Time-Constrained Continuous Subgraph Matching Using Temporal Information for Filtering and Backtracking

Seunghwan Min[†], Jihoon Jang[†], Kunsoo Park[*†], Dora Giammarresi[‡], Giuseppe F. Italiano[§], Wook-Shin Han[¶]
[†]*Seoul National University, South Korea* [‡]*Università Roma "Tor Vergata"*
[§]*LUISS University, Italy* [¶]*Pohang University of Science and Technology (POSTECH), South Korea*
[†]{shmin,jhjang,kpark}@theory.snu.ac.kr [‡]giammarr@mat.uniroma2.it
[§]gitaliano@luiss.it [¶]wshan@dblab.postech.ac.kr

***Abstract*—Real-time analysis of graphs containing temporal information, such as social media streams, Q&A networks, and cyber data sources, plays an important role in various applications. Among them, detecting patterns is one of the fundamental graph analysis problems. In this paper, we study time-constrained continuous subgraph matching, which detects a pattern with a strict partial order on the edge set in real-time whenever a temporal data graph changes over time. We propose a new algorithm based on two novel techniques. First, we introduce a filtering technique called time-constrained matchable edge that uses temporal information for filtering with polynomial space. Second, we develop time-constrained pruning techniques that reduce the search space by pruning some of the parallel edges in backtracking, utilizing temporal information. Extensive experiments on real and synthetic datasets show that our approach outperforms the state-of-the-art algorithm by up to two orders of magnitude in terms of query processing time.**

***Index Terms*—time-constrained continuous subgraph matching, temporal order, time-constrained matchable edge, max-min timestamp, time-constrained pruning**

I. Introduction

Graphs are structures widely used to represent relationships between objects. For example, communication between users on social media, financial transactions between bank accounts, and computer network traffic can be modeled as graphs. In many cases, relationships between objects in real-world graph datasets contain temporal information about when they occurred. A graph containing temporal information is called a temporal graph. Social media streams [1], [2], Q&A networks [3], and cyber data sources [4], [5] such as computer network traffic and financial transaction networks, are examples of temporal graphs.

There has been extensive research on the efficient analysis of temporal graphs such as graph mining [6], [7], graph simulation [8], [9], motif counting [10], [11], subgraph matching [12]–[14], and network clustering [15]. In this paper, we model a temporal data graph as a streaming graph [16] and address continuous subgraph matching on temporal graphs called *time-constrained continuous subgraph matching* [17]. Given a temporal data graph G and a temporal query graph q, the time-constrained continuous subgraph matching problem is to find all matches of q that are isomorphic to q and satisfy the temporal order of q over the streaming graph of G. By considering the temporal order along with the topological structure, one can more clearly represent various real-world scenarios, such as tracking the flow of money in financial transaction networks and monitoring the flow of packets in computer network traffic. Thus, time-constrained continuous subgraph matching is a fundamental problem that can play an important role in applications such as money laundering detection and cyber attack detection [17].

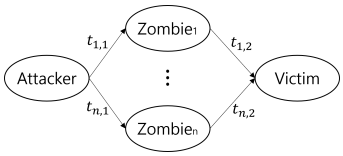
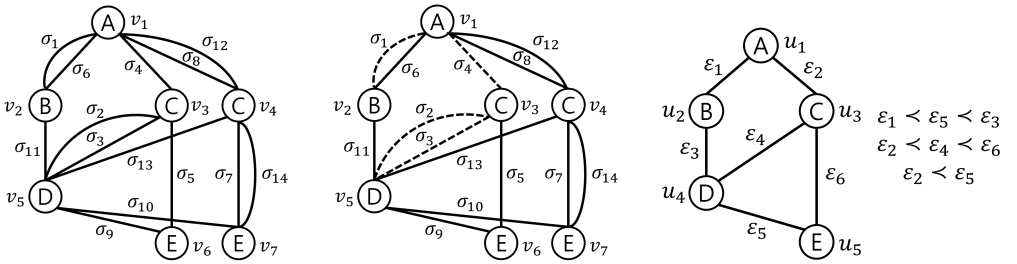


Fig. 1: DDoS attack pattern in network traffic

The query depicted in Figure 1 illustrates the essential pattern in DDoS attacks. Each zombie computer (Zombie$_i$) is infected through one of multiple contamination paths, which is not shown in the query graph. The zombies receive commands from the attacker ($t_{i,1}$), and then they attack the victim ($t_{i,2}$). Thus there is a temporal order $t_{i,1} \prec t_{i,2}$ for each i. Real-world DDoS attacks can be more complex than the query pattern in Figure 1, but they include the query pattern as a subgraph because the query pattern is the core of DDoS attacks. Hence, by detecting and recognizing the query pattern in network traffic data, one can identify real-world DDoS attacks that include the query pattern as a subgraph (note that one can identify the attacker by using this query pattern!). US communications company Verizon has analyzed 100,000 security incidents from the past decade that reveal that 90% of the incidents fall into ten attack patterns [18], which can be described as graph patterns.

Many researchers have studied the continuous subgraph matching problem on non-temporal graphs [19]–[23]. It is possible to solve the time-constrained continuous subgraph matching problem with these algorithms. However, these algorithms also find matches that do not satisfy the temporal order, which we don't need. Therefore, we have to remove

*Contact author



(a) Temporal data graph G where edge σ_i arrives at time i

(b) G at $t = 14$ with window $\delta = 10$

(c) Query graph q with temporal order constraints

Fig. 2: A running example

unnecessary matches at the end, which is inefficient. Recently, several studies have been conducted to find matches by taking the temporal order into account [17], [24]. However, existing work shows limitations in that it requires exponential space to store all partial matches or utilize only temporal order constraints between incident edges.

A temporal graph typically has many parallel edges because each edge has temporal information. In the case of computer network traffic or financial transaction networks, there are many edges (i.e., data transmissions or transaction records) between two nodes. Multigraphs (and thus algorithms for multigraphs) are valuable in such applications due to their unique feature of parallel edges, which sets them apart from simple graphs [25]–[27]. In this paper, we propose a new time-constrained continuous subgraph matching algorithm `TCM` for temporal graphs that adopts two novel techniques which fully exploit temporal information in both filtering and backtracking.

- To address the limitations of existing approaches, we propose an efficient filtering technique named *time-constrained matchable edge* that can make full use of temporal relationships between non-incident edges for filtering. Our filtering approach is based on the basic idea that in order to match a query edge e with a data edge $\overline{e}$, there must exist a path starting from $\overline{e}$ that satisfies the temporal order of e along every possible path starting from e in the query graph. Consider the process of applying the previous idea to the temporal data graph G in Figure 2a and the temporal query graph q in Figure 2c. The ID of each edge in G indicates the arrival time of the edge (i.e., edge σ_i arrives at time i). Since there is no path starting from σ_4 in Figure 2a satisfying the temporal order constraint $\varepsilon_2 \prec \varepsilon_4$ for the path $\varepsilon_2 \rightarrow \varepsilon_4$ in Figure 2c, we can safely exclude σ_4 from the matching candidates of ε_2.
  We introduce a data structure called *max-min timestamp* with polynomial space, which can determine whether an edge is filtered by the time-constrained matchable edge and can be updated efficiently for each graph update. For example, consider the situation of determining whether there is a path starting from σ_1 that satisfies the temporal order of ε_1 ($\varepsilon_1 \prec \varepsilon_5$ and $\varepsilon_1 \prec \varepsilon_3$) for the path of $\varepsilon_1 \rightarrow \varepsilon_3 \rightarrow \varepsilon_5$. The max-min timestamp for ε_1 at σ_1 stores the maximum value among the minimum timestamps of the corresponding edges following ε_1 (i.e., ε_3 and ε_5) from the various paths starting from σ_1. Since the minimum values of timestamps in two paths $\sigma_{11} \rightarrow \sigma_9$ and $\sigma_{11} \rightarrow \sigma_{10}$ (corresponding to the path $\varepsilon_3 \rightarrow \varepsilon_5$) are 9 and 10, respectively, the max-min timestamp for ε_1 at σ_1 stores the maximum value of 10. We can determine that there is a path starting from σ_1 that satisfies the temporal order of ε_1 for the path of $\varepsilon_1 \rightarrow \varepsilon_3 \rightarrow \varepsilon_5$ because the max-min timestamp for ε_1 at σ_1 is greater than 1. We also introduce how to update the max-min timestamp efficiently in Section IV-C.
- Second, we develop a set of time-constrained pruning techniques to reduce the search space in backtracking. Unlike non-temporal graphs, parallel edges between two vertices in a temporal graph are distinguished because they have different timestamps. We can prune some of the parallel edges by utilizing temporal relationships. When finding a match for query graph q (Figure 2c) in the data graph G (Figure 2a) that satisfies the temporal order of q, assume that we try to match query edges ε_5, ε_6, ε_4, ε_3, ε_1, and ε_2 in this order. Now, consider a situation where ε_5 and ε_6 are matched to σ_9 and σ_5, respectively, and ε_4 is the next edge to be matched to either σ_2 or σ_3. Given that $\varepsilon_4 \prec \varepsilon_6$ holds true regardless of whether σ_2 or σ_3 is chosen, the temporal order constraint that needs to be satisfied for ε_4 is reduced to $\varepsilon_2 \prec \varepsilon_4$. To increase the likelihood of finding the desired match, it is more advantageous to match ε_4 to an edge with a larger timestamp. Consequently, we conduct backtracking in the direction of matching ε_4 with σ_3 prior to matching it with σ_2. The backtracking process ends without finding the desired match due to the absence of any edge that can be matched with ε_2. When we retrace our steps to the point at which we have to match ε_4 to σ_2 instead of σ_3, we can deduce from the previously unsuccessful backtracking attempt that any further backtracking will not yield the desired match. Therefore, we prune the search space that matches ε_4 to σ_2 and explore other search spaces.
  We classify the temporal relationship between the query edge e and other query edges into three cases: no temporally related edges, all edges with the same temporal

relationship, and the remaining case. For each case, we demonstrate a situation where a match cannot exist and use it to prune edges.

We conduct experiments on six real and synthetic datasets comparing our algorithm with existing algorithms. Experimental results show that TCM outperforms existing approaches by up to two orders of magnitude in terms of query processing time.

The rest of the paper is organized as follows. Section II defines the problem statement and provides related work. Section III gives an overview of our algorithm. Section IV introduces our filtering technique and Section V presents techniques to prune out a part of search space. Section VI shows the results of our performance evaluation. Finally, Section VII concludes the paper.

II. Preliminaries

In this paper, we focus on undirected and vertex-labeled graphs. Our techniques can be easily extended to directed graphs with multiple labels on vertices or edges.

Definition II.1. [14], [28] A *temporal graph* $G = (V(G), E(G), L_G, T_G)$ consists of a set $V(G)$ of vertices, a set $E(G)$ of edges, a labeling function $L_G : V(G) \to \Sigma$ that assigns a label to each vertex from the set Σ of labels, and a timing function $T_G : E(G) \to \mathbb{N}$ that assigns a timestamp to each edge. The timestamp of an edge indicates when the edge arrived. We represent timestamps as natural numbers.

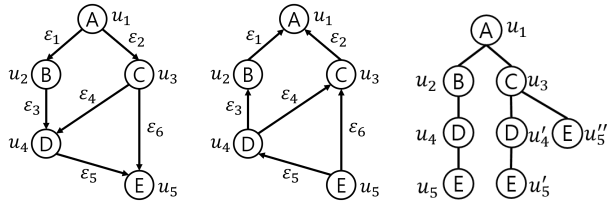
A temporal graph can have parallel edges with different timestamps between two vertices. To distinguish edges with different timestamps between vertices u and v, we denote each edge as (u, v, t), where t is the timestamp of the edge. Note that there are two edges $\sigma_1 = (v_1, v_2, 1)$ and $\sigma_6 = (v_1, v_2, 6)$ between v_1 and v_2 in Figure 2a.

Definition II.2. A *temporal query graph* q is defined as $(V(q), E(q), L_q, \prec)$ where $(V(q), E(q), L_q)$ is a graph and $\prec$ is a strict partial order relation on $E(q)$ called the *temporal order*. We say that e_1 and e_2 are temporally related when $e_1 \prec e_2$ or $e_2 \prec e_1$.

Definition II.3. Given a temporal query graph $q = (V(q), E(q), L_q, \prec)$ and a temporal data graph $G = (V(G), E(G), L_G, T_G)$, a *time-constrained embedding* of q in G is a mapping $M : V(q) \cup E(q) \to V(G) \cup E(G)$ such that (1) M is injective, (2) $L_q(u) = L_G(M(u))$ for every $u \in V(q)$, (3) $M(e) \in E(G)$ is an edge between $M(u)$ and $M(u')$ for every $e = (u, u') \in E(q)$ (i.e., $M(e) = (M(u), M(u'), t)$), and (4) $e_1 \prec e_2 \Rightarrow T_G(M(e_1)) < T_G(M(e_2))$ for any two $e_1, e_2 \in E(q)$.

A mapping that satisfies (2) and (3) is called a *homomorphism*. A homomorphism satisfying (1) is called an *embedding*. Homomorphism and embedding are widely used in graph matching problems. In this paper, we use embeddings as matching semantics.

Example II.1. In a temporal data graph G in Figure 2a and a temporal query graph q in Figure 2c, two mappings



(a) Query DAG $\hat{q}$ (b) Query DAG $\hat{q}^{-1}$ (c) Path tree of $\hat{q}$

Fig. 3: DAGs of q in Figure 2c for the running example

$\{(\varepsilon_1, \sigma_1), (\varepsilon_2, \sigma_8), (\varepsilon_3, \sigma_{11}), (\varepsilon_4, \sigma_{13}), (\varepsilon_5, \sigma_{10}), (\varepsilon_6, \sigma_{14})\}$ and $\{(\varepsilon_1, \sigma_6), (\varepsilon_2, \sigma_8), (\varepsilon_3, \sigma_{11}), (\varepsilon_4, \sigma_{13}), (\varepsilon_5, \sigma_{10}), (\varepsilon_6, \sigma_{14})\}$ are embeddings and also time-constrained embeddings of q in G. In contrast, a mapping $M = \{(\varepsilon_1, \sigma_1), (\varepsilon_2, \sigma_4), (\varepsilon_3, \sigma_{11}), (\varepsilon_4, \sigma_2), (\varepsilon_5, \sigma_9), (\varepsilon_6, \sigma_5)\}$ is an embedding of q in G, but it is not a time-constrained embedding because $\varepsilon_2 \prec \varepsilon_4$ in q but $T_G(M(\varepsilon_2)) = T_G(\sigma_4) = 4 \not< T_G(M(\varepsilon_4)) = T_G(\sigma_2) = 2$ in G. When representing the mapping M, we omit a pair of a vertex $u \in V(q)$ and its image $M(u)$ for simplicity of notation.

In order not to find time-constrained embeddings where the time interval between the minimum and maximum timestamps of its edges is too long, we model a temporal graph as a streaming graph with a time window as in [17]. For a given time window δ and current time t, the edges with timestamp less than or equal to $t - \delta$ expire. In other words, we keep only the edges that arrive between $(t - \delta, t]$ and find time-constrained embeddings over these edges.

Example II.2. Figure 2b shows the temporal graph at time $t = 14$ with the time window $\delta = 10$. At time $t = 14$, the edge σ_{14} arrives and the edge σ_4 expires. When σ_{14} arrives, a time-constrained embedding $\{(\varepsilon_1, \sigma_6), (\varepsilon_2, \sigma_8), (\varepsilon_3, \sigma_{11}), (\varepsilon_4, \sigma_{13}), (\varepsilon_5, \sigma_{10}), (\varepsilon_6, \sigma_{14})\}$ occurs. In this case, since the edge σ_1 has already expired, a time-constrained embedding $\{(\varepsilon_1, \sigma_1), (\varepsilon_2, \sigma_8), (\varepsilon_3, \sigma_{11}), (\varepsilon_4, \sigma_{13}), (\varepsilon_5, \sigma_{10}), (\varepsilon_6, \sigma_{14})\}$ does not occur. At time $t = 16$, the edge σ_6 expires and the time-constrained embedding containing σ_6 also expires.

Problem Statement. Given a temporal data graph G, a temporal query graph q, and a time window δ, the *time-constrained continuous subgraph matching problem* is to find all time-constrained embeddings of q that occur or expire with the time window δ according to the arrival or expiration of each edge in G.

Theorem II.1. [17] The time-constrained continuous subgraph matching problem is NP-hard.

One naive solution to this problem is to solve the continuous subgraph matching (i.e., find embeddings without considering the temporal order) and then exclude the embeddings that do not satisfy the temporal order to obtain the time-constrained embeddings. In general, it would be better to filter the edges in consideration of the temporal order or prune the partial embeddings that do not fit the constraint rather than the naive solution.

In this paper, we will use a directed acyclic graph (DAG) and a weak embedding [29] as a tool to filter edges using

temporal order (Section IV). A *directed acyclic graph* (DAG) $\hat{q}$ is a directed graph that contains no cycles. A *root* (resp., *leaf*) of a DAG is a vertex with no incoming (resp., outgoing) edges. A DAG $\hat{q}$ is a *rooted DAG* if there is only one root. The DAG $\hat{q}$ in Figure 3a is one of the rooted DAGs that can be obtained from q in Figure 2c. Its reverse $\hat{q}^{-1}$ in Figure 3b is the same as $\hat{q}$ except that all of the edges are reversed.

In a DAG, we say that u is a *parent* of v (v is a *child* of u) if there exists a directed edge from u to v. An *ancestor* of a vertex v is a vertex which is either a parent of v or an ancestor of a parent of v. A *descendant* of a vertex v is a vertex which is either a child of v or a descendant of a child of v. When representing an edge of a DAG as a pair (u_1, u_2) of vertices, we will use the convention that the first vertex u_1 is always the parent of the second vertex u_2. For edges, we say that $e_1 = (u_1, u_1')$ is an ancestor of $e_2 = (u_2, u_2')$ if $u_1' = u_2$ or u_1' is an ancestor of u_2. For example, ε_2 is an ancestor of ε_4, ε_5, and ε_6 in Figure 3a. Let Child(u) and Parent(u) denote the children and parents of u in $\hat{q}$, respectively.

Definition II.4. Given a temporal query graph q and its DAG $\hat{q}$, e_1 is a *temporal ancestor* of e_2 in $\hat{q}$, denoted by $e_1 \rightsquigarrow_{\hat{q}} e_2$ if e_1 is an ancestor of e_2 in $\hat{q}$ and two edges have a temporal relation (i.e., $e_1 \prec e_2$ or $e_2 \prec e_1$). We say that such e_2 is a *temporal descendant* of e_1.

For simplicity of presentation, when e_1 is a temporal ancestor of e_2 in $\hat{q}$, we will only consider the case $e_1 \prec e_2$ (i.e., we omit the explanation for the case $e_2 \prec e_1$). The other case (i.e., $e_2 \prec e_1$) has been implemented in a symmetrical way.

Definition II.5. A *sub-DAG of $\hat{q}$ starting at u or e*, denoted by $\hat{q}_u$ or $\hat{q}_e$, is the subgraph of $\hat{q}$ consisting of edges of all paths starting at u or e. For example, $\hat{q}_{u_3}$ in Figure 3a is the subgraph that contains ε_4, ε_5 and ε_6, and $\hat{q}_{\varepsilon_2}$ in Figure 3a consists of $\varepsilon_2, \varepsilon_4, \varepsilon_5$ and ε_6.

Definition II.6. [29] The *path tree* of a rooted DAG $\hat{q}$ is defined as the tree $\hat{q}^T$ such that each root-to-leaf path in $\hat{q}^T$ corresponds to a distinct root-to-leaf path in $\hat{q}$, and $\hat{q}^T$ shares common prefixes of its root-to-leaf paths. Figure 3c shows the path tree of $\hat{q}$ in Figure 3a.

Definition II.7. [29] A *weak embedding* M' of $\hat{q}_u$ at $v \in V(G)$ is defined as a homomorphism of the path tree of $\hat{q}_u$ in G such that $M'(u) = v$. Similarly, a weak embedding M' of $\hat{q}_e$ at $\overline{e} \in E(G)$ is defined as a homomorphism of the path tree of $\hat{q}_e$ in G such that $M'(e) = \overline{e}$.

Table I lists the notations frequently used in this paper.

TABLE I: Frequently used notations

Symbol	Description
G	Temporal data graph
q	Temporal query graph
$\hat{q}$	Query DAG
$M(u)$ and $M(e)$	Mapping of u and e in (partial) embedding M
$\mathcal{T}(\hat{q})$	Max-min timestamps of $\hat{q}$
$EC_M(e)$	Set of candidate edges of e regarding a partial time-constrained embedding M

A. Related Work

Subgraph matching. Subgraph matching finds all embeddings of a query graph over the static data graph. Ullmann [30] first proposes a backtracking algorithm to find all embeddings of the query graph. Extensive studies have been done to improve the backtracking algorithm, such as `Turbo`$_{\texttt{iso}}$ [31], `CFL-Match` [32], `DAF` [29] and `VEQ` [33].

Continuous subgraph matching finds all newly occurred or expired embeddings of the query graph over the dynamic data graph. There also has been extensive studies on continuous subgraph matching problem, such as `IncIsoMatch` [19], `Graphflow` [20], `SJ-tree` [21], `TurboFlux` [22], `SymBi` [23] and `RapidFlow` [34]. `TurboFlux` proposes an auxiliary data structure called *data-centric graph*, or `DCG`, which uses a spanning tree of q to filter out candidate edges that can match each edge of the spanning tree. `SymBi` proposes an auxiliary data structure called *dynamic candidate space*, or `DCS`, which uses a query DAG of q instead of the spanning tree of q. Since it uses a query DAG, it can filter out more candidate edges by considering non-tree edges of the spanning tree of q, while achieving polynomial space. `RapidFlow` addresses and solves the following two issues in existing works: matching orders always starting from the inserted edge can lead to inefficiency, and automorphism on query graph causes duplicate computation.

Time-constrained subgraph matching. There are several studies to solve subgraph matching problem on temporal graphs, such as `TOM` [14], `Timing` [17] and `Hasse` [24].

`TOM` finds all the time-constrained embeddings within a time window over the static temporal graph. It builds an index structure called *Temporal-order tree* (TO-tree for short) to filter the candidate edges that can be matched to the query edge, by checking adjacency and temporal order between matching candidates. After the TO-tree construction, it enumerates all embeddings of the query graph in an edge-by-edge expansion manner. It does not fully utilize the temporal order constraints while filtering because it does not consider the constraints between non-incident edges.

`Timing` and `Hasse` solves the time-constrained continuous subgraph matching problem. They decompose a query graph into the subqueries, and store all partial embeddings of each subquery in its index structure. When a partial embedding occurs or expires due to the edge arrival or expiration, they compute new or expired embeddings of the query graph by joining the partial embeddings. Since they materialize the partial embeddings, they may require exponential space to the size of the query graph.

III. Overview of our algorithm

Algorithm 1 shows the overview of our time-constrained continuous subgraph matching algorithm. Given a temporal data graph G, a temporal query graph q, and a time window δ, our algorithm consists of the following steps.

Building a query DAG. We first build a rooted DAG $\hat{q}$ from q by assigning directions to the edges in q. The built query DAG

Algorithm 1: Time-Constrained Continuous Subgraph Matching

Input: a temporal data graph G, a temporal query graph q, and a time window δ
Output: all time-constrained embeddings

1 $S_{max} \leftarrow -\infty$;
2 **foreach** $r \in V(q)$ **do**
3 $(\hat{q}_r, S_r) \leftarrow$ `BuildDAG`(q, r);
4 **if** $S_{max} < S_r$ **then**
5 $S_{max} \leftarrow S_r$;
6 $\hat{q} \leftarrow \hat{q}_r$;
7 $g \leftarrow$ an empty temporal graph;
8 $L \leftarrow \{(\overline{e}, t, +), (\overline{e}, t+\delta, -) \mid \overline{e} \in E(G), t = T_G(\overline{e})\}$;
9 **while** $L \neq \emptyset$ **do**
10 $(\overline{e}, t, op) \leftarrow$ pop from L where t is minimum;
11 **if** $op = +$ **then**
12 `InsertEdge`$(g, \overline{e})$;
13 $E^+_{DCS} \leftarrow$`TCMInsertion`$(g, \hat{q}, \overline{e})$;
14 `DCSInsertion`$(\texttt{DCS}, E^+_{DCS})$;
15 `FindMatches`$(\texttt{DCS}, \overline{e}, \emptyset)$;
16 **if** $op = -$ **then**
17 `DeleteEdge`$(g, \overline{e})$;
18 $E^-_{DCS} \leftarrow$`TCMDeletion`$(g, \hat{q}, \overline{e})$;
19 `DCSDeletion`$(\texttt{DCS}, E^-_{DCS})$;
20 `FindMatches`$(\texttt{DCS}, \overline{e}, \emptyset)$;

$\hat{q}$ is used to filter edges according to the definition of "time-constrained matchable edge" (Section IV-A). Our filtering process considers each ordered pair of edges which are in the temporal ancestor-descendant relationship in the query DAG. Therefore, we build a query DAG through a greedy algorithm so that many ordered pairs can be considered for filtering. Procedure `BuildDAG` takes a query graph q and a vertex $r \in V(q)$ as inputs, and outputs a query DAG $\hat{q}_r$ (whose root is r, and that is obtained through the greedy algorithm) and its score S_r (Section IV-B). Here, the score S_r of DAG $\hat{q}_r$ is the number of the ordered pairs of edges in the temporal ancestor-descendant relationship in $\hat{q}_r$. We try every vertex once as the root and use the query DAG with the highest score (Lines 1–6).

L is a set that contains events of the arrival and expiration of edges in G. Events for edge arrival are denoted by $+$ and events for edge expiration are denoted by $-$. After building the query DAG, our algorithm performs steps 2 and 3 for each event in L in chronological order.

Updating the data structures. For each arrived edge $\overline{e}$, we update the data structures *DCS* and *max-min timestamp*. First, we update the data graph g that represents the current state of G by inserting the edge into g (Line 12). We store the edges in an adjacency list in the chronologically sorted order. When inserting an edge, therefore, simply adding it to the end of the adjacency list is sufficient. Next, we update the max-min timestamp by invoking procedure `TCMInsertion` (Line 13). This procedure returns a set E^+_{DCS} consisting of pairs of edges $e' \in E(q)$ and $\overline{e'} \in E(g)$ such that e' newly becomes a time-constrained matchable edge of $\overline{e'}$ due to the arrived edge. Finally, we update the auxiliary data structure `DCS` with E^+_{DCS} (Line 14). `DCS` is the data structure used in [23] to solve the continuous subgraph matching problem. It stores an intermediate result of filtering vertices using the weak embedding of a DAG, which is a necessary condition for embedding. When filtering vertices, `DCS` in [23] considers all pairs of edges in g and q, but our `DCS` considers only the edge pairs remaining after the filtering by time-constrained matchable edges. Because fewer edge pairs are used for our `DCS`, the overall running time to update `DCS` and the number of remaining candidate vertices after filtering are reduced, compared to `DCS` in [23]. In a similar process, the data graph g and the data structures can be updated for expired edges (Lines 17–19). `DeleteEdge` can be done by removing the edge from the front of the adjacency list. The pseudocodes of `DCSInsertion` and `DCSDeletion` are available in [23], as these operations are related to the update of the data structure `DCS` proposed in [23].

Matching. After updating the data structures, we find new or expired time-constrained embeddings from the updated data structures by calling the backtracking procedure `FindMatches`. `FindMatches` is based on the backtracking algorithm that solve the continuous subgraph matching problem. Since the continuous subgraph matching problem deals with non-temporal graphs, it doesn't matter which one among parallel edges a query edge matches. So the continuous subgraph matching algorithms generally assume simple graphs and focus only on mapping of vertices. In contrast, in our problem, the search tree varies depending on which one of the parallel edges matches a query edge due to the temporal order. Therefore, we consider parallel edges and the temporal order during backtracking (Section V).

IV. Filtering by Temporal Order

In this section, we introduce our edge filtering technique using the temporal order called time-constrained matchable edge and describe the greedy algorithm that generates a query DAG for effective filtering. Afterwards, we propose a data structure called max-min timestamp with polynomial space and an efficient way to update the data structure. This data structure is used to determine if an edge is filtered by the time-constrained matchable edge.

A. Time-Constrained Matchable Edge

There are various ways to filter edges considering the temporal order. `Timing` [17] and `Hasse` [24] store all time-constrained partial embeddings of the temporal query graph to prune some arrived edges, but it requires an exponential storage space and time to maintain and update partial embeddings. To avoid this, the TO-tree in `TOM` [14] only stores matching candidates of each query edge and filters candidates by considering temporal order between incident query edges. However, temporal relationships between non-incident edges are not utilized for filtering. Our proposed method aims to occupy less storage space by not storing time-constrained partial embeddings and consider as many ordered pairs in the temporal order as possible for filtering.

Definition IV.1. Given a temporal data graph G, a temporal query graph q, and a query DAG $\hat{q}$, a weak embedding M'

of $\hat{q}_e$ at $\overline{e} \in E(G)$ is *a time-constrained weak embedding* (*TC-weak embedding* for short) if $T_G(M'(e)) = T_G(\overline{e}) < T_G(M'(e'))$ for every e' where $e \rightsquigarrow_{\hat{q}} e'$. We say that e is a *time-constrained matchable edge* (*TC-matchable edge* for short) of $\overline{e}$ regarding $\hat{q}$ if there exists a TC-weak embedding of $\hat{q}_e$ at $\overline{e}$.

Example IV.1. Let us consider ε_2 in q (Figure 2c) and σ_8 in G (Figure 2a). There exists a weak embedding $M'_1 = \{(\varepsilon_2, \sigma_8), (\varepsilon_4, \sigma_{13}), (\varepsilon_5, \sigma_{10}), (\varepsilon_6, \sigma_{14})\}$ of $\hat{q}_{\varepsilon_2}$ at σ_8. This weak embedding is a TC-weak embedding of $\hat{q}_{\varepsilon_2}$ at σ_8 because $T_G(\sigma_8) < T_G(M'_1(\varepsilon_4)) = T_G(\sigma_{13})$, $T_G(\sigma_8) < T_G(M'_1(\varepsilon_5)) = T_G(\sigma_{10})$, and $T_G(\sigma_8) < T_G(M'_1(\varepsilon_6)) = T_G(\sigma_{14})$ hold. Therefore, ε_2 is a TC-matchable edge of σ_8 regarding $\hat{q}$. On the other hand, there is no TC-weak embedding of $\hat{q}_{\varepsilon_2}$ at σ_{12} since there is a constraint $\varepsilon_2 \prec \varepsilon_5$, but σ_9 and σ_{10}, which can be matched to ε_5, have timestamps not greater than $T_G(\sigma_{12}) = 12$. Hence, ε_2 is not a TC-matchable edge of σ_{12} regarding $\hat{q}$.

Lemma IV.1. If $e \in E(q)$ is not a TC-matchable edge of $\overline{e} \in E(G)$ regarding some DAG of q, then there is no time-constrained embedding M such that $M(e) = \overline{e}$.

Proof. Suppose that there exists a time-constrained embedding M such that $M(e) = \overline{e}$. Then, we need to show that e is a TC-matchable edge of $\overline{e}$ for any DAG $\hat{q}$ of q. For any query DAG $\hat{q}$, consider the weak embedding M' of $\hat{q}_e$ at $\overline{e}$ such that the image of each element under M' is the same as the image under M. For a temporal descendant e' of e in $\hat{q}$, $T_G(M'(e)) = T_G(\overline{e}) < T_G(M'(e'))$ holds by the definition of time-constrained embedding. Thus, M' is a TC-weak embedding of $\hat{q}_e$ at $\overline{e}$ and so e is a TC-matchable edge of $\overline{e}$ regarding $\hat{q}$. □

According to Lemma IV.1, if $e \in E(q)$ is not a TC-matchable edge of $\overline{e} \in E(G)$ regarding some DAG $\hat{q}$ of q, we can filter $\overline{e}$ from the set of data edges that are possible to match e. For example, since ε_2 is not a TC-matchable edge of σ_{12}, we do not need to map ε_2 to σ_{12} while backtracking, so the search space is reduced. In comparison to `TOM` [14], which only checks the temporal order constraints between incident edges, `TOM` does not filter σ_{12} because ε_2 and ε_5 are not incident. To enhance the filtering, we select the query DAG $\hat{q}$ containing as many temporal ancestor-descendant relationships as possible. We will describe in Section IV-B how to build the query DAG $\hat{q}$ to achieve this goal. Additionally, to prune more data edges which cannot be matched to e, we use both $\hat{q}$ and $\hat{q}^{-1}$ when filtering the candidates by Lemma IV.1.

B. Building Query DAG

Because the temporal ancestor-descendant relationship is affected by the shape of the query DAG, the results of filtering through TC-matchable edges are also affected by the shape of the query DAG. In particular, the ordered pair consisting of two edges with no path between them in the query DAG is not considered for filtering. Therefore, we propose a greedy algorithm (Algorithm 2) that builds a query DAG so that there are many ordered pairs in the temporal ancestor-descendant relationships.

Algorithm 2: BuildDAG

Input: a temporal query graph q and a root vertex r
Output: a query DAG $\hat{q}_r$ and the score S_r of $\hat{q}_r$

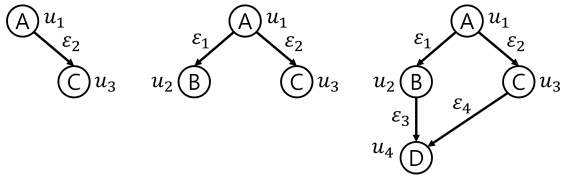
```
1  q̂_r ← an empty graph, S_r ← 0, cand ← {r};
2  Score[u] ← 0 for u ∈ V(q);
3  while cand ≠ ∅ do
4      u ← pop from cand whose Score[u] is maximum;
5      add u into V(q̂_r);
6      foreach (u, u') ∈ E(q) do
7          if u' ∈ V(q̂_r) then
8              add an edge (u', u) into E(q̂_r);
9          else
10             if u' ∉ cand then
11                 cand ← cand ∪ {u'}
12             compute Score[u'];
13     S_r ← S_r + Score[u];
14 return (q̂_r, S_r);
```

Algorithm 2 that makes a DAG with $r \in V(q)$ as its root proceeds as follows. The vertices in $cand$ are the candidates to come next in the topological order of the current DAG. $Score[u]$ stores the number of ordered pairs that will have temporal ancestor-descendant relationships when the DAG is constructed by selecting u as the next vertex from $cand$. Now, we repeat the process of selecting a query vertex u from $cand$, adding u with directed edges into the current DAG, and updating $cand$ and $Score$ until there are no more vertices in $cand$. Specifically, we greedily select u with the highest $Score[u]$ among the candidates in $cand$ so that the score of the query DAG can be high. If there is more than one vertex with the highest score, we select the first vertex inserted into $cand$ among them. Finally, the score of the DAG S_r is the sum of $Score[u]$. We invoke Algorithm 2 with every vertex of the temporal query graph as the root once and then select the DAG with the highest score.

Example IV.2. Figure 4 shows some processes of building $\hat{q}$ in Figure 3a from q in Figure 2c through Algorithm 2 with u_1 as the root. After u_1 is selected from $cand$ and processed, there are two candidates u_2 and u_3 in $cand$. $Score[u_2]$ is 1 because ε_1 becomes a temporal ancestor of ε_3 if u_2 is selected as the next vertex. Similarly, $Score[u_3]$ is 2 because of two ordered pairs $\varepsilon_2 \prec \varepsilon_4$ and $\varepsilon_2 \prec \varepsilon_6$. Therefore, u_3 with a higher score is selected among u_2 and u_3, and the DAG is changed as shown in Figure 4a. Then, given $Score[u_2] = Score[u_4] = 1$ and $Score[u_5] = 0$, we choose u_2 because u_2 was inserted into $cand$ before u_4, so the DAG in Figure 4b is created. Next, we choose u_4 whose score is 2 and then choose u_5 whose score is 0. Finally, we return the DAG in Figure 3a with a score of 5 (=2+1+2).

Lemma IV.2. Given a query graph q, the time complexity of Algorithm 2 is $O(\sum_{u \in V(q)} \deg(u)^2 \times |E(q)|)$ where $\deg(u)$ represents the degree of vertex u.

Proof. Lines 1 and 2 can be done in $O(1)$ time and $O(|V(q)|)$



(a) After selecting u_3 (b) After selecting u_2 (c) After selecting u_4

Fig. 4: Process of building DAG from q in Figure 2c with u_1 as the root

time, respectively. The while loop is iterated $|V(q)|$ times, as each vertex in $V(q)$ can only be added to $cand$ once. Lines 5 and 13 can be executed in constant time. The process of pushing and popping vertex in $cand$ takes $O(\log|V(q)|)$ time for each operation (Lines 4 and 11). Lines 7-8, which add an edge to the DAG $\hat{q}_r$, take a total of $O(|E(q)|)$ time as it is performed for each edge. Therefore, the time complexity of the algorithm up to this point is $O(|V(q)|\log|V(q)|+|E(q)|)$.

The remaining part involves the computation of $Score[u']$ (Line 12). To compute $Score[u']$, we iterate over the neighbors u'_n of u' and count the number of temporal ancestors for (u', u'_n) in the DAG $\hat{q}_r$ if u'_n does not belong to $V(\hat{q}_r)$. Since counting the number of temporal ancestors for each edge requires $O(|E(q)|)$ time and there are at most $\deg(u')$ neighbors, Line 12 takes $O(\deg(u') \times |E(q)|)$ to execute once. Line 12 is executed when visiting an edge (u, u'), and since each edge is visited at most once, Line 12 is performed up to $\deg(u')$ times for the vertex u'. Thus, for vertex u', the total run time of Line 12 becomes $O(\deg(u')^2 \times |E(q)|)$. Since vertices are visited only once, Line 12 requires $O(\sum_{u\in V(q)} \deg(u)^2 \times |E(q)|)$ time in total. Consequently, the time complexity of Algorithm 2 is $O(|V(q)|\log|V(q)| + |E(q)| + \sum_{u\in V(q)} \deg(u)^2 \times |E(q)|)$, which simplifies to $O(\sum_{u\in V(q)} \deg(u)^2 \times |E(q)|)$. □

Algorithm 2 is called $|V(q)|$ times in Algorithm 1, so the total time complexity of building DAG is $O(\sum_{u\in V(q)} \deg(u)^2 \times |V(q)| \times |E(q)|)$.

C. Computing TC-Matchable Edge

In this subsection, we first propose a data structure *max-min timestamp*, which helps determine whether TC-weak embeddings exist or not and requires polynomial space. Also, we describe how to efficiently update the data structure when a new edge arrives or an existing edge expires in the data graph. There are two main points we considered when designing the data structure.

One is that when the data graph changes due to a newly arrived or expired edge $\overline{e}$, it does not only affect the TC-matchable edges for $\overline{e}$, but also affects other TC-matchable edges. For example, ε_2 of q in Figure 2c becomes a TC-matchable edge of σ_8 of G in Figure 2a regarding $\hat{q}$ after σ_{14} arrives. However, since it is expensive to recompute the filtering results for all edges, it is necessary to focus on the part affected by the changed edge $\overline{e}$.

The other point is that it is inefficient to search for TC-weak embeddings from scratch to determine whether a query edge is a TC-matchable edge of a data edge for each update. If we store information about partial TC-weak embeddings we found before, we can utilize it to determine if there is a TC-weak embedding. For instance, when determining whether ε_2 is a TC-matchable edge of σ_8 after the arrival of σ_{14}, we need to find the TC-weak embedding $M' = \{(\varepsilon_2, \sigma_8), (\varepsilon_4, \sigma_{13}), (\varepsilon_5, \sigma_{10}), (\varepsilon_6, \sigma_{14})\}$. A part of M', $\{(\varepsilon_4, \sigma_{13}), (\varepsilon_5, \sigma_{10})\}$, is created when σ_{13} arrives, so if we store the information about that part, we can use it to find M' when σ_{14} arrives.

Definition IV.2. Given a weak embedding M' of $\hat{q}_u$, the *min timestamp for* $e \in E(q)$ of M' is defined as the minimum timestamp among $M'(e')$ where e' is a temporal descendant of e in $\hat{q}$. If there is no such e', it is defined as ∞.

Definition IV.3. For each $u \in V(q)$, $v \in V(G)$, and $e \in E(q)$, the *max-min timestamp for* e *of* $\hat{q}_u$ *at* v, denoted by $\mathcal{T}(\hat{q})[u, v, e]$, is the largest min timestamp for e among weak embeddings of $\hat{q}_u$ at v. If there is no weak embedding of $\hat{q}_u$ at v, it is defined as $-\infty$. For each $e' \in E(q), \overline{e'} \in E(G)$, and $e \in E(q)$, similarly, $\mathcal{T}(\hat{q})[e', \overline{e'}, e]$ denotes the *max-min timestamp for* e *of* $\hat{q}_{e'}$ *at* $\overline{e'}$, which is the largest min timestamp for e among weak embeddings of $\hat{q}_{e'}$ at $\overline{e'}$. For brevity, we will use the simplified notation $\mathcal{T}$ instead of $\mathcal{T}(\hat{q})$ when the context is unambiguous.

Example IV.3. In a temporal data graph G in Figure 2a, a temporal query graph q in Figure 2c, and a query DAG $\hat{q}$ in Figure 3a, there exists four weak embeddings $\{(\varepsilon_4, \sigma_{13}), (\varepsilon_5, \sigma_9), (\varepsilon_6, \sigma_7)\}$, $\{(\varepsilon_4, \sigma_{13}), (\varepsilon_5, \sigma_9), (\varepsilon_6, \sigma_{14})\}$, $\{(\varepsilon_4, \sigma_{13}), (\varepsilon_5, \sigma_{10}), (\varepsilon_6, \sigma_7)\}$, and $\{(\varepsilon_4, \sigma_{13}), (\varepsilon_5, \sigma_{10}), (\varepsilon_6, \sigma_{14})\}$ of $\hat{q}_{u_3}$ at v_4. Since ε_4, ε_5, and ε_6 are temporal descendants of ε_2, the min timestamps for ε_2 of each weak embedding are 7, 9, 7, and 10, respectively. Thus, $\mathcal{T}[u_3, v_4, \varepsilon_2]$ stores the maximum value of 10 among them.

For a TC-weak embedding M' of $\hat{q}_e$ at $\overline{e}$, the min timestamp for e of any partial weak embedding of M' is greater than the timestamp of $\overline{e}$ by definition of TC-weak embedding. That is, if there is no partial weak embedding with a min timestamp for e greater than the timestamp of $\overline{e}$, there is no TC-weak embedding of $\hat{q}_e$ at $\overline{e}$. Therefore, we can use the max-min timestamps $\mathcal{T}$ to determine if $e \in E(q)$ is a TC-matchable edge of $\overline{e} \in E(G)$ regarding $\hat{q}$. In Example IV.3, since $\mathcal{T}[u_3, v_4, \varepsilon_2] = 10$, we can see that there is a weak embedding M'_1 of $\hat{q}_{u_3}$ at v_4 whose min timestamp for ε_2 is 10. To determine whether $\varepsilon_2 = (u_1, u_3)$ is a TC-matchable edge of $\sigma_8 = (v_1, v_4, 8)$ regarding $\hat{q}$, consider a weak embedding M'_2 of $\hat{q}_{\varepsilon_2}$ at σ_8 by adding $\{(\varepsilon_2, \sigma_8)\}$ to the previous weak embedding M'_1. Then, $T_G(M'_2(\varepsilon_2)) = T_G(\sigma_8) = 8 < 10 \leq T_G(M'_2(\varepsilon))$ holds for every ε where $\varepsilon_2 \rightsquigarrow_{\hat{q}} \varepsilon$ because $T_G(M'_2(\varepsilon))$ is greater than or equal to 10 which is the min timestamp for ε_2 of M'_1. Thus, M'_2 is a TC-weak embedding $\hat{q}_{\varepsilon_2}$ at σ_8 that we want to find and ε_2 is a TC-matchable edge of σ_8 regarding $\hat{q}$. On the other hand, $\varepsilon_2 = (u_1, u_3)$ is not a TC-matchable edge of $\sigma_{12} = (v_1, v_4, 12)$ regarding $\hat{q}$. This is because we can confirm that the min timestamp for ε_2 in all weak embeddings of $\hat{q}_{u_3}$ at v_4 is not greater than $T_G(\sigma_{12}) = 12$, which is derived from $\mathcal{T}[u_3, v_4, \varepsilon_2] = 10$. We can generalize this as follows.

Lemma IV.3. Given a temporal data graph G, a temporal query graph q, and a query DAG $\hat{q}$, $e = (u_1, u_2) \in E(q)$ is a TC-matchable edge of $\overline{e} = (v_1, v_2, t) \in E(G)$ regarding $\hat{q}$ if and only if the timestamp of $\overline{e}$ $(= t)$ is less than $\mathcal{T}[u_2, v_2, e]$.

Proof. First, we show the 'only if' part. Let e be a TC-matchable edge of $\overline{e}$, implying the existence of a TC-weak embedding M of $\hat{q}_e$ at $\overline{e}$. According to Definition IV.1, for every e' such that $e \rightsquigarrow_{\hat{q}} e'$, it holds that $T_G(M(e'))$ is greater than $T_G(M(e)) = T_G(\overline{e}) = t$. That is, for a weak embedding $M' = M - \{(e, \overline{e})\}$ of $\hat{q}_{u_2}$ at v_2, the min timestamp for e of M' $(= t')$ is greater than t. Since $\mathcal{T}[u_2, v_2, e]$ is the largest min timestamp for e among weak embeddings of $\hat{q}_{u_2}$ at v_2, it follows that $\mathcal{T}[u_2, v_2, e] \geq t' > t$.
Now, we demonstrate the 'if' part. According to Definition IV.3, there exists a weak embedding M' of $\hat{q}_{u_2}$ at v_2 with a min timestamp for e equal to $\mathcal{T}[u_2, v_2, e]$. By utilizing Definition IV.2 and the given assumption, $T_G(M'(e')) \geq \mathcal{T}[u_2, v_2, e] > t$ holds for every e' where $e \rightsquigarrow_{\hat{q}} e'$. Hence, the weak embedding $M = M' \cup \{(e, \overline{e})\}$ of $\hat{q}_e$ at $\overline{e}$ satisfies the conditions of a TC-weak embedding, as defined in Definition IV.1. Therefore, e is a TC-matchable edge of $\overline{e}$ and we proved the statement. □

According to Lemma IV.3, we can determine at constant time whether $e \in E(q)$ is a TC-matchable edge of $\overline{e} \in E(G)$ regarding $\hat{q}$, given an array $\mathcal{T}$. Therefore, when a change occurs in the data graph, it is sufficient to update $\mathcal{T}$ without searching TC-weak embeddings. In order to solve the two issues mentioned at the beginning of the subsection, we should recompute only the portion of $\mathcal{T}$ whose values may change, and also utilize the portion of $\mathcal{T}$ already computed when recomputing $\mathcal{T}[u, v, e]$.

We address the second issue by using the fact that a weak embedding of $\hat{q}_u$ consists of weak embeddings of $\hat{q}_{(u,u_c)}$ where u_c is a children of u. For each (u, u_c), consider a weak embedding with the largest min timestamp for e among weak embeddings of $\hat{q}_{(u,u_c)}$. Then the weak embedding of $\hat{q}_u$ composed of such weak embeddings also has the largest min timestamp for e. Furthermore, a weak embedding of $\hat{q}_{(u,u_c)}$ can be divided into the mapping of (u, u_c) and the weak embedding of $\hat{q}_{u_c}$. Since $\mathcal{T}[u_c, \cdot, e]$ stores the max-min timestamps for e among weak embeddings of $\hat{q}_{u_c}$, it can be utilized to compute $\mathcal{T}[u, v, e]$. Given $\mathcal{T}[u_c, \cdot, e]$ for every u_c, we can compute $\mathcal{T}[u, v, e]$ as follows.

First, assuming that (u, u_c) matches (v, v_c, t), $\mathcal{T}[(u, u_c), (v, v_c, t), e]$, which is the largest min timestamp for e among weak embeddings of $\hat{q}_{(u,u_c)}$ at (v, v_c, t), can be obtained from $\mathcal{T}[u_c, v_c, e]$ that we have. If (u, u_c) is not a temporal descendant of e, then $\mathcal{T}[(u, u_c), (v, v_c, t), e]$ is not related to (u, u_c). So, $\mathcal{T}[(u, u_c), (v, v_c, t), e]$ is the same as the max-min timestamp for e of $\hat{q}_{u_c}$ at v_c (i.e., $\mathcal{T}[u_c, v_c, e]$). If (u, u_c) is a temporal descendant of e, $\mathcal{T}[(u, u_c), (v, v_c, t), e]$ is the smaller of $\mathcal{T}[u_c, v_c, e]$ and t. The following equation summarizes this.

$$\mathcal{T}[(u, u_c), (v, v_c, t), e] = \begin{cases} \min(\mathcal{T}[u_c, v_c, e], t) & \text{if } e \rightsquigarrow_{\hat{q}} (u, u_c) \\ \mathcal{T}[u_c, v_c, e] & \text{otherwise} \end{cases}$$

Algorithm 3: TCMInsertion

Input: a data graph g, a query DAG $\hat{q}$, and a new edge $\overline{e} = (v_1, v_2, t)$
Output: a set of edge pairs E^+_{DCS}

```
1  E+_DCS ← ∅;
2  Q ← empty queue;
3  foreach e = (u1, u2) ∈ E(q̂) that matches to ē do
4      if t < T[u2, v2, e] then
5          E+_DCS ← E+_DCS ∪ {(e, ē)}
6      foreach ancestor e' of e do
7          Compute T[u1, v1, e'] by Eq. (1);
8          if T[u1, v1, e'] changes then
9              Q.push((u1, v1, e'))
10     while Q ≠ ∅ do
11         (u, v, e') ← Q.pop;
12         foreach u_p ∈ Parent(u) do
13             foreach (v_p, v, t') that (u_p, u) can match do
14                 if e' = (u_p, u) and t' < T[u, v, e'] and
                     ((u_p, u), (v_p, v, t')) ∉ DCS then
15                     E+_DCS ←
                         E+_DCS ∪ {((u_p, u), (v_p, v, t'))}
16                 if e' is an ancestor of (u_p, u) then
17                     Compute T[u_p, v_p, e'] by Eq. (1);
18                     if T[u_p, v_p, e'] changes then
19                         Q.push((u_p, v_p, e'))
20 return E+_DCS;
```

Now, consider a weak embedding of $\hat{q}_{(u,u_c)}$ for $\mathcal{T}[u, v, e]$. Since there may be several edges incident on v such that (u, u_c) can be matched, the largest min timestamp for e among weak embeddings of $\hat{q}_{(u,u_c)}$ is the maximum of the max-min timestamps when (u, u_c) is matched to each edge. Finally, if the largest min timestamp is computed for all children of u, $\mathcal{T}[u, v, e]$ becomes the smallest of them. Summarizing this process, the following recurrence can be obtained.

$$\mathcal{T}[u, v, e] = \min_{u_c \in \text{Child}(u)} \left(\max_{(v, v_c, t)} (\mathcal{T}[(u, u_c), (v, v_c, t), e]) \right) \quad (1)$$

Based on the above recurrence, we can compute $\mathcal{T}$ by dynamic programming in a bottom-up fashion in DAG $\hat{q}$.

What is now left is to recompute only the portions that may change, not the whole of $\mathcal{T}$, when $\mathcal{T}$ needs to be updated due to the arrival or expiration of an edge. We handle this by recomputing $\mathcal{T}[u, v, e]$ for only two cases: (i) when an edge (v, v_c, t) that matches (u, u_c) arrives or expires, or (ii) when $\mathcal{T}[u_c, v_c, e]$ changes in the process of updating $\mathcal{T}$.

Algorithm 3 shows the process of updating the portion of $\mathcal{T}$ through the above method and Equation (1) when a new edge $\overline{e}$ arrives. It performs the update process of cases (i) (Lines 4–9) and (ii) (Lines 10–19). This procedure also returns a set E^+_{DCS} consisting of pairs of edges $e' \in E(q)$ and $\overline{e'} \in E(g)$, where e' newly becomes a time-constrained matchable edge of $\overline{e'}$ in the process of updating $\mathcal{T}$. `TCMDeletion`, which updates $\mathcal{T}$ when an edge expires, is the same as `TCMInsertion` except that Line 14 checks if $((u_p, u), (v_p, v, t'))$ is in `DCS`.

Example IV.4. Consider when $\sigma_{14} = (v_4, v_7, 14)$ in Figure 2a arrives. Since $\varepsilon_6 = (u_3, u_5)$ in Figure 2c can match σ_{14}, we first check whether ε_6 is a TC-matchable edge of σ_{14} regarding $\hat{q}$. Indeed, ε_6 is a TC-matchable edge of σ_{14} regarding $\hat{q}$ because $T_G(\sigma_{14}) < \mathcal{T}[u_5, v_7, \varepsilon_6] = \infty$. So, we insert $(\varepsilon_6, \sigma_{14})$ into E^+_{DCS}. Next, we recompute $\mathcal{T}[u_3, v_4, \varepsilon_2]$. For u_4, which is a child of u_3, there is only $\sigma_{13} = (v_4, v_5, 13)$ which is incident on v_4 and can be matched to $\varepsilon_4 = (u_3, u_4)$. Thus, the largest min timestamp for ε_2 among weak embeddings of $\hat{q}_{\varepsilon_4}$ is equal to $\mathcal{T}[\varepsilon_4, \sigma_{13}, \varepsilon_2] = \min(\mathcal{T}[u_4, v_5, \varepsilon_2], 13) = \min(10, 13) = 10$. For the other child u_5 of u_3, there are two edges $\sigma_7 = (v_4, v_7, 7)$ and $\sigma_{14} = (v_4, v_7, 14)$ which are incident on v_4 and can be matched to $\varepsilon_6 = (u_3, u_5)$. The max-min timestamps for ε_2 of $\hat{q}_{\varepsilon_6}$ when ε_6 is matched to σ_7 and σ_{14} are $\mathcal{T}[\varepsilon_6, \sigma_7, \varepsilon_2] = \min(\infty, 7) = 7$ and $\mathcal{T}[\varepsilon_6, \sigma_{14}, \varepsilon_2] = \min(\infty, 14) = 14$, respectively. Therefore, the largest min timestamp for ε_2 among weak embeddings of $\hat{q}_{\varepsilon_6}$ is $\max(7, 14) = 14$. Finally, $\mathcal{T}[u_3, v_4, \varepsilon_2]$ becomes the minimum value of 10 among the timestamps 10 and 14 obtained from the children of u_3. As $\mathcal{T}[u_3, v_4, \varepsilon_2]$ is updated from 7 to 10, ε_2 becomes a TC-matchable edge of σ_8, so we insert $(\varepsilon_2, \sigma_8)$ into E^+_{DCS}. However, since the timestamp of σ_{12} is still greater than $\mathcal{T}[u_3, v_4, \varepsilon_2] = 10$, ε_2 does not become a TC-matchable edge of σ_{12} and we do not have to insert $(\varepsilon_2, \sigma_{12})$ into E^+_{DCS}.

Lemma IV.4. Given a temporal data graph G, a temporal query graph q, and a time window δ, let n_{max} and m_{max} be the maximum number of vertices and edges in the temporal data graph G within the time window δ, respectively. Then the space complexity of $\mathcal{T}$ is $O(|E(q)|^2 \times m_{max})$.

Proof. There are two types of array: $\mathcal{T}[u, v, e]$ and $\mathcal{T}[e', \overline{e'}, e]$. Regarding $\mathcal{T}[u, v, e]$, the variables u and e are bounded by $O(|V(q)|)$ and $O(|E(q)|)$, respectively. Furthermore, it is sufficient to store the max-min timestamps at v within the current time window, rather than storing timestamps for all vertices of the temporal data graph. Therefore, the space complexity of $\mathcal{T}[u, v, e]$ is determined as $O(|V(q)| \times |E(q)| \times n_{max})$. Similarly, the space complexity of $\mathcal{T}[e', \overline{e'}, e]$ can be expressed as $O(|E(q)|^2 \times m_{max})$. Consequently, the overall space complexity of $\mathcal{T}$ is given by $O(|V(q)| \times |E(q)| \times n_{max} + |E(q)|^2 \times m_{max}) = O(|E(q)|^2 \times m_{max})$. □

Lemma IV.5. Let P be the set of (u, v) where $\mathcal{T}[u, v, \cdot]$ changes. Then the time complexity of the `TCMInsertion` and `TCMDeletion` is $O(d_{max} \times \sum_{p \in P} \deg(p) + d_{max} \times |E(q)|^2)$, where $\deg(p)$ is the number of edges in `DCS` connected to p [23] and d_{max} is the maximum value among $\deg(p)$.

Proof. The function `TCMDeletion` is similar to the `TCMInsertion`, so we will only show the time complexity of `TCMInsertion` of Algorithm 3. First, Lines 4–5 are executed $O(|E(q)|)$ times and Lines 6–9 are executed $O(|E(q)|^2)$ times. Lines 4–5 and 8–9 take a constant time. Line 7 takes $O(d_{max})$ time to update $\mathcal{T}[u_1, v_1, e']$ since we need to check $\mathcal{T}[(u_1, u'), (v_1, v', \cdot), e']$ for every `DCS` edge $((u_1, u'), (v_1, v', \cdot))$ connected to (u_1, v_1). Therefore, the total execution time of Lines 3–9 is $O(d_{max} \times |E(q)|^2)$. Next, the while loop of Lines 11–19 except Line 17 takes a time proportional to the number of parents of (u, v) in `DCS`, which is equal to or less than $\deg((u, v))$. Line 17 is executed $O(\deg((u, v)))$ times, and it takes $O(d_{max})$ time for each execution. Since the while loop (Line 10) is executed for (u, v) where $\mathcal{T}[u, v, \cdot]$ changes, the total execution time of Lines 11-19 is $O(d_{max} \times \sum_{p \in P} \deg(p))$. Hence, the time complexity of `TCMInsertion` is $O(d_{max} \times \sum_{p \in P} \deg(p) + d_{max} \times |E(q)|^2)$, and `TCMDeletion` has the same time complexity. □

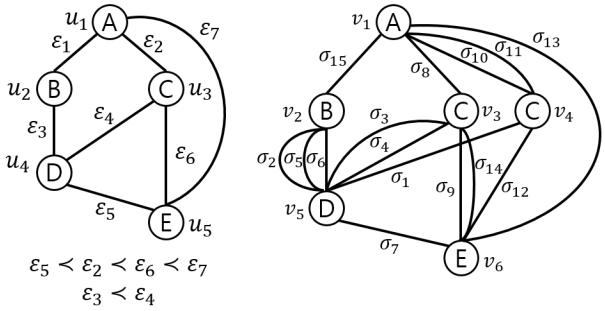


Fig. 5: A new running example of a temporal query graph q and a temporal data graph G

V. Time-Constrained Pruning in Backtracking

In this section, we present three time-constrained pruning techniques in backtracking. Our matching algorithm gradually extends a partial embedding until it finds a time-constrained embedding like the existing backtracking-based algorithms that solve the continuous subgraph matching problem [22], [23]. To extend a partial embedding M, existing backtracking-based algorithms select an unmapped vertex u of a query graph q and match u to each candidate of u. In time-constrained matching, mapping between edges (rather than mapping between vertices) is essential because each edge has temporal information, and thus we use mapping between edges. We find a set of candidate edges (v, v') for an edge (u, u') and then match (u, u') to each candidate edge (v, v') to extend a partial embedding M. In backtracking, we prune some candidates (v, v') of (u, u') if it is guaranteed that there is no time-constrained embedding when (u, u') matches (v, v') by considering parallel edges and the temporal order. Algorithm 4 shows this backtracking process.

Definition V.1. For each edge $e \in E(q)$ and a (partial) embedding M, a set $R^+_M(e)$ is the set of edges $e' \in E(q)$ that are temporally related to e (i.e., $e \prec e'$ or $e' \prec e$) and are members of M. Conversely, a set $R^-_M(e)$ is the set of edges $e' \in E(q)$ that are temporally related to e and are not members of M.

Definition V.2. A set $EC_M(e)$ of candidate edges of $e = (u_1, u_2) \in E(q)$ regarding a partial time-constrained embedding M is defined as the set of edges in $E(G)$ between $M(u_1)$ and $M(u_2)$ that are TC-matchable edges for e and satisfy the temporal relationships with $M(e')$ where $e' \in R^+_M(e)$.

Algorithm 4: FindMatches

Input: DCS, a data edge $\overline{e}$, and a partial time-constrained embedding M
Output: all occurred/expired time-constrained embeddings including $\overline{e}$

if $|M| = |V(q)| + |E(q)|$ **then**
  Report M as a match;
else if $|M| = 0$ **then**
  Let $\overline{e} = (v, v', t)$;
  foreach $e = (u, u') \in E(q)$ *such that* $((u, u'), (v, v', t)) \in$ *DCS* **do**
    $M \leftarrow \{(u, v), (u', v'), (e, \overline{e})\}$;
    FindMatches(DCS, $\overline{e}$, M);
else
  if $\exists$ *an unmapped query edge with both endpoints mapped* **then**
    $e \leftarrow$ an edge satisfying the above condition;
    Compute a set of candidates $EC_M(e)$ of e;
    foreach $\overline{e} \in EC_M(e)$ *which is not pruned* **do**
      $M' \leftarrow M \cup \{(e, \overline{e})\}$;
      FindMatches(DCS, $\overline{e}$, M');
  else
    $u \leftarrow$ next vertex according to the matching order in [23];
    Compute a set of candidates $C_M(u)$ of u as in [23];
    foreach $v \in C_M(u)$ **do**
      $M' \leftarrow M \cup \{(u, v)\}$;
      FindMatches(DCS, $\overline{e}$, M');

As a new running example, we use a temporal query graph q and a temporal data graph G in Figure 5.

Example V.1. Consider the partial time-constrained embedding $M_1 = \{(\varepsilon_1, \sigma_{15}), (\varepsilon_2, \sigma_8)\}$ and ε_3 as the next matching edge. Then $R^+_{M_1}(\varepsilon_3) = \emptyset$, $R^-_{M_1}(\varepsilon_3) = \{\varepsilon_4\}$, and $EC_{M_1}(\varepsilon_3) = \{\sigma_2, \sigma_5, \sigma_6\}$. If we extend M_1 to $M_2 = M_1 \cup \{(\varepsilon_3, \sigma_2)\}$ by matching ε_3 to σ_2, then $R^+_{M_2}(\varepsilon_4) = \{\varepsilon_3\}$, $R^-_{M_2}(\varepsilon_4) = \emptyset$, and $EC_{M_2}(\varepsilon_4) = \{\sigma_3, \sigma_4\}$ for the next matching edge ε_4. If ε_3 matches σ_5 instead of σ_2, then $EC_{M'_2}(\varepsilon_4) = \emptyset$ where $M'_2 = M_1 \cup \{(\varepsilon_3, \sigma_5)\}$ because $T_G(\sigma_5) \not\prec T_G(\sigma_3)$ and $T_G(\sigma_5) \not\succ T_G(\sigma_4)$ while $\varepsilon_3 \prec \varepsilon_4$.

When there are multiple candidate edges in $EC_M(e)$, we introduce a method to prune some candidate edges according to the three cases of $R^-_M(e)$ to reduce the search space.

First, if $R^-_M(e)$ is an empty set (i.e., no temporally related edges remain), the search trees are the same no matter which edge of $EC_M(e)$ matches e to extend M. Therefore, we visit the node $M \cup \{(e, \overline{e})\}$ for only one candidate edge $\overline{e} \in EC_M(e)$ rather than extending M over all candidate edges in $EC_M(e)$. When a time-constrained embedding is found in the subtree rooted at $M \cup \{(e, \overline{e})\}$, we can find other time-constrained embeddings without exploring other subtrees by matching e to another candidate edge in $EC_M(e)$.

Second, consider $R^-_M(e)$ where all edges in $R^-_M(e)$ have the same temporal relationship with e. If $e \prec e'$ for all $e' \in R^-_M(e)$, we match e to candidate edges in $EC_M(e)$ in chronological order. If no time-constrained embedding is found when we match e to $\overline{e}$, we can skip the candidate edges after $\overline{e}$ in $EC_M(e)$. Conversely, when $e \succ e'$ for all $e' \in R^-_M(e)$, we match e to candidate edges in $EC_M(e)$ in reverse chronological order.

Finally, when $R^-_M(e)$ is not in the previous two cases, we consider the case where we failed to find any time-constrained embedding in the search tree rooted at $M = \{\ldots, (e, \overline{e})\}$ whose last mapping is $(e, \overline{e})$, and prune the other candidate edges if possible. If $\overline{e}$ did not cause any failures related to the ordered pair e and $e' \in R^-_M(e)$, we can prune other candidate edges of e. For each search tree node M, we compute the set of query edges related to the failures in the search tree rooted at M and utilize it to prune other candidates.

Definition V.3. Let M be a search tree node whose last mapped query edge is e and there is no time-constrained embeddings in the subtree rooted at M. A *temporal failing set* $TF_M \subseteq E(q)$ of node M is defined as follows:

1. If the node M is a leaf node (i.e., $M = \{\ldots, (e, \emptyset)\}$), $TF_M = R^+_M(e)$.
2. Otherwise,
   2.1 If there exists a child node $M_i = M \cup \{(e_i, \overline{e_i})\}$ of M such that $e_i \notin TF_{M_i}$, we set $TF_M = TF_{M_i} \cup R^+_M(e)$.
   2.2 Otherwise, $TF_M = \bigcup_{i=1}^{k} TF_{M_i} \cup R^+_M(e)$, where $M_1, \ldots, M_k$ are the children of M.

For the search tree node M whose last mapping is $(e, \overline{e})$, we can prune the other candidate edges of e by testing whether e is in the temporal failing set TF_M of M. If $e \notin TF_M$, $\overline{e}$ did not cause any failures related to e, so the sibling nodes of M, i.e., $M - \{(e, \overline{e})\} \cup \{(e, \overline{e}')\}$ for every $\overline{e}' \in EC_M(e)$, can be pruned.

Example V.2. Figure 6 shows the search tree of q and G in Figure 5 when σ_{15} arrives. In this example, to focus on pruning techniques in backtracking, we do not consider the filtering in Section IV and also use an arbitrary matching order. Except for u_3, since there is only one vertex in G that can be matched to a query vertex, we omit the mappings of these vertices from the search tree. Furthermore, when representing a mapping M, we enumerate the pairs in M in the order in which they are added to M.

First, consider the partial time-constrained embedding $M_1 = \{(\varepsilon_1, \sigma_{15}), (\varepsilon_2, \sigma_8), (\varepsilon_3, \sigma_2)\}$ in Figure 6 and ε_4 is to be matched next. At this time, since $R^-_{M_1}(\varepsilon_4) = \emptyset$ and $EC_{M_1}(\varepsilon_4) = \{\sigma_3, \sigma_4\}$, the subtrees rooted at $M_1 \cup \{(\varepsilon_4, \sigma_3)\}$ and $M_1 \cup \{(\varepsilon_4, \sigma_4)\}$ will be the same. Thus, we visit only the subtree rooted at $M_1 \cup \{(\varepsilon_4, \sigma_3)\}$. When we find the time-constrained embedding $\{(\varepsilon_1, \sigma_{15}), (\varepsilon_2, \sigma_8), (\varepsilon_3, \sigma_2), (\varepsilon_4, \sigma_3), (\varepsilon_5, \sigma_7), (\varepsilon_6, \sigma_9), (\varepsilon_7, \sigma_{14})\}$ in that subtree, we replace σ_3 with σ_4 to find the other time-constrained embedding without visiting the other subtree rooted at $M_1 \cup \{(\varepsilon_4, \sigma_4)\}$.

Next, suppose that we came back to the node $M_2 = \{(\varepsilon_1, \sigma_{15}), (\varepsilon_2, \sigma_8)\}$ after the exploration of the subtree rooted at $M_1 = M_2 \cup \{(\varepsilon_3, \sigma_2)\}$. Since all edges ε in $R^-_{M_2}(\varepsilon_3) = \{\varepsilon_4\}$ satisfy $\varepsilon_3 \prec \varepsilon$, we match ε_3 to σ_5, which is the next edge of $EC_{M_2}(\varepsilon_3) = \{\sigma_2, \sigma_5, \sigma_6\}$ in chronological order. In the subtree rooted at $M_2 \cup \{(\varepsilon_3, \sigma_5)\}$, there is no time-constrained embedding because there is no edge in G where ε_4 can be matched (i.e., $EC_{M_2 \cup \{(\varepsilon_3, \sigma_5)\}}(\varepsilon_4) = \emptyset$). When we return to

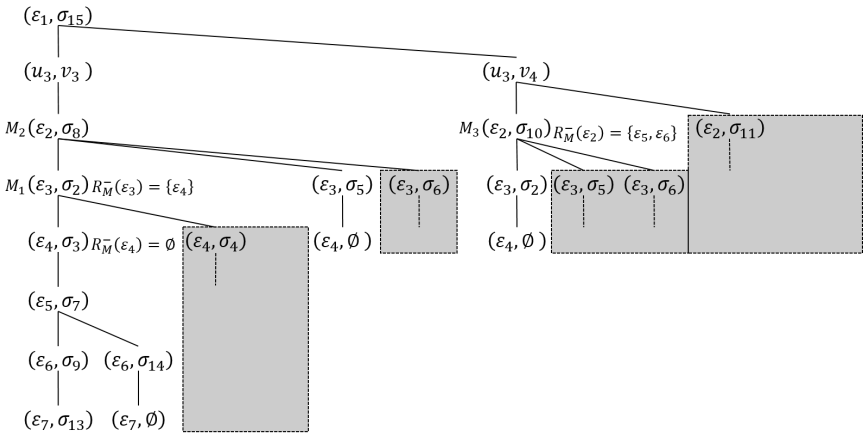


Fig. 6: Search tree when σ_{15} arrives. Nodes enclosed by dashed boxes are pruned

the node M_2, we know that time-constrained embeddings do not exist even in the subtree rooted at $M_2 \cup \{(\varepsilon_3, \sigma_6)\}$, so we prune that subtree.

Now, suppose that we explored the subtree rooted at $M_3 = \{(\varepsilon_1, \sigma_{15}), (\varepsilon_2, \sigma_{10})\}$ and came back to the node M_3. We failed to find time-constrained embeddings and the failure occurred in ε_4, which is temporally related to ε_3. The same failure will occur even if we match ε_2 to σ_{11} instead of σ_{10} because the failure is related only to ε_3 and not to ε_2. Therefore, we prune the subtree rooted at $M_3 - \{(\varepsilon_2, \sigma_{10})\} \cup \{(\varepsilon_2, \sigma_{11})\}$.

Lemma V.1. The space complexity of Algorithm 4 is $O(|E(q)| \times m_{max})$, where n_{max} and m_{max} are the same as defined in Lemma IV.4.

Proof. For each search tree node M, we map a vertex $u \in V(q)$ or an edge $e \in E(q)$ to its candidate vertex or candidate edge. If a vertex u is mapped in M, we need $O(n_{max})$ space to store the set of candidate vertices $C_M(u)$ [23] of u. Otherwise (i.e., an edge e is mapped in M), we need $O(m_{max})$ space to store the set of candidate edges $EC_M(e)$ of e. Additionally, we store the temporal failing set TF_M of M, which requires $O(|E(q)|)$ space. To sum up, each node requires $O(\max(n_{max}, m_{max}) + |E(q)|)$ space and the maximum depth of the search tree is $|V(q)| + |E(q)|$, resulting the space complexity $O((|E(q)| + m_{max}) \times |E(q)|)$ of Algorithm 4. Since we need to invoke `FindMatches` only when $m_{max} \geq |E(q)|$, we obtain the space complexity $O(|E(q)| \times m_{max})$. □

Function	Time complexity	Space complexity
`TCMUpdate`	$O(d_{max} \times \Sigma_{p \in P_1} \deg(p) + d_{max} \times \lvert E(q)\rvert^2)$	$O(\lvert E(q)\rvert^2 \times m_{max})$
`DCSUpdate`	$O(\Sigma_{p \in P_2} \deg(p) + \lvert E^+_{DCS}\rvert)$	$O(\lvert E(q)\rvert \times n_{max})$
`FindMatches`	exponential	$O(\lvert E(q)\rvert \times m_{max})$

TABLE II: Time and space complexities of each function

The time and space complexities of `DCSUpdate` are the same as those in [23]. Compared to `DCSUpdate`, `TCMUpdate` introduces an additional term d_{max} to update $\mathcal{T}$ (Lines 7 and 17 of Algorithm 3). While `TCMUpdate` and `DCSUpdate` achieve polynomial time complexities, the exponential time complexity of `FindMatches` is inevitable due to the NP-hardness of the problem. The space complexity of `TCMUpdate` is the size of the max-min timestamp $\mathcal{T}$, and that of `FindMatches` is the depth of the search tree (i.e., $|E(q)|$) times the number of candidate edges $EC_M(e)$ for an edge $e \in E(q)$ which is bounded by m_{max}.

Table II presents the time and space complexities of each function, where `Update` means both `Insertion` and `Deletion`.

VI. Performance Evaluation

In this section, we evaluate the performance of our algorithm against `Timing` [17] (there is no source code available for `Hasse` [24]). Since our algorithm uses the data structure `DCS` in `SymBi` [23], we implemented our algorithm by extending data structures (`DCS` and max-min timestamp) to incorporate temporal information and adding proposed techniques to `SymBi`. Furthermore, we modified the state-of-the-art continuous subgraph matching algorithms (`SymBi` [23] and `RapidFlow` [34]) to solve our problem by additionally checking whether the embeddings found by `SymBi` and `RapidFlow` satisfy the temporal order, and included them in our comparisons. Experiments are conducted on a CentOS machine with two Intel Xeon E5-2680 v3 2.50GHz CPUs and 256 GB memory. The source codes of `Timing`, `SymBi`, and `RapidFlow` were obtained from the authors and all methods are implemented in C++. Unlike the implementation of other comparative algorithms, `RapidFlow` assumes undirected edges in both data graph and query graph. We modified `RapidFlow` to output only the embeddings with correct directions.

Datasets. We use six datasets referred to as Netflow, Wiki-talk, Superuser, StackOverflow, Yahoo, and LSBench. Netflow is the anonymized network passive traffic data from CAIDA Internet Annoymized Traces 2015 Dataset [35]. Edges in Netflow are labeled with a triple of source ports, protocols, and destination ports. Wiki-talk is a temporal network representing

Wikipedia users editing each other's talk page from the Stanford SNAP library [36]. We label the vertices in Wiki-talk as the first character of the user's name as in [17]. Superuser and StackOverflow are networks of user-to-user interactions on the stack exchange websites [36]. The edges are labeled based on the type of interaction: (1) a user answering another user's question, (2) a user commenting on another user's question, and (3) a user commenting on another user's answer. Yahoo is a social network that captures the communication between users via Yahoo Messenger [37]. LSBench is synthetic social media stream data generated by the Linked Stream Benchmark data generator [38]. We set the number of users to 0.1 million and applied the default settings for other parameters. For Superuser, StackOverflow, and Yahoo, we randomly label the vertices among five distinct labels so that a reasonable number of queries can finish within the time limit. The characteristics of the datasets are summarized in Table III, including the number of vertices $|V|$, the number of edges $|E|$, the number of distinct vertex labels $|\Sigma_V|$, the number of distinct edge labels $|\Sigma_E|$, the average degree d_{avg}, and the average number of parallel edges between a pair of adjacent vertices m_{avg}.

TABLE III: Characteristics of datasets

Datasets	$\|V\|$	$\|E\|$	$\|\Sigma_V\|$	$\|\Sigma_E\|$	d_{avg}	m_{avg}
Netflow	0.37M	15.96M	1	346,672	85.4	27.6
Wiki-talk	1.14M	7.83M	365	1	13.7	2.37
Superuser	0.19M	1.44M	5	3	14.9	1.56
StackOverflow	2.60M	63.50M	5	3	48.8	1.75
Yahoo	0.10M	3.18M	5	1	63.6	3.51
LSBench	13.12M	21.04M	11	19	3.21	1.00

Queries. We adopt a query generation method in the previous study [17]. To generate query graphs, we first traverse the data graph by random walk. To ensure that there is a time-constrained embedding of the query graph in the data graph, the temporal order is determined as follows. We create a random permutation of edges in the query graph q generated by random walk, and then for any two edges $e, e' \in E(q)$, we set $e \prec e'$ if (1) e precedes e' in the permutation and (2) the timestamp of e is less than that of e'.

For each dataset, we set six different query sizes: 5, 7, 9, 11, 13, 15. The size of a query is defined as the number of edges. We generate 100 queries as mentioned above for each dataset and query size. For each query graph, we create 5 different temporal orders. The density of a temporal order is defined as the number of edge pairs with temporal relationships divided by the number of edge pairs in the query graph. To see the difference in performance according to the density of the temporal order, we generate two temporal orders with densities of 0 (empty) and 1 (total order), and the others to have densities close to 0.25, 0.50, and 0.75.

Performance Measurement We measure the elapsed time of time-constrained continuous subgraph matching for a dataset and a query graph. Since this problem is NP-hard, some queries may not finish in a reasonable time. Therefore, we set a time limit of 1 hour for each query. If an algorithm does not finish a query within the time limit, we regard the elapsed time of the query as 1 hour. We say that a query graph is *solved* if it ends within the time limit. Each query set consists of 100 query graphs with a same size. For each query set, we measure the average elapsed time and the number of solved query graphs. For a reasonable comparison, we compute the average time excluding query graphs that all algorithms failed to solve. In addition, we measure the peak memory usage using the "ps" utility to compare the memory usage of programs.

A. Experimental Results

We evaluate the performance of the algorithms by varying the query size, the density, and the time window size. Because each dataset has a different time span, we set each unit of the window size as the average time span between two consecutive edges in the dataset. We used 5 different window sizes in our experiments: 10k, 20k, 30k, 40k, 50k. Table IV shows the parameters used in the experiments. Values in boldface are used as default parameters.

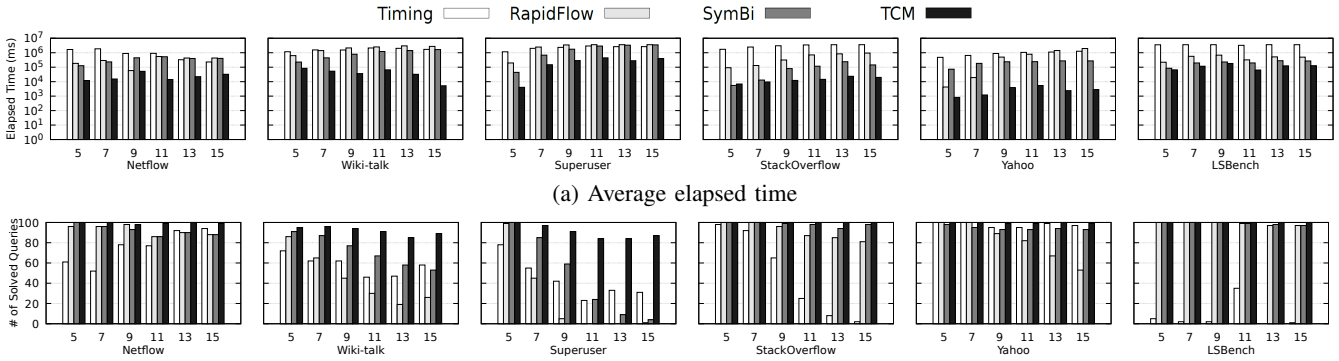
TABLE IV: Experiment settings

Parameter	Value Used
Datasets	Netflow, Wiki-talk, Superuser, StackOverflow, Yahoo, LSBench
Query size	5, 7, **9**, 11, 13, 15
Density	0, 0.25 **0.50**, 0.75, 1
Window size	10k, 20k, **30k**, 40k, 50k

Varying the query size. Figure 7 shows the performance results for varying the query size. We set the density to 0.50 and the window size to 30k.

Our algorithm `TCM` outperforms the other algorithms in terms of both query processing time and the number of solved queries for all datasets and query sizes. Specifically, `TCM` is more than two orders of magnitude faster than `Timing` in terms of query processing time for all query sizes in both StackOverflow and Yahoo. Moreover, `TCM` is usually more than 10 times faster than `RapidFlow` and `SymBi` on most datasets and query sizes, and achieves more than 100 times speed up on large query sizes in Yahoo. With respect to the number of solved queries, in LSBench, `Timing` rarely solves queries except when the query size is 11, while `TCM` successfully solves all queries in all query sizes. Furthermore, `TCM` continues to solve a significant number of queries as the query size increases, whereas the number of queries solved by the other three algorithms drops sharply in most datasets. Each of the three comparing algorithms solves a large number of queries when the query size is small, but it solves less than 10 queries when the query size increases for some datasets. These results suggest that `TCM` is a robust algorithm as it performs well across different datasets, while the other algorithms' performance varies significantly depending on the dataset.

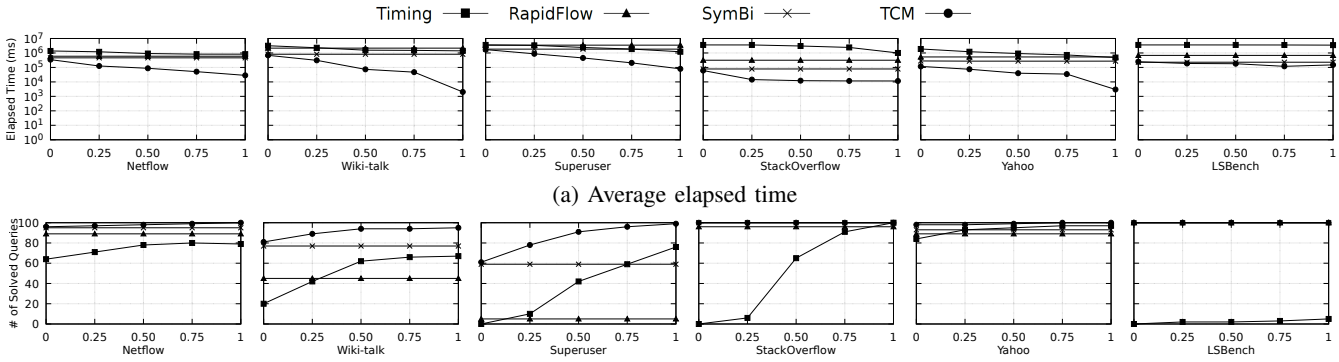
In Wiki-talk, the performance gap between `TCM` and the other algorithms widens as the query size increases. As the query size increases, the query processing time of `TCM` decreases, while the query processing time of `Timing`, `RapidFlow`, and `SymBi` generally increases. This is because as the query size increases, the time constraints associated



(a) Average elapsed time

(b) Solved queries

Fig. 7: Query processing time and the number of solved queries for varying query size



(a) Average elapsed time

(b) Solved queries

Fig. 8: Query processing time and the number of solved queries for varying density

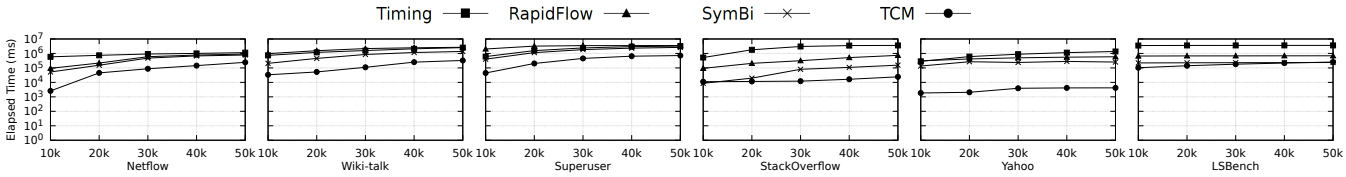
with one edge in the query graph increase, leading to better filtering and pruning in TCM. The following experiments with varying the density show that Wiki-talk is more affected by time constraints than other datasets.

Varying the density. Next, we fix the query size to 9 and the window size to 30k, and vary the density from 0 to 1 as shown in Figure 8. To see the effect of the density, we exclude queries that all algorithms failed to solve in every density when computing the average query processing time.

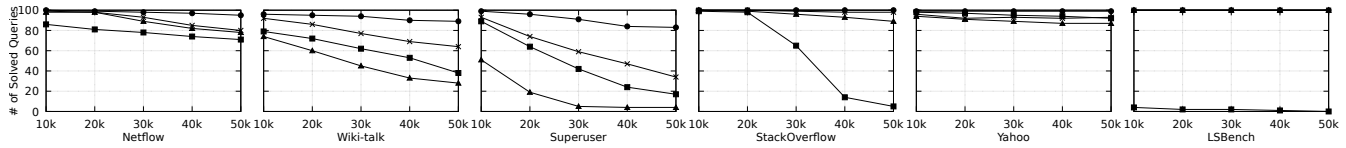
Both RapidFlow and SymBi find embeddings without considering a temporal order to solve the problem, so the query processing time does not change even if the density changes. On the other hand, Timing and TCM generally decrease as the density increases. Figure 8a shows that the query processing time of TCM is noticeably reduced as the density increases. In contrast, the query processing time of Timing does not decrease significantly like TCM. As a result, TCM outperforms Timing in all six datasets, with the performance gap increasing as the density increases. Specifically, when the density is 1, TCM is several hundred times faster than Timing in Wiki-talk and several times faster in the other datasets. Moreover, even when the density is 0, TCM still achieves a notable performance improvement compared to Timing. These results show that TCM utilizes the temporal order better than Timing to solve the time-constrained continuous subgraph matching problem. Also, this supports that query processing time is more affected by time constraints in Wiki-talk than in other datasets.

Varying the window size. In this experiment, we examine the effect of the window size on the performance by varying it from 10k to 50k, while using a query size of 9 and density of 0.50. Figure 9 represents the performance results. We can see that the query processing time increases and the number of solved queries decreases as the window size becomes larger. This is because as the window size increases we have to maintain more data edges and more time-constrained embeddings occur. In elapsed time, TCM maintains its superior performance in most cases, compared to other algorithms. As shown in Figure 9b, as the window size increases, the number of queries solved by Timing, RapidFlow, and SymBi decreases sharply, but TCM does not.

Memory usage. Figure 10 describes the peak memory within the time limit for varying the query size, which is averaged over 100 queries. We measure the peak memory as the maximum virtual set size in the "ps" utility output. In datasets other than Yahoo and LSBench, TCM uses 10 times less memory than Timing. In particular, there is a difference of more than 100 times in several query sizes of Netflow. This difference in memory usage comes from TCM using a data structure with polynomial space, while Timing stores all partial matches and thus requires exponential space. Moreover, as the query size increases, the gap in memory usage between TCM and Timing tends to widen.



(a) Average elapsed time



(b) Solved queries

Fig. 9: Query processing time and the number of solved queries for varying window size

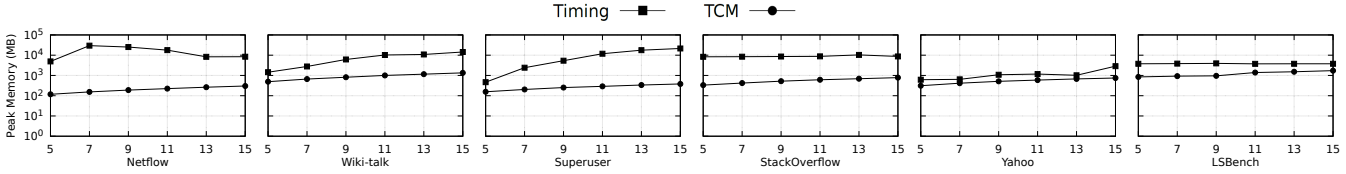
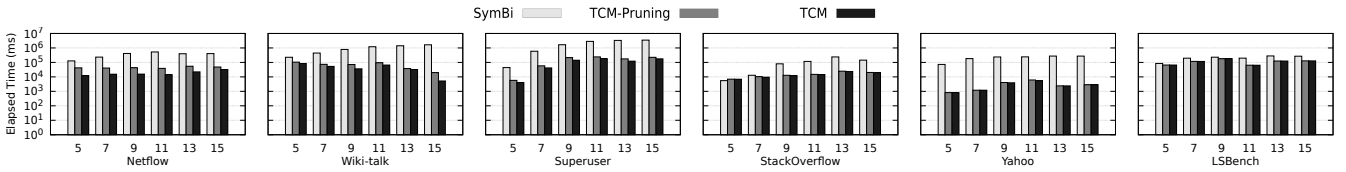
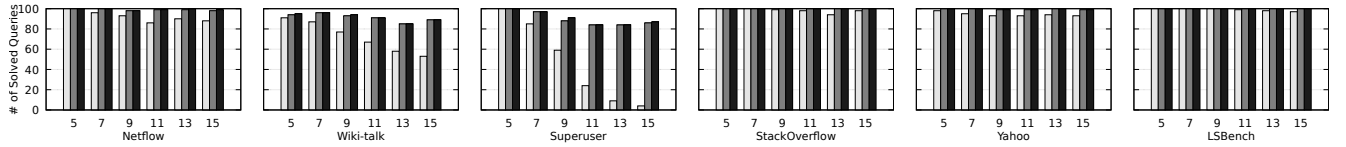


Fig. 10: Average peak memory for varying query size



(a) Average elapsed time



(b) Solved queries

Fig. 11: Evaluating techniques for varying query size

B. Effectiveness of Our Techniques

We evaluate the effectiveness of our techniques in this subsection. To measure the performance gains obtained by each technique, we implement a variant of our algorithm and compare it with `SymBi` and `TCM`.

- `SymBi`: a baseline for comparison.
- `TCM-Pruning`: using TC-matchable edges and not using time-constrained pruning techniques.
- `TCM`: using all techniques.

In this evaluation, we vary the query size, and fix the density to 0.50 and the window size to 30k. Figure 11 shows the results.

Effectiveness of the TC-matchable edge. To see the difference in the use of the TC-matchable edge technique, we compare the two algorithms `SymBi` and `TCM-Pruning`. Figure 11a shows that `TCM-pruning` outperforms `SymBi` in terms of query processing time. Specifically, `TCM-Pruning` is more than 10 times faster than `SymBi` across multiple datasets. Especially, `TCM-Pruning` outperforms `SymBi` by 152.75 times in Yahoo when the query size is 7. In terms of the number of solved queries, `TCM-Pruning` solves much more queries than `SymBi` in all datasets within the time limit.

To evaluate the effectiveness of the TC-matchable edge, we also compare the filtering power with and without the TC-matchable edge. We measure the filtering power with two factors. One is the number of edges in `DCS`, i.e., the number of edge pairs (pairs of edges in the query graph and edges in the data graph) that pass filtering when the TC-matchable edge is used, and the number of edge pairs with the same label when the TC-matchable edge is not used. The other is the number of vertices remaining in `DCS` after the filtering in [23]. Table V shows the average values obtained by dividing the value measured when the TC-matchable edge is used by the value measured when not using the TC-matchable edge. A smaller value in the table means more filtering when using the TC-matchable edge.

Effectiveness of time-constrained pruning techniques. When comparing the query processing time of `TCM-Pruning` and `TCM`, our time-constrained pruning techniques showed significant improvements in performance across various datasets (Figure 11a). Specifically, the average improvement in query processing time is 2.60 times in Netflow, 1.83 times in Wiki-talk, 1.40 times in Superuser,

TABLE V: Filtering power with and without the TC-matchable edge. Top: the ratio of the number of edges in DCS, bottom: the ratio of the number of vertices remaining in DCS after the filtering in [23].

	5	7	9	11	13	15	avg
Netflow	0.286	0.234	0.255	0.265	0.260	0.227	0.254
Wiki-talk	0.286	0.204	0.172	0.176	0.151	0.127	0.186
Superuser	0.337	0.270	0.223	0.247	0.185	0.149	0.235
StackOverflow	0.233	0.173	0.132	0.124	0.102	0.068	0.138
Yahoo	0.245	0.172	0.117	0.112	0.072	0.049	0.128
LSBench	0.608	0.609	0.557	0.503	0.443	0.455	0.529
Netflow	0.705	0.684	0.501	0.393	0.262	0.139	0.447
Wiki-talk	0.499	0.306	0.221	0.208	0.199	0.131	0.261
Superuser	0.619	0.497	0.427	0.401	0.313	0.234	0.415
StackOverflow	0.311	0.204	0.159	0.100	0.067	0.077	0.153
Yahoo	0.338	0.198	0.111	0.082	0.036	0.032	0.133
LSBench	0.415	0.423	0.319	0.233	0.255	0.209	0.309

1.04 times in StackOverflow, 1.03 times in Yahoo, and 1.00 times in LSBench. Figure 11b shows that TCM using time-constrained pruning techniques solves the same or more queries within the time limit than TCM−Pruning without the techniques.

VII. Conclusion

In this paper, we proposed two key techniques for time-constrained continuous subgraph matching to address the existing limitations: (1) TC-matchable edge for filtering and (2) a set of pruning techniques in backtracking. The former allows us to utilize temporal relationships between non-incident edges in filtering, and the latter allows us to prune some of the edges in the backtracking. We also suggest a data structure called max-min timestamp for the TC-matchable edge with polynomial space and an efficient method to update the data structure. Extensive experiments on real and synthetic datasets show that our approach outperforms the state-of-the-art algorithm in terms of query processing time and the number of queries solved within the time limit. Parallelizing our approach is an interesting future work.

Acknowledgements

S. Min, J. Jang, and K. Park were supported by Institute of Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2018-0-00551, Framework of Practical Algorithms for NP-hard Graph Problems). Giuseppe F. Italiano was partially supported by MUR. the Italian Ministry of University and Research, under PRIN Project n. 2022TS4Y3N - EXPAND: scalable algorithms for EXPloratory Analyses of heterogeneous and dynamic Networked Data. W.-S. Han was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2021R1A2B5B03001551).

References

[1] D. Leprovost, L. Abrouk, N. Cullot, and D. Gross-Amblard, "Temporal semantic centrality for the analysis of communication networks," in *Proceedings of the International Conference on Web Engineering*. Springer, 2012, pp. 177–184.

[2] W. Fan, "Graph pattern matching revised for social network analysis," in *Proceedings of the International Conference on Database Theory*, 2012, pp. 8–21.

[3] V. Bhat, A. Gokhale, R. Jadhav, J. Pudipeddi, and L. Akoglu, "Effects of tag usage on question response time," *Social Network Analysis and Mining*, vol. 5, no. 1, pp. 1–13, 2015.

[4] C. Joslyn, S. Choudhury, D. Haglin, B. Howe, B. Nickless, and B. Olsen, "Massive scale cyber traffic analysis: a driver for graph database research," in *First International Workshop on Graph Data Management Experiences and Systems*, 2013, pp. 1–6.

[5] B. Haslhofer, R. Karl, and E. Filtz, "O bitcoin where art thou? insight into large-scale transaction graphs." in *Joint Proceedings of the Posters and Demos Track of the International Conference on Semantic Systems and the International Workshop on Semantic Change & Evolving Semantics*, 2016.

[6] X. Sun, Y. Tan, Q. Wu, B. Chen, and C. Shen, "Tm-miner: Tfs-based algorithm for mining temporal motifs in large temporal network," *IEEE Access*, vol. 7, pp. 49 778–49 789, 2019.

[7] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: a graph engine for temporal graph analysis," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 1–14.

[8] C. Song, T. Ge, C. Chen, and J. Wang, "Event pattern matching over graph streams," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 413–424, 2014.

[9] Y. Ma, Y. Yuan, M. Liu, G. Wang, and Y. Wang, "Graph simulation on large scale temporal graphs," *GeoInformatica*, vol. 24, no. 1, pp. 199–220, 2020.

[10] L. Kovanen, M. Karsai, K. Kaski, J. Kertész, and J. Saramäki, "Temporal motifs in time-dependent networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2011, no. 11, pp. 119–133, 2011.

[11] P. Liu, A. R. Benson, and M. Charikar, "Sampling methods for counting temporal motifs," in *Proceedings of the ACM International Conference on Web Search and Data Mining*, 2019, pp. 294–302.

[12] U. Redmond and P. Cunningham, "Temporal subgraph isomorphism," in *Proceedings of IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. IEEE, 2013, pp. 1451–1452.

[13] U. Redmond and P. Cunningham, "Subgraph isomorphism in temporal networks," *arXiv preprint arXiv:1605.02174*, 2016.

[14] F. Li and Z. Zou, "Subgraph matching on temporal graphs," *Information Sciences*, vol. 578, pp. 539–558, 2021.

[15] J. Crawford and T. Milenković, "Cluenet: Clustering a temporal network based on topological similarity rather than denseness," *PloS one*, vol. 13, no. 5, p. e0195993, 2018.

[16] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, "Practice of streaming and dynamic graphs: Concepts, models, systems, and parallelism," *arXiv preprint arXiv:1912.12740*, 2020.

[17] Y. Li, L. Zou, M. T. Özsu, and D. Zhao, "Time constrained continuous subgraph search over streaming graphs," in *Proceedings of the International Conference on Data Engineering*. IEEE, 2019, pp. 1082–1093.

[18] Verizon. (2020) Data Breach Investigations Report. [Online]. Available: https://enterprise.verizon.com/resources/reports/2020-data-breach-investigations-report.pdf

[19] W. Fan, X. Wang, and Y. Wu, "Incremental graph pattern matching," *ACM Transactions on Database Systems*, vol. 38, no. 3, pp. 1–47, 2013.

[20] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu, "Graphflow: An active graph database," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2017, pp. 1695–1698.

[21] S. Choudhury, L. Holder, G. Chin, K. Agarwal, and J. Feo, "A selectivity based approach to continuous pattern detection in streaming graphs," in *Proceedings of the International Conference on Extending Database Technology*, 2015, pp. 157–168.

[22] K. Kim, I. Seo, W.-S. Han, J.-H. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong, "Turboflux: A fast continuous subgraph matching system for streaming graph data," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2018, pp. 411–426.

[23] S. Min, S. G. Park, K. Park, D. Giammarresi, G. F. Italiano, and W. Han, "Symmetric continuous subgraph matching with bidirectional dynamic programming," *Proceedings of the VLDB Endowment*, vol. 14, no. 8, pp. 1298–1310, 2021.

[24] X. Sun, Y. Tan, Q. Wu, and J. Wang, "Hasse diagram based algorithm for continuous temporal subgraph query in graph stream," in *Proceedings of the International Conference on Computer Science and Network Technology*. IEEE, 2017, pp. 241–246.

[25] E. E. Papalexakis, L. Akoglu, and D. Ience, "Do more views of a graph help? community detection and clustering in multi-graphs," in *Proceedings of the 16th International Conference on Information Fusion*. IEEE, 2013, pp. 899–905.

[26] P. Wang, Y. Qi, Y. Sun, X. Zhang, J. Tao, and X. Guan, "Approximately counting triangles in large graph streams including edge duplicates with a fixed memory usage," *Proceedings of the VLDB Endowment*, vol. 11, no. 2, pp. 162–175, 2017.

[27] T. Shafie, "A multigraph approach to social network analysis," *journal of Social Structure*, vol. 16, no. 1, pp. 1–21, 2015.

[28] A. Paranjape, A. R. Benson, and J. Leskovec, "Motifs in temporal networks," in *Proceedings of the ACM International Conference on Web Search and Data Mining*, 2017, pp. 601–610.

[29] M. Han, H. Kim, G. Gu, K. Park, and W.-S. Han, "Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2019, pp. 1429–1446.

[30] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM*, vol. 23, no. 1, pp. 31–42, 1976.

[31] W.-S. Han, J. Lee, and J.-H. Lee, "Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2013, pp. 337–348.

[32] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2016, pp. 1199–1214.

[33] H. Kim, Y. Choi, K. Park, X. Lin, S.-H. Hong, and W.-S. Han, "Versatile equivalences: Speeding up subgraph query processing and subgraph matching," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2021, pp. 925–937.

[34] S. Sun, X. Sun, B. He, and Q. Luo, "Rapidflow: an efficient approach to continuous subgraph matching," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 2415–2427, 2022.

[35] "Anonymized Internet Traces 2015," https://catalog.caida.org/dataset/passive_2015_pcap.

[36] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, 2014.

[37] R. Rossi and N. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, no. 1, 2015.

[38] D. Le-Phuoc, M. Dao-Tran, M.-D. Pham, P. Boncz, T. Eiter, and M. Fink, "Linked stream data processing engines: Facts and figures," in *International Semantic Web Conference*. Springer, 2012, pp. 300–312.