning edge (u, v, t_1, t_2) comes before the other (u', v', t_1', t_2') in the ordering if (1) $t_2 < t_2'$ or (2) $t_2 = t_2'$ but $t_1 > t_1'$.

We summarize AFDecomp in Algorithm 1. AFDecomp iteratively considers adding a spanning edge (u, v, t_1, t_2) to a graph $(V, \overline{\mathcal{S}})$, which is initially empty and globally maintained (as a spanning

forest), based on a certain edge ordering $\langle \overline{e}_1, \overline{e}_2, ..., \overline{e}_{\overline{m}} \rangle$ (Lines 3-11). In particular, if u and v are not connected in $(V, \overline{\mathcal{S}})$, the spanning edge (u, v, t_1, t_2) is directly added to $\overline{\mathcal{S}}$ (Lines 4-5); otherwise, it is replaced with the spanning edge $\overline{e}_{min}$ with the smallest value of ω_2 among those edges in the path connecting u and v (Lines 6-10). We note that there exists some spanning edges that do not fall into the above two cases, i.e., u is connected to v in $(V, \overline{\mathcal{S}})$ and $\omega_2(\overline{e}) \leq \omega_2(\overline{e}_{min})$ (Line 11). We remark that maintaining $\mathcal{S}$ only needs a single set and the notation $\mathcal{S}^{(i)}$ is used only to refer to the edge set $\mathcal{S}$ obtained after the i-th round at Line 3.

Correctness. We then show the correctness of AFDecomp. Recall that BuildAF could build different sets of affiliated forests. Based on the ω_1-edge ordering, we have the following definition.

Definition 3.9 (ω_1-based affiliated forests). The ω_1-based affiliated forests is a set of affiliated forests built by BuildAF based on the ω_1-edge ordering as described below.

- In Phase-I, $(V, E_{t,t})$ is the spanning forest of $\overline{G}_{[t,t]}$ with the smallest lexicographic order of $o_1 o_2 ... o_{|E_{t,t}|}$, where o_i is the index of edge $\overline{e}_{o_i}$ in the ω_1-edge ordering;
- In Phase-II, we consider each edge (u, v, t_1, t_2) or (u, v, t_s, t_e) to be added to $\mathcal{F}_{t_s,t_e}$ based on the ω_1-edge ordering.

We next show that AFDecomp returns the set of 3-D vectors for ω_1-based affiliated forests in the following lemma.

Lemma 3.10. AFDecomp *finds the set of 3-D vectors for the ω_1-based affiliated forests.*

Proof. Let $\{(V, \mathcal{F}_{0,0}), ..., (V, \mathcal{F}_{t_{\max},t_{\max}})\}$ be the set of ω_1-based affiliated forests. For any $0 \leq y \leq t_{\max}$, let $\overline{e}_{I(y)} = (u, v, t_1, t_2)$ with $0 \leq I(y) \leq \overline{m}$ be the last edge in the ω_1-edge ordering such that $t_1 = 0$ and $t_2 = y$. We observe that $\mathcal{F}_{0,y}$ $(0 \leq y \leq t_{\max})$ can be built by $\mathcal{S}^{(I(y))}$, formally,

$$\forall 0 \leq y \leq t_{\max},\ \mathcal{S}^{(I(y))} = \mathcal{F}_{0,y}. \tag{10}$$

The correctness of the above equation can be proved through induction, using a method similar to the one used to prove the correctness of BuildAF in Section 3.2.1. Based on Equation (10), those edges removed from $\mathcal{S}$ between the $I(y-1)$-th round and the $I(y)$-th round at Line 10, i.e., $\mathcal{S}^{(I(y-1))} \backslash \mathcal{S}^{(I(y))}$, are in $\mathcal{F}_{0,y-1}$ but not in $\mathcal{F}_{0,y}$. Then, by Equation (7), these edges have the τ value of $y-1$. For the remaining edges in $\mathcal{S}^{\overline{m}}$ (i.e., $\mathcal{F}_{0,\overline{m}}$), they have the τ value of $t_{\max}$. In summary, the correctness of AFDecomp can be guaranteed. □

Time complexity. We note that the iterative procedure corresponds to the problem of incremental maximum spanning forest maintenance, which can be solved using a link-cut tree structure implemented with common binary trees and splay trees [44]. Hence, the time complexity is bounded by $O(\overline{m} \log \overline{m})$.

Lemma 3.11. AFDecomp *runs in $O(\overline{m} \log \overline{m})$.*

3.3 An Advanced Index-based Solution: AF-BTAS

We note that 3-D orthogonal queries with $(t_s, t_e, t_s, t_e, t_s, t_{\max})$ conducted on AF-3DT involve only 2 input variables, i.e., t_s and t_e; and the 3-D range tree is designed for answering the general 3-D orthogonal queries with 6 input variables. Motivated by this, we propose to design a *specialized* index structure, called *Binary Tree with Auxiliary Substructures* (**BTAS**), for better answering the former queries (which finds all data points (x, y, z) such that $x \leq t_s \leq y \leq t_e \leq z$ for any input time window $[t_s, t_e]$). As a result, our affiliated-forest-based index solution with **BTAS**, named AF-BTAS (Affiliated-Forest-based BTAS Index), can be built in $O(\overline{m} \log \overline{m})$ and answers a spanning-CC query

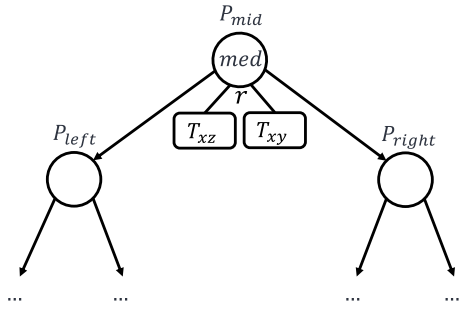


Fig. 6. The recursive structure of **BTAS**

in $O(n + \log^2 \overline{m})$, which is the same as AF-3DT. We remark that AF-BTAS takes $O(\overline{m})$ space which is smaller than that of AF-3DT. Below, we elaborate on the details.

Index structure and construction. Given the set of 3-D vectors $P = \{(x_1, y_1, z_1), ..., (x_{|P|}, y_{|P|}, z_{|P|})\}$ (or, data points), our proposed **BTAS** T is a binary tree with auxiliary substructures. It can be built by recursively partitioning the set of data points into several smaller subsets, with each subset maintained in the corresponding subtrees or auxiliary substructures. Specifically, the **BTAS** T can be defined recursively as follows. See Figure 6 for illustration.

(1) If the data point set $P = \emptyset$, **BTAS** T is an empty tree;

(2) Otherwise, we define *med* by the median of the set of numbers $\{y_1, \ldots, y_{|P|}, z_1, \ldots, z_{|P|}\}$. We can partition the data points in P into three disjoint sets of data point P_{left}, P_{mid}, and P_{right}:

$$P_{left} := \{(x, y, z) \in P \mid z < med\},$$
$$P_{mid} := \{(x, y, z) \in P \mid y \leq med \leq z\},$$
$$P_{right} := \{(x, y, z) \in P \mid y > med\}.$$

Let the root node of the current **BTAS** T be r. Then, we can define:

- the left sub-tree T_{left} of r is a **BTAS** tree associated with the set of data points P_{left};
- the right sub-tree T_{right} of r is a **BTAS** tree associated with the set of data points P_{right}.

Moreover, the root node r stores the median *med* and is associated with the set of P_{mid} using two auxiliary substructures:

- T_{xy} facilitates querying 2-D data points in the set $\{(x, y) \mid (x, y, z) \in P_{mid}\}$, within the range $q_1 \leq x \leq q_2$ and $y \leq q_2$.
- T_{xz} facilitates querying 2-D data points in the set $\{(x, z) | (x, y, z) \in P_{mid}\}$, within the range $q_1 \leq x \leq q_2$ and $q_2 \leq z$.

We remark that the two auxiliary substructures in **BTAS** are implemented using the well-known *priority search tree*. In particular, a priority search tree takes input as three parameters q_x, q'_x, q_y and returns data points (x, y) that satisfy $q_x \leq x \leq q'_x$ and $q_y \leq y$ in $O(\log a + b)$ time, where a is the total number of data points stored in the priority search tree and b is the number of reported data points [35]. For completeness, we summarize the construction of **BTAS** in Appendix C of [1]. We note that **BTAS** T is a balanced search tree, since it partitions P using the median *med*, resulting in two subtrees of comparable size. Moreover, the height of **BTAS** is $O(\log \overline{m})$. Finally, we analyze the time and space complexity as follows.

Lemma 3.12. *The construction of **BTAS** runs in $O(\overline{m} \log \overline{m})$ and* BTAS-Index *costs $O(\overline{m})$ space.*

Proof. The **BTAS** structure is built recursively. For each recursion, the time complexity is dominated by the index construction time of the priority search tree T_{xy} and T_{xz} in Lines 9-10 of

Algorithm **??**. According to [35], the priority search trees T_{xy} and T_{xz} with $|P_{mid}|$ data points take $O(|P_{mid}| \log |P_{mid}|)$ time. Hence, the time in each recursion is $O(\overline{m} + |P_{mid}| \log |P_{mid}|)$. Then, it is easy to see the total complexity of Algorithm **??** is $O(\overline{m} \log \overline{m})$.

For the space complexity, we can observe that (1) each data point is only stored in one tree node in a **BTAS** tree, and (2) each data point is stored at most twice in the two priority search trees. Thus, the total space complexity is $O(\overline{m})$. □

Query processing. Given a time window $[t_s, t_e]$, based on the **BTAS** structure, we can efficiently identify the affiliated forest by finding data points (x, y, z) that satisfy the condition $t_s \le x \le y \le t_e \le z$ (Lines 1-2), and thus answer a spanning-CC query efficiently (Line 3). In particular, the procedure `BTAS-Query-Rec` corresponds to the procedure of finding data points from `BTAS`. In general, it descends recursively along a path in `BTAS`, which starts from the root node to one of the leaf nodes. Along the path, it aggregates the sets of data points that meet the required condition in both the tree nodes and the auxiliary substructures. Upon reaching the leaf of T, the final set is returned. Specifically, starting with the root, we have the following cases.

(1) We first consider P_{left} and P_{right} in sub-trees of the current T.
   (a) If $t_e < med$, we search along the left sub-tree since any data point in P_{right} satisfies $y > med > t_e$ and thus violates the query condition $y \le t_e$.
   (b) If $t_e > med$, we search along the right sub-tree since any data point in P_{left} satisfies $t_e > med > z$ and thus violates the query condition $t_e \le z$.
   (c) If $t_e = med$, the data points in the final result set cannot lie in either sub-tree, and we handle these data points in the auxiliary substructures.

(2) We next consider P_{mid} and its two auxiliary substructures in the root node of the current T. In particular, we adopt the well-known priority search tree [35] to implement the auxiliary substructures; and `PST-Query` is the procedure for querying 2-D data points (details can be found in the technical report [1]).
   (a) If $t_e \le med$, we know that $t_e \le z$ is already satisfied for all data points in P_{mid}. Thus, we focus on the query condition $t_s \le x \le y \le t_e$, and obtain the related data points using the auxiliary substructure T_{xy}.
   (b) If $t_e > med$, we know that $y \le t_e$ is already satisfied for all data points in P_{mid}. Thus, we focus on the query condition $t_s \le x \wedge t_e \le z$, and obtain the related data points using the auxiliary substructure T_{xz}.

We summarize the query processing based on **BTAS** T in Appendix C of [1]. It is easy to verify the correctness of Algorithm **??** based on the above discussion. We then analyze the time complexity as below.

Lemma 3.13. `BTAS-Query` *answers a query in* $O(n + \log^2 \overline{m})$.

Proof. The time complexity is bounded by the part of finding data points from **BTAS** at Line 2. Specifically, **??** would be recursively invoked $O(\log \overline{m})$ rounds and each round will visit a tree node since the height of a **BTAS** T is $O(\log \overline{m})$ and there are at most $\overline{m}$ data points in **BTAS**. Besides, for each round, the time cost is bounded by that of `PST-Query`, which runs in $O(\log \overline{m})$ time [35]. Finally, at most $n - 1$ data points will be returned since each corresponds to an edge in the affiliated forest $\mathcal{F}_{[t_s,t_e]}$. Therefore, the total complexity is $O(\log^2 \overline{m} + n)$. □

3.4 Summary and His-CSM Query Instances

Given a temporal graph $\mathcal{G}$ and a non-overlapping and monotonic CSM $\mathcal{P}$, our framework involves three components, namely affiliated graph construction, index construction over affiliated forests, and query processing, as summarized in Table 1.

Table 1. Comparison among different algorithms, where n and m denote the number of vertices and edges in $\mathcal{G}$, respectively. Let m^* be the number of edges in $G_{[0,t_{\max}]}$, and let $\overline{m}$ be the number of edges in the affiliated graph. The quantities $\bar{t}$ and $\bar{l}$ are theoretically bounded by $t_{\max}$, and $D_{\max}$ is the maximum degree. The authors of [57] claim that TSF answers a query in $O(n)$, which ignores the cost of collecting edges from the index. Based on the TSF implementation, the actual query time is $O(n + t_s)$.

Algorithm	CSM model	Affiliated graph construction	Index construction	Index space	Query processing
AF-3DT	CSM-free	$O(n^2 \cdot t_{\max} \cdot f(\mathcal{P}, G))$	$O(\overline{m} \log \overline{m})$	$O(\overline{m}(\frac{\log \overline{m}}{\log \log \overline{m}})^2)$	$O(n + \log^2 \overline{m})$
AF-BTAS	CSM-free	$O(n^2 \cdot t_{\max} \cdot f(\mathcal{P}, G))$	$O(\overline{m} \log \overline{m})$	$O(\overline{m})$	$O(n + \log^2 \overline{m})$
BTAS-CC	His-CC	$O(n + m)$	$O(\overline{m} \log \overline{m})$	$O(\overline{m})$	$O(n + \log^2 \overline{m})$
TSF [57]	His-CC	N.A.	$O(\overline{m} t_{\max})$	$O(\overline{m})$	$O(n + t_s)$
BTAS-Core	His-Core	$O(m^* \cdot \bar{t} \cdot D_{\max})$	$O(\overline{m} \log \overline{m})$	$O(\overline{m})$	$O(n + \log^2 \overline{m})$
PHC [63, 64]	His-Core	N.A.	$O(m^* \cdot \bar{t} \cdot D_{\max})$	$O(m^* \cdot \bar{t})$	$O(n \cdot \log \bar{t})$
MCTS [50]	His-Core	N.A.	$O(m^* \cdot \bar{l} \cdot D_{\max})$	$O(m^* \cdot \bar{l})$	$O(n \cdot \log \bar{l})$

Algorithm 2: `Construction`$(\mathcal{P}, \mathcal{G})$

Input: The temporal graph $\mathcal{G} = (V, \mathcal{E})$ and a CSM $\mathcal{P}$
Output: An affiliated graph $\overline{\mathcal{G}}$

```
1  Ē ← ∅;
2  foreach pair of vertices u and v in V do
3      [t_s, t_e] ← [0, 0];
4      while t_e is no larger than t_max do
5          P(G_[t_s,t_e]) ← OnlineQ⟨P⟩(G_[t_s,t_e]);
6          if u, v are in the same subgraph in P(G_[t_s,t_e]) then
7              while t_s ≤ t_e and u, v are in the same subgraph in P(G_[t_s,t_e]) do
                   [t_s, t_e] ← [t_s + 1, t_e] ;
8              Ē ← Ē ∪ {(u, v, t_s − 1, t_e)};
9          [t_s, t_e] ← [t_s, t_e + 1];
10 return Ḡ(V, Ē);
```

One remaining issue is *how to compute* $\overline{\mathcal{E}}$ for constructing $\overline{\mathcal{G}}$ based on $\mathcal{G}$ and $\mathcal{P}$. Let `OnlineQ`$\langle\mathcal{P}\rangle(G)$ be a procedure which receives a graph as the input and returns all CSMs in $\mathcal{P}(G)$. We note that its implementation has been widely studied for many CSMs, e.g., k-core [13, 24], (α,β)-core [30], and connected components [16]. Besides, we denote by $f(\mathcal{P}, G)$ the running time of `OnlineQ`$\langle\mathcal{P}\rangle(G)$. Based on this procedure, we present an algorithm `Construction` for constructing $\overline{\mathcal{G}}$ in Algorithm 2. The correctness can be easily verified. Besides, we can derive that (1) `Construction` runs in $O(|V|^2 \cdot t_{\max} \cdot f(\mathcal{P}, G))$ and (2) the number of spanning edges in $\overline{\mathcal{G}}$ is bounded by $|V|^2 t_{\max}$.[1] We remark that `Construction` is a general method for various CSMs, which can be further improved for specific CSMs, as described in the following.

We then adopt our `AF-BTAS` for solving two His-CSM query instances, namely His-CC query and His-Core query, which have been studied in the literature [57, 63]. The corresponding index

[1]Note that each spanning edge in the affiliated graph is associated with a pair of timestamps $[t_s, t_e]$ representing the interval instead of one timestamp. Thus, an arbitrary affiliated graph can contain $O(|V|^2 t_{\max}^2)$ spanning edges since (1) the number of edges in the projected graph is bound by $|V|^2$ and (2) an edge can have up to $t_{\max}^2$ time intervals. In addition, the bound $|V|^2 t_{\max}$ applies generally to all CSMs. For some specific CSM, a tighter bound can be derived.

solutions are called **BTAS-CC** and **BTAS-Core**, respectively. We note that both CC and k-core are non-overlapping and monotonic according to Lemma 2.3. Below, we provide the details.

BTAS-CC for His-CC query. Given a temporal graph $\mathcal{G} = (V, \mathcal{E})$, we observe that $\overline{\mathcal{G}} = (V, \{(u, v, t, t) \mid (u, v, t) \in \mathcal{E}\})$ is an affiliated graph according to Definition 3.3 since (1) each spanning edge (u, v, t, t) corresponds to a minimal u-v-TW, i.e., u and v are in the same CC in $G_{[t,t]}$, and (2) the constraint (that u and v are in the same CC in $G_{[t,t]}$ if and only if u is reachable to v in $\overline{G}_{[t_s,t_e]}$) is satisfied since $G_{[t_s,t_e]}$ is the same as $\overline{G}_{[t_s,t_e]}$.

We note that the affiliated graph $\overline{\mathcal{G}}$ can be efficiently obtained in $O(n + m)$, which is also sparser than that obtained by `Construction`. Then, we can construct the corresponding **BTAS** index structure in $O(\overline{m} \log \overline{m})$ by `AFDecomp` and `BTAS-Construction` (note that $\overline{m}$ is no greater than m based on the built affiliated graph). Finally, we can answer the His-CC queries in $O(n + \log^2 \overline{m})$ by using `BTAS-Query`. We call the above adaptation **BTAS-CC** and summarize in Table 1. In particular, compared to the SOTA TSF [57] for answering His-CC query, our **BTAS-CC** has following benefits. First, **BTAS-CC** has the time cost for index construction and query processing practically and theoretically smaller than that of TSF (as verified in Figure 10). The possible reasons include (1) `AFDecomp` will filter out some redundant edges from the affiliated graph, and (2) TSF needs to iterate each timestamp from 1 to t_s for answering a His-CC query. Second, **BTAS-CC** has a space cost comparable to that of TSF in both theory and practice (as verified in the experiments).

BTAS-Core for His-Core query. The affiliated graph can be built by adopting existing historical k-core enumeration methods [45, 58]. Specifically, they can list all distinct k-cores along with their minimal time intervals in $O(m^* \cdot \overline{t} \cdot D_{\max})$, as summarized in Table 1. Then, we follow our index-based framework for building the index and answering the queries. We observe that our **BTAS-Core** outperforms the SOTA PHC theoretically and practically. In the worst case, **BTAS-Core** has the time cost for query processing bounded by $O(n + \log^2(\overline{m}))$ [2], while PHC has that bounded by $O(n \log t_{\max})$.

Remark on index maintenance.

Our index structure supports incremental updates when new edges arrive. **First**, when edges with timestamp $t_{\max} + 1$ are inserted, we update the refined affiliated graph. To do this, we run Algorithm 1 only on the new edges (Lines 3-13), initializing with the previously computed refined affiliated graph (i.e., $\mathcal{S}^{(0)}$ in Line 2 now corresponds to the previously computed refined affiliated graph with timestamp at most $t_{\max}$). Because edges are processed in increasing order of ω_1, the new edges are correctly incorporated into the existing refined affiliated graph. Let m^+ be the number of new edges and $\overline{m}$ the number of edges in the previous refined affiliated graph. The time complexity of this phase is $O(m^+ \log(\overline{m} + m^+))$. **Second**, these modifications are propagated to our index structure **BTAS**. For any edge whose τ value has changed or which becomes a new spanning edge, the corresponding entry in **BTAS** is updated. As **BTAS** is a tree-of-trees with balanced structures, each update requires traversing only a logarithmic-height path. Since there are $O(m^+)$ such updates, this phase takes $O(m^+ \log(\overline{m} + m^+))$. Therefore, the overall time complexity of maintaining both the refined affiliated graph and **BTAS** under an incremental update of m^+ edges is $O(m^+ \log(\overline{m} + m^+))$. The pseudo-code for our index maintenance is provided in Appendix [1].

Extensions to other temporal models. Our framework can be generalized to continuous-time and interval-based temporal graphs.

Continuous-time graphs. While our model assumes discrete timestamps, it extends to continuous-time graphs through a standard discretization strategy. Since the set of temporal edges is finite, we

[2]It is challenging to give a simple formula for the number of spanning edges $\overline{m}$, but the size is bounded according to the index size analysis in [64].

Table 2. Statistics of all datasets

Datasets	$\|V\|$	$\|\mathcal{E}\|$	$t_{\max}$
contact (CT)	275	28,244	15,662
ColledgeMsg (CM)	1,862	59,835	58,910
sx-superuser (SU)	159,481	144,339	143,198
email-Eu-core (EE)	966	332,334	207,879
sx-mathoverflow (MO)	21,980	506,550	505,783
facebook-wosn-links (FL)	63,732	817,035	333,924
sx-askubuntu (AU)	127,047	964,437	960,865
mit (MI)	97	1,086,404	33,452
wiki-talk-temporal (WT)	1,120,716	7,833,140	7,375,041
youtube-u-growth (YG)	3,223,590	9,375,374	203
wikipedia-growth (WG)	1,870,710	39,953,145	2,198
sx-stackoverflow (SO)	2,296,666	63,497,050	52,423,236

can identify all unique edge appearance and disappearance timestamps to form a sorted sequence of critical event points. These points define a discretized timeline where the graph topology remains invariant between any two consecutive event points. Because this discretization captures every potential topological change, it preserves the exact semantics of Historical-CSM queries. Consequently, our index construction and query processing algorithms remain applicable to this discretized timeline without modification.

Interval-based graphs. Similarly, our framework naturally supports general interval-based temporal graphs where each edge is associated with an active duration $[t_1, t_2]$. To support this, we only need to adapt the definition of the projected graph $G_{[t_s,t_e]}$. Specifically, we redefine the projected edge set $E_{[t_s,t_e]}$ to include all edges whose active intervals overlap with the query window: $E_{[t_s,t_e]} = \{(u,v) \mid (u,v,t_1,t_2) \in \mathcal{E}, [t_1,t_2] \cap [t_s,t_e] \neq \emptyset\}$. With this updated projection rule, our underlying index structure and search algorithms remain valid and effective.

4 Experiments

We conduct extensive experiments to evaluate the performance of our proposed AF-BTAS. In particular, we adapt our AF-BTAS to His-CC and His-Core queries (described in Section 3.4) and compare the resulting adaptations, namely **BTAS-CC** and **BTAS-Core**, with the state-of-the-art methods TSF [57] (for His-CC) and PHC [63] and MCTS [50] (for His-Core), respectively. All algorithms are implemented in C++, compiled with g++ -O3, and run on a Linux machine with Intel Xeon 2.60GHz CPU and 256GB RAM. Our source codes are available at https://anonymous.4open.science/r/Querying-Cohesive-Subgraphs-in-Temporal-Graphs-C38D. We evaluate the algorithms on 12 real-world datasets collected from SNAP[3] and KONECT projects[4]. The statistics of all datasets are shown in Table 2.

4.1 Experiments for Historical-CC Queries

Query processing. We generate 10,000 random queries with different time intervals of $[t_s, t_e]$,

We follow the experimental setup in previous works [50, 57, 63, 64] to sample both the query length L and the start time t_s uniformly. Specifically, intervals are generated by first sampling the query length L uniformly from $[0, t_{\max}]$, and then sampling the start time $t_s \sim \text{Uniform}[0, t_{\max} - L]$,

[3]https://snap.stanford.edu/data/
[4]http://konect.cc/networks/

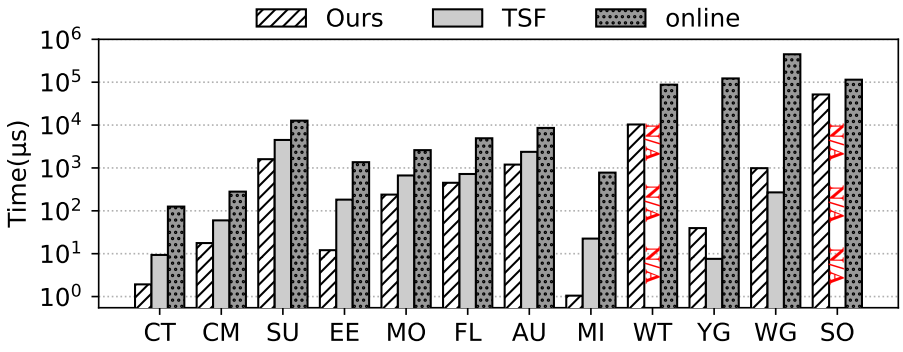


Fig. 7. Average time cost per query for Historical-CC

with $t_e = t_s + L$. The rationale is to conduct experiments that cover the entire parameter space, ranging from short to long queries, without bias towards any specific pattern. [5]

The average time cost per query on each dataset is shown in Figure 7. We have the following observations. **First**, our approach **BTAS-CC** is generally faster and achieves up to 20× speedups (on the EE and MI datasets) compared with the SOTA TSF. Moreover, we note that the results of the TSF index on the WT and SO datasets are marked as **N/A** since we are unable to complete the index construction within 24 hours. Besides, the **online** method does not need to construct the related index but shows the worst performance in terms of query processing. **Second**, the TSF index has a better performance on the YG and WG datasets. This may be due to the fact that the largest values of timestamp $t_{\max}$ in these two datasets — 203 for YG and 2,198 for WG — are relatively small compared with the scale of datasets (YG and WG have 9,375,374 and 39,953,145 temporal edges, respectively). In such cases, iterating through timestamps in the TSF approach becomes less time-consuming.

We also evaluate the time cost per query against different query time intervals, i.e., $t_e - t_s$, in Figure 8. Specifically, we set the interval sizes to as 20%, 40%, 60%, and 80% of $t_{\max}$ and record the average running time of 10,000 queries with each query interval size. We can see that our method **BTAS-CC** outperforms TSF and **online** in most cases, particularly when the size of query interval is small. For example, our method achieves up to 60× speedups on the EE and MI datasets when the interval size is 20% of $t_{\max}$. However, as the time interval size increases, the performance gap narrows, where TSF has a better performance on the query interval of $80\%t_{\max}$ in some datasets. This is because the query performance of TSF depends on the value of $t_{\max}$ and performs better when the query interval is large, say at least $80\%t_{\max}$. On the other hand, our approach **BTAS-CC** is suitable for different query time intervals.

To investigate the impact of $t_{\max}$ on query performance, we conduct additional experiments on synthetic datasets based on EE and MO. We systematically vary the maximum timestamp $t_{\max}$ from 5,000 to 50,000 with a step size of 5,000 by aggregating temporal snapshots to alter the time granularity. For the queries, we randomly select intervals whose lengths were fixed at 40% of $t_{\max}$. The average query time as a function of $t_{\max}$ is shown in Figure 9. We observe that the query time for both algorithms tends to increase as $t_{\max}$ grows. While TSF is initially faster when $t_{\max}$ is small, its query time increases more rapidly than that of our method as $t_{\max}$ increases. This may be due to

[5]We acknowledge that real-world workloads are often skewed (e.g., financial fraud detection favors short windows, whereas trend analysis typically spans longer periods). Thus, in Figures 8 and 13, we also evaluate the practical performance of all algorithms using fixed short or long query windows, e.g., fixed at 20%, 40%, 60%, or 80% of $t_{\max}$.

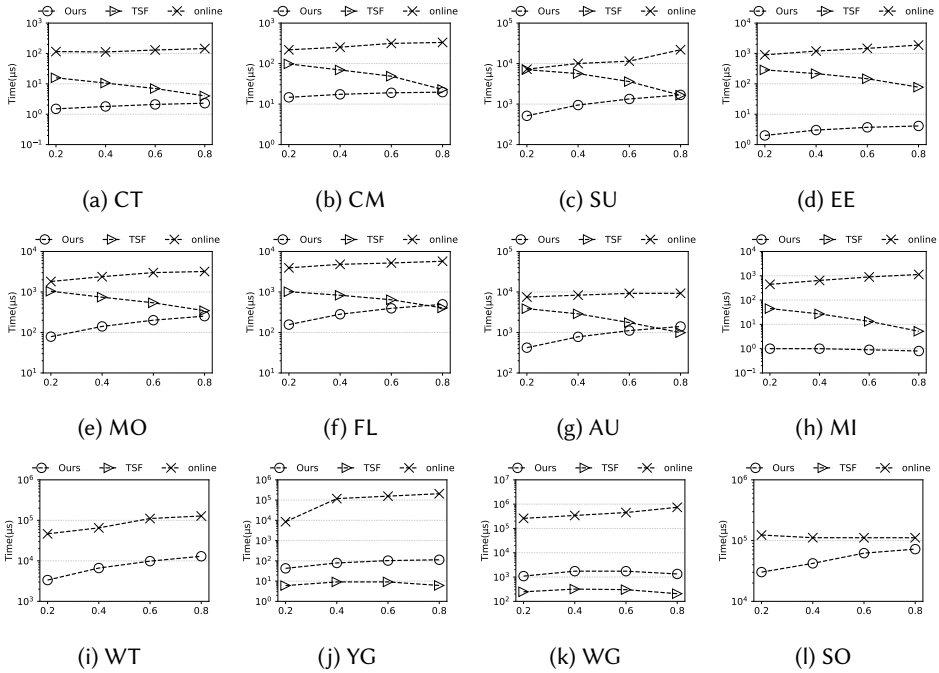
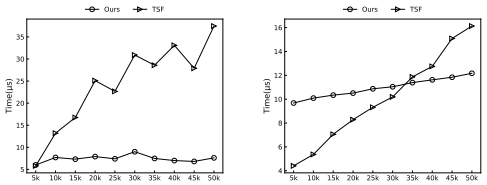


Fig. 8. Historical-CC query time by varying query time interval sizes



(a) EE-based Synthetic Dataset (b) MO-based Synthetic Dataset

Fig. 9. Average query time on synthetic datasets with varying $t_{\max}$

TSF's reliance on timestamp iteration, which becomes a bottleneck as $t_{\max}$ increases; in contrast, our query algorithm is robust to changes in $t_{\max}$.

Index construction. We first compare the efficiency of index construction on all datasets in Figure 10. As mentioned, the TSF method cannot construct the index within 24 hours on the WT and SO datasets. From Figure 10, it is clear to see that in most cases, our index construction method outperforms the TSF in terms of running time. In particular, we can construct our **BTAS-CC** index in 7.5 seconds on the WT dataset, while we are unable to build the TSF index even after spending more than 24 hours. The performance for index construction is in line with the theoretical result that the construction time of our method costs $O(\overline{m} \log \overline{m})$ while that of TSF requires $O(\overline{m} \cdot t_{\max})$ (as in Table 1). Besides, we report the space consumption of different methods in Figure 11. We can see that these two index structures **BTAS-CC** and TSF occupy nearly the same amount of space across most datasets, which is also implied by the same space complexity of $O(\overline{m})$ for both index structures. However, on the YG and WG datasets, **BTAS-CC** consumes slightly more space than TSF. The observed differences in index size may be attributed to the fact that TSF is more sensitive

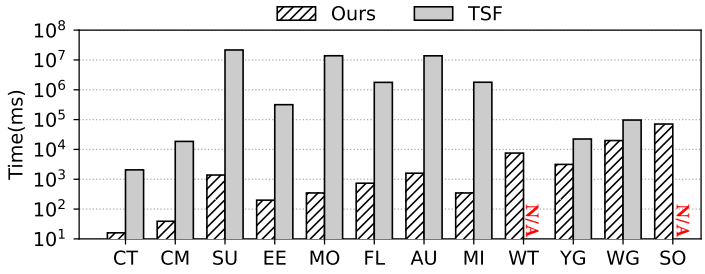


Fig. 10. Index construction time for Historical-CC

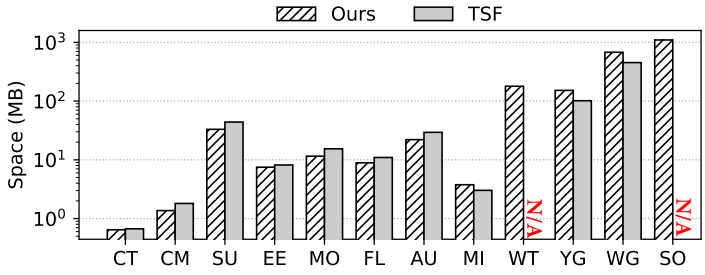


Fig. 11. Index space consumption for Historical-CC

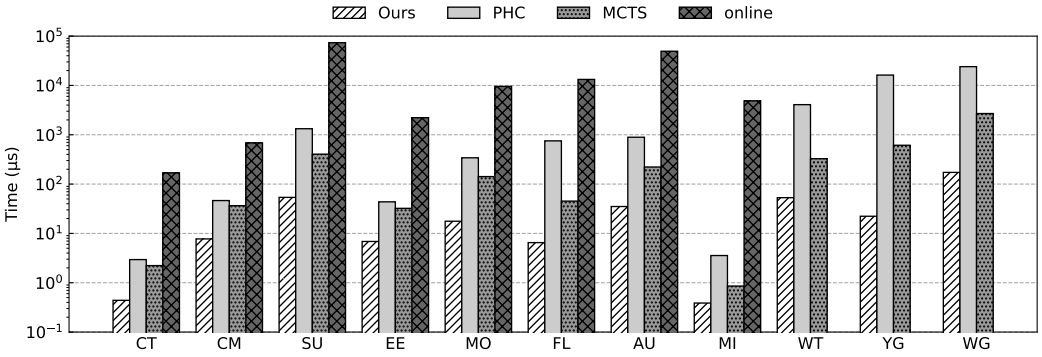


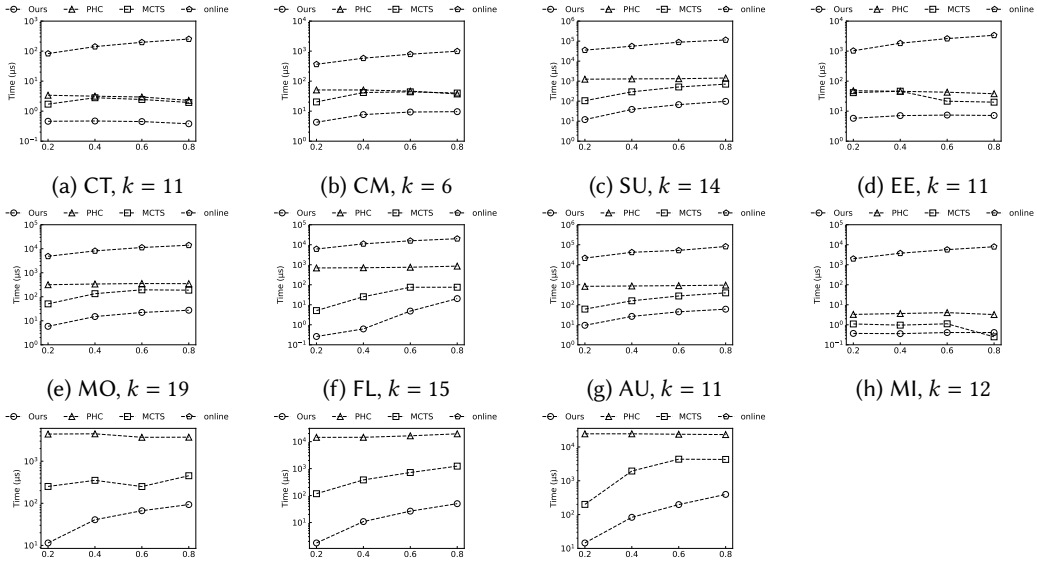
Fig. 12. Average time cost per query with $k = 30\%k_{\max}$ for Historical-Core

to the maximum timestamp $t_{\max}$ due to implementation details: when $t_{\max}$ is relatively small (e.g., 203 for YG and 2,198 for WG), TSF stores fewer temporal entries and therefore consumes less space.

4.2 Experiments for Historical-Core Queries

Query processing. We investigate the performance of query processing by varying the size of the input query time interval. Specifically, we first evaluate the average time cost for queries with 100,000 randomly selected pairs of t_s and t_e when $k = 30\%k_{\max}$ in Figure 12. From the results, we can see that (1) the **online** method cannot complete 100,000 queries in 12 hours for the WT, YG, and WG datasets; and (2) for the YG dataset, our method **BTAS-Core** runs 730× faster than PHC index and 30× faster than MCTS index.

Then, we vary the query interval size as 20%, 40%, 60%, and 80% of $t_{\max}$, and set the parameter k as 10%, 30%, 50%, 70% and 90% of $k_{\max}$. For each combination of query interval size L and k, we randomly generate 100,000 queries by sampling the start time t_s uniformly from $[0, t_{\max} - L]$ (setting $t_e = t_s + L$), following the same procedure as for His-CC queries. Figure 13 shows the



(a) CT, $k = 11$ (b) CM, $k = 6$ (c) SU, $k = 14$ (d) EE, $k = 11$

(e) MO, $k = 19$ (f) FL, $k = 15$ (g) AU, $k = 11$ (h) MI, $k = 12$

(i) WT, $k = 34$ (j) YG, $k = 26$ (k) WG, $k = 61$

Fig. 13. Historical-Core query time with $k = 30\%k_{\max}$ by varying query time interval sizes

average time cost per query for $k = 30\%k_{\max}$ (the results for other values of k are provided in Appendix E of [1]).

From Figure 13, we can observe that our method **BTAS-Core** outperforms the His-Core method PHC and the **online** method in all instances. Moreover, it surpasses MCTS in most cases, except on the MI dataset with $k = 12$ and the query interval size is 80% of $t_{\max}$. In a few corner cases, particularly when the query interval is small and the value of k is large, there is often no resulting k-core. In these situations, both MCTS and **BTAS-Core** can terminate early with minimal computation. In contrast, PHC suffers from having to iterate over all vertices, leading to significantly higher computation time. For example, **BTAS-Core** outperforms PHC by at least 100 times across all parameter combinations tested on the YG dataset. Moreover, the query time of **BTAS-Core** increases as the interval size increases. This is because a larger interval size involves more vertices in the k-core, requiring **BTAS-Core** to search through more tree nodes in a **BTAS-Core** tree. We next conduct experiments by varying the values of k on the FL and SU datasets in Appendix E of [1]. We conclude that the query time of all indices decreases monotonically with increasing k, since a larger k implies fewer vertices in the resulting k-core.

Index construction. We report the time cost required for constructing the index structures for the datasets used in our experiments. Note that the construction methods for Historical-Core queries build the structures for all values of k at once. The results are shown in Figure 14. All three algorithms **BTAS-Core**, PHC, and MCTS involve computing changes in vertex core numbers over time windows, which is the most time-consuming step, leading to similar execution time for both methods. Note that the result on the SO dataset is not included as we are unable to construct both index structures within 24 hours. Additionally, we measure and report the space consumption of index structure in Figure 15. **BTAS-Core** generally occupies slightly more space than PHC and MCTS.

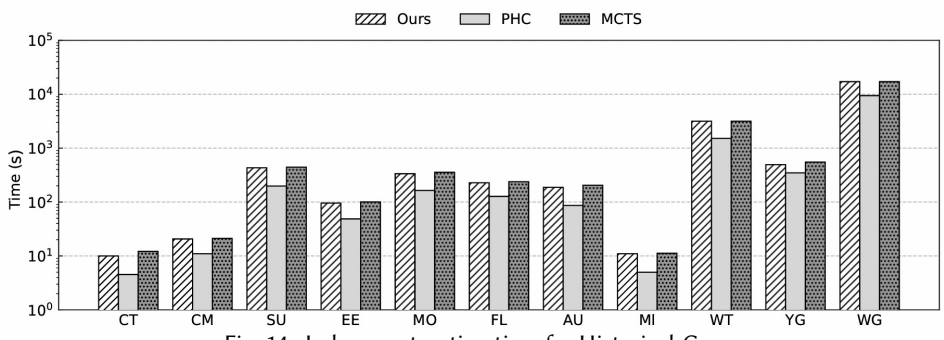


Fig. 14. Index construction time for Historical-Core

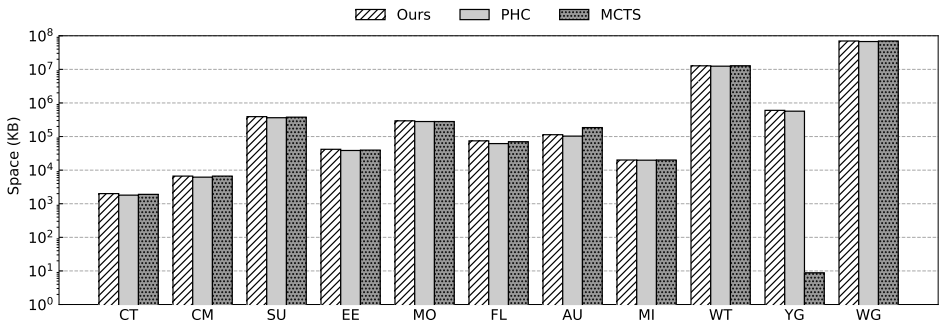


Fig. 15. Index space consumption for Historical-Core

5 Related Work

Historical-CSM queries in temporal graphs. The problem of Historical-CSM queries in temporal graphs has attracted tremendous attention, which aims to design indices for efficient retrieval of CSMs within an arbitrary time interval. The k-core model has been a central topic in the study of Historical-CSM queries. Yu et al. [63, 64] introduced the problem of historical k-core query and proposed the PHC-Index to efficiently address the problem. Building upon this work, Wang et al. [50] further improved the indexing strategy by proposing the MCTS-Index, which organizes vertices in a list of shells. This design demonstrates superior efficiency, particularly when the resulting k-core is relatively small. Later, Yang et al. [59] developed the EF-index to support the query of historical k-core component, given a specific vertex and time interval. Additionally, Tian et al. [48] and Li et al. [27] investigated the historical (α, β)-core model in temporal bipartite graphs. Besides core-like structures, Xie et al. [57] focused on the (strongly) connected components and studied the corresponding historical queries in temporal graphs. Recently, Chen et al. [10] considered the historical queries for maximal clique. In parallel, Tan et al. [46] studied the problem of efficiently querying k-trusses on temporal graphs and proposed two indices, which significantly improve query performance by avoiding repeated truss decompositions.

Other CSM problems in temporal graphs. There exist other CSM problems with different goals in temporal graphs. Yang et al. [58] and Zhong et al. [65] developed efficient online algorithms to identify all distinct historical k-cores, as well as k-cores that meet specific constraining conditions, across various time intervals. Galimberti et al. [19] introduced the concept of span-core, where two vertices must interact throughout the entire interval, and developed a core decomposition algorithm for this model. Similarly, Lotito and Montresor [31] introduced the concept of span-truss. Other studies have explored various time-relevant metrics for k-cores in temporal graphs, such as interaction frequency [2, 55], persistence [26], burstiness [40], continuity [28], temporal proximity [29], and reliability [47]. Besides, Guo and Sekerinski [20] studied how to maintain the

core structure when a new edge is appended in the temporal setting, while Yang et al. [60] studied the quasi-clique in temporal graphs. Another related structure is the (k, l)-plex, where a vertex set forms a k-plex in at least l timestamps [56].

There are some other related studies include (1) the CSM-related problems on static graphs (e.g., k-plex [7, 8], k-core [3, 13], k-truss [15, 23, 49], γ-quasi-clique [62], k-defective clique [6, 17, 18], and densest subgraph [32]); and (2) queries (that are not related to CSMs) on temporal graphs (e.g., shortest path [54], historical connectivity between two vertices [45], reachability [43, 53], shortest path [54], communication motifs [21], and structural diversity [11]).

We note that our index is theoretically grounded in CSMs that satisfy both the *non-overlapping* and *monotonic* properties. This scope covers a broad range of connectivity- and core-based structures, including connected components, k-cores, connected k-cores, and k-ECCs. It also extends to variants on specialized graphs, such as (α, β)-cores in bipartite graphs and strongly connected components in directed graphs. However, our method does not support CSMs that permit arbitrary overlaps and lack a strict containment hierarchy under edge insertions, such as quasi-cliques or k-plexes. Within the extensive list of CSMs, the monotonic and non-overlapping models constitute a dominant and fundamental class. This is because (1) monotonicity guarantees that communities are preserved or can grow when the graph is expanded, ensuring consistent community membership as the graph evolves; and (2) the non-overlapping property enables a strict partitioning of the vertices in the graph, which is crucial for ensuring unambiguous community membership. As a result, CSMs satisfying these properties typically admit polynomial-time (often linear-time) solutions, making them the standard choice for scalable mining on massive graphs. Conversely, relaxation models lacking these properties often lead to NP-hard problems (e.g., maximum k-plex problem), restricting their utility in large-scale system applications. By targeting this class, our index covers many of the widely adopted and algorithmically tractable definitions in the literature.

6 Conclusion

In this paper, we study the problem of historical non-overlapping and monotonic cohesive subgraph models query (Historical-CSM), which includes the well-known Historical-CC and Historical-Core queries. To solve the problem, we propose two unified index-based frameworks, called `AF-3DT` and `AF-BTAS`. Finally, extensive experiments are conducted and the results demonstrate the practical superiority of our method `AF-BTAS` over the state-of-the-art methods for answering Historical-CC and Historical-Core queries. In the future, we will explore the possibility of adapting our unified framework to other types of CSMs.

Acknowledgments

Shengxin Liu is supported by the Guangdong Basic and Applied Basic Research Foundation (2025A1515060015). Xun Zhou is supported by the National Natural Science Foundation of China (62472125), Guangdong Basic and Applied Basic Research Foundation (2025A1515011258), Key Technologies R&D Program of Guangdong Province (2026B0909060001) and Shenzhen Science and Technology Programs (GXWD20231128102922001, ZDCY20250901111705007, ZDSYS20230626091203008). Cheng Long is supported in part by the Ministry of Education, Singapore, under its Academic Research Fund (Tier 2 Award MOE-T2EP20224-0011 and Tier 1 Award (RG20/24)). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

References

[1] Technical Report and Source Code. https://github.com/blue-fox-maker/Querying-Cohesive-Subgraphs-in-Temporal-Graphs.

[2] Wen Bai, Yadi Chen, and Di Wu. 2020. Efficient Temporal Core Maintenance of Massive Graphs. *Information Sciences* 513 (2020), 324–340.

[3] Vladimir Batagelj and Matjaž Zaveršnik. 2003. An $O(m)$ Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003).

[4] Larissa M Batrancea, Ömer Akgüller, Mehmet Ali Balcı, and Anca Nichita. 2024. Financial Network Communities and Methodological Insights: A Case Study for Borsa Istanbul Sustainability Index. *Humanities and Social Sciences Communications* 11, 1 (2024), 1–27.

[5] Ulrik Brandes. 2005. *Network Analysis: Methodological Foundations.* Springer Science & Business Media.

[6] Lijun Chang. 2023. Efficient Maximum k-Defective Clique Computation with Improved Time Complexity. *Proceedings of the ACM on Management of Data (SIGMOD)* 1, 3 (2023), 1–26.

[7] Lijun Chang, Mouyi Xu, and Darren Strash. 2022. Efficient Maximum k-Plex Computation over Large Sparse Graphs. *Proceedings of the VLDB Endowment* 16, 2 (2022), 127–139.

[8] Lijun Chang and Kai Yao. 2024. Maximum k-Plex Computation: Theory and Practice. *Proceedings of the ACM on Management of Data (SIGMOD)* 2, 1 (2024), 1–26.

[9] Bernard Chazelle. 1990. Lower Bounds for Orthogonal Range Searching: Part II. The Arithmetic Model. *J. ACM* 37 (1990), 439–463.

[10] Kaiyu Chen, Dong Wen, Wentao Li, Zhengyi Yang, and Wenjie Zhang. 2024. On Compressing Historical Cliques in Temporal Graphs. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*. 37–53.

[11] Kaiyu Chen, Dong Wen, Wenjie Zhang, Ying Zhang, Xiaoyang Wang, and Xuemin Lin. 2024. Querying Structural Diversity in Streaming Graphs. *Proceedings of the VLDB Endowment* 17, 5 (2024), 1034–1046.

[12] Yuexing Chen, Maoxi Li, Mengying Shu, Wenyu Bi, and Siwei Xia. 2024. Multi-Modal Market Manipulation Detection in High-Frequency Trading Using Graph Neural Networks. *Journal of Industrial Engineering and Applied Science* 2, 6 (2024), 111–120.

[13] James Cheng, Yiping Ke, Shumo Chu, and M Tamer Özsu. 2011. Efficient Core Decomposition in Massive Networks. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 51–62.

[14] Martino Ciaperoni, Edoardo Galimberti, Francesco Bonchi, Ciro Cattuto, Francesco Gullo, and Alain Barrat. 2020. Relevance of Temporal Cores for Epidemic Spread in Temporal Networks. *Scientific Reports* 10, 1 (2020), 12529.

[15] Jonathan Cohen. 2008. Trusses: Cohesive Subgraphs for Social Network Analysis. *National Security Agency Technical Report* 16, 3.1 (2008).

[16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms.* MIT press.

[17] Qiangqiang Dai, Ronghua Li, Donghang Cui, and Guoren Wang. 2024. Theoretically and Practically Efficient Maximum Defective Clique Search. *Proceedings of the ACM on Management of Data (SIGMOD)* 2, 4 (2024), 1–27.

[18] Qiangqiang Dai, Rong-Hua Li, Meihao Liao, and Guoren Wang. 2023. Maximal Defective Clique Enumeration. *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)* 1, 1 (2023), 1–26.

[19] Edoardo Galimberti, Alain Barrat, Francesco Bonchi, Ciro Cattuto, and Francesco Gullo. 2018. Mining (Maximal) Span-Cores from Temporal Networks. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*. 107–116.

[20] Bin Guo and Emil Sekerinski. 2024. Simplified Algorithms for Order-Based Core Maintenance. *The Journal of Supercomputing* 80, 13 (2024), 19592–19623.

[21] Saket Gurukar, Sayan Ranu, and Balaraman Ravindran. 2015. Commit: A Scalable Approach to Mining Communication Motifs from Dynamic Networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 475–489.

[22] Chuhan Hu, Ming Zhong, Yuanyuan Zhu, Tieyun Qian, Ting Yu, Hongyang Chen, Mengchi Liu, and X Yu Jeffrey. 2024. Querying Cohesive Subgraph Regarding Span-Constrained Triangles on Temporal Graphs. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 3338–3350.

[23] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-Truss Community in Large and Dynamic Graphs. In *Proceedings of the ACM on Management of Data (SIGMOD)*. 1311–1322.

[24] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. 2015. k-Core Decomposition of Large Networks on a Single PC. *Proceedings of the VLDB Endowment* 9, 1 (2015), 13–23.

[25] Udayan Khurana and Amol Deshpande. 2013. Efficient Snapshot Retrieval over Historical Graph Data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 997–1008.

[26] Rong-Hua Li, Jiao Su, Lu Qin, Jeffrey Xu Yu, and Qiangqiang Dai. 2018. Persistent Community Search in Temporal Networks. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 797–808.

[27] Shunyang Li, Kai Wang, Xuemin Lin, Wenjie Zhang, Yizhang He, and Long Yuan. 2024. Querying Historical Cohesive Subgraphs over Temporal Bipartite Graphs. In *Proceedings of the IEEE International Conference on Data Engineering*

(ICDE). 2503–2516.

[28] Yuan Li, Jinsheng Liu, Huiqun Zhao, Jing Sun, Yuhai Zhao, and Guoren Wang. 2021. Efficient Continual Cohesive Subgraph Search in Large Temporal Graphs. *World Wide Web* 24 (2021), 1483–1509.

[29] Longlong Lin, Pingpeng Yuan, Rong-Hua Li, Chunxue Zhu, Hongchao Qin, Hai Jin, and Tao Jia. 2024. QTCS: Efficient Query-Centered Temporal Community Search. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1187–1199.

[30] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2020. Efficient (α, β)-Core Computation in Bipartite Graphs. *The VLDB Journal* 29, 5 (2020), 1075–1099.

[31] Quintino Francesco Lotito and Alberto Montresor. 2020. Efficient Algorithms to Mine Maximal Span-Trusses from Temporal Graphs. *arXiv preprint arXiv:2009.01928* (2020).

[32] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks VS Lakshmanan, Wenjie Zhang, and Xuemin Lin. 2021. On Directed Densest Subgraph Discovery. *ACM Transactions on Database Systems (TODS)* 46, 4 (2021), 1–45.

[33] Qiuyang Mang, Jingbang Chen, Hangrui Zhou, Yu Gao, Yingli Zhou, Richard Peng, Yixiang Fang, and Chenhao Ma. 2025. Efficient Historical Butterfly Counting in Large Temporal Bipartite Networks via Graph Structure-aware Index. *Proceedings of the VLDB Endowment* 18, 6 (2025), 1607–1620.

[34] Naoki Masuda and Renaud Lambiotte. 2016. *A Guide to Temporal Networks*. World Scientific.

[35] Edward M. McCreight. 1985. Priority Search Trees. *SIAM J. Comput.* 14, 2 (1985), 257–276.

[36] Mark Newman. 2010. *Networks: An Introduction*. OUP Oxford.

[37] Vladimír Pažitka, Michael Urban, and Dariusz Wójcik. 2021. Connectivity and Growth: Financial Centres in Investment Banking Networks. *Environment and Planning A: Economy and Space* 53, 7 (2021), 1789–1809.

[38] Tianhao Peng, Haitao Yuan, Yongqi Zhang, Yuchen Li, Peihong Dai, Qunbo Wang, Senzhang Wang, and Wenjun Wu. 2025. TagRec: Temporal-Aware Graph Contrastive Learning With Theoretical Augmentation for Sequential Recommendation. *IEEE Transactions on Knowledge and Data Engineering* 37, 5 (2025), 3015–3029.

[39] Hongchao Qin, Rong-Hua Li, Ye Yuan, Yongheng Dai, and Guoren Wang. 2023. Densest Periodic Subgraph Mining on Large Temporal Graphs. *IEEE Transactions on Knowledge and Data Engineering* 35, 11 (2023), 11259–11273.

[40] Hongchao Qin, Rong-Hua Li, Ye Yuan, Guoren Wang, Lu Qin, and Zhiwei Zhang. 2022. Mining Bursting Core in Large Temporal Graphs. *Proceedings of the VLDB Endowment* 15, 13 (2022), 3911–3923.

[41] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. 2011. On Querying Historical Evolving Graph Sequences. *Proceedings of the VLDB Endowment* 4, 11 (2011), 726–737.

[42] Konstantinos Semertzidis and Evaggelia Pitoura. 2016. Durable Graph Pattern Queries on Historical Graphs. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 541–552.

[43] Konstantinos Semertzidis, Evaggelia Pitoura, and Kostas Lillis. 2015. TimeReach: Historical Reachability Queries on Evolving Graphs. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 121–132.

[44] Daniel D. Sleator and Robert E. Tarjan. 1981. A Data Structure for Dynamic Trees. In *Proceedings of the Annual ACM Symposium on Theory of Computing (STOC)*. 114–122.

[45] Jingyi Song, Dong Wen, Lantian Xu, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2024. On Querying Historical Connectivity in Temporal Graphs. *Proceedings of the ACM on Management of Data (SIGMOD)* 2, 3 (2024), 1–25.

[46] Yuting Tan, Chunhua Wang, Junfeng Zhou, Ming Du, Guohao Sun, and Weiguo Zheng. 2024. Efficient Querying k-Trusses on Temporal Graphs. *Information Processing & Management* (2024), 104014.

[47] Yifu Tang, Jianxin Li, Nur Al Hasan Haldar, Ziyu Guan, Jiajie Xu, and Chengfei Liu. 2022. Reliable Community Search in Dynamic Networks. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2826–2838.

[48] Anxin Tian, Alexander Zhou, Yue Wang, Xun Jian, and Lei Chen. 2024. Efficient Index for Temporal Core Queries over Bipartite Graphs. *Proceedings of the VLDB Endowment* 17, 11 (2024), 2813–2825.

[49] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *Proceedings of the VLDB Endowment* 5, 9 (2012), 812–823.

[50] Zhi Wang, Ming Zhong, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffrey Xu Yu. 2025. On More Efficiently and Versatilely Querying Historical k-Cores. *Proceedings of the VLDB Endowment* 18, 5 (2025), 1335–1347.

[51] Stanley Wasserman. 1994. Social Network Analysis: Methods and Applications. *The Press Syndicate of the University of Cambridge* (1994).

[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 *Proceedings of the IEEE 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] 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.

[55] Huanhuan Wu, James Cheng, Yi Lu, Yiping Ke, Yuzhen Huang, Da Yan, and Hejun Wu. 2015. Core Decomposition in Large Temporal Graphs. In *Proceedings of the IEEE International Conference on Big Data (BigData)*. 649–658.

[56] Yanping Wu, Renjie Sun, Xiaoyang Wang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2024. Efficient Maximal Temporal Plex Enumeration. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 3098–3110.
[57] Haoxuan Xie, Yixiang Fang, Yuyang Xia, Wensheng Luo, and Chenhao Ma. 2023. On Querying Connected Components in Large Temporal Graphs. *Proceedings of the ACM on Management of Data (SIGMOD)* 1, 2 (2023), 1–27.
[58] Junyong Yang, Ming Zhong, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffery Xu Yu. 2023. Scalable Time-Range k-Core Query on Temporal Graphs. *Proceedings of the VLDB Endowment* 16, 5 (2023), 1168–1180.
[59] Junyong Yang, Ming Zhong, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffrey Xu Yu. 2024. Evolution Forest Index: Towards Optimal Temporal k-Core Component Search via Time-Topology Isomorphic Computation. *Proceedings of the VLDB Endowment* 17, 11 (2024), 2840–2853.
[60] Yi Yang, Da Yan, Huanhuan Wu, James Cheng, Shuigeng Zhou, and John CS Lui. 2016. Diversified Temporal Subgraph Pattern Mining. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 1965–1974.
[61] Kaiqiang Yu and Cheng Long. 2021. Graph mining meets fake news detection. In *Data Science for Fake News: Surveys and Perspectives*. Springer, 169–189.
[62] Kaiqiang Yu and Cheng Long. 2023. Fast Maximal Quasi-Clique Enumeration: A Pruning and Branching Co-Design Approach. *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)* 1, 3 (2023), 1–26.
[63] Michael Yu, Dong Wen, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2021. On Querying Historical k-Cores. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2033–2045.
[64] Yuanhang Yu, Dong Wen, Michael Yu, Lu Qin, Ying Zhang, Wenjie Zhang, and Xuemin Lin. 2025. Querying Historical k-Cores in Large Temporal Graphs. *The VLDB Journal* 34, 2 (2025), 26.
[65] Ming Zhong, Junyong Yang, Yuanyuan Zhu, Tieyun Qian, Mengchi Liu, and Jeffrey Xu Yu. 2024. A Unified and Scalable Algorithm Framework of User-Defined Temporal $(k, \mathcal{X})$-Core Query. *IEEE Transactions on Knowledge and Data Engineering* 36, 7 (2024), 2831–2845.

Received October 2025; revised January 2026; accepted February 2026