On Temporal-Constraint Subgraph Matching

Xiaoyu Leng†, Guang Zeng‡, Hongchao Qin†, Longlong Lin#, Rong-Hua Li†

†*Beijing Institute of Technology, China;* ‡*Ant Group;* #*Southwest University, China*

xiaoyuleng@bit.edu.cn; zengguang_77@qq.com; hcqin@bit.edu.cn; longlonglin@swu.edu.cn; rhli@bit.edu.cn

***Abstract*—Temporal-constraint subgraph matching has emerged as a significant challenge in the study of temporal graphs, which model dynamic relationships across various domains, such as social networks and transaction networks. However, the problem of temporal-constraint subgraph matching is NP-hard. Furthermore, because each *temporal-constraint* contains a permutation of times, existing subgraph matching acceleration techniques are difficult to apply to graphs with temporal-constraints. For instance, the baseline RI-DS algorithm takes 22 hours to match a query q under a temporal-constraint $\mathbb{TC}$ on an 8-million-scale dataset WT. This paper addresses the challenge of identifying subgraphs that not only structurally align with a given query graph but also satisfy specific temporal-constraints on the edges. We introduce three novel algorithms to tackle this issue: the TCSM-V2V algorithm, which uses a vertex-to-vertex expansion strategy and effectively prunes non-matching vertices by integrating both query and temporal-constraints into a temporal-constraint query graph; the TCSM-E2E algorithm, which employs an edge-to-edge expansion strategy, significantly reducing matching time by minimizing vertex permutation processes; and the TCSM-EVE algorithm, which combines edge-vertex-edge expansion to eliminate duplicate matches by avoiding both vertex and edge permutations. The best TCSM-EVE algorithm matches query q under $\mathbb{TC}$ on WT in just 84 seconds, achieving a three-order-of-magnitude speedup. Extensive experiments conducted across 7 datasets demonstrate that our approach outperforms existing methods in terms of both accuracy and computational efficiency.**

I. Introduction

Subgraph matching is a fundamental problem in graph theory. Given a data graph d and a query graph q, the goal is to identify all subgraphs of d that are isomorphic to q. While most recent studies focus on subgraph matching in static graphs [1]–[4], real-world graphs are often enriched with temporal information. The following scenarios highlight the importance of temporal subgraph matching in practical applications.

- In the financial sector, account transfers form a temporal graph with accounts as vertices and transactions as time-stamped edges. Money laundering often involves complex, time-spread transaction patterns [5]–[7]. Detecting these requires identifying suspicious subgraphs. Temporal subgraph isomorphism can uncover risky transactions, hidden connections, and money laundering activities, enhancing financial system security and integrity.
- In telecommunication networks, call and messaging logs generate temporal graphs, with users as vertices and communications as timestamped edges. These interactions can uncover patterns like frequent bursts or specific sequences of messages. Identifying such temporal subgraphs is crucial for detecting fraudulent activities, such as scam operations or coordinated attacks [8]–[10]. Analyzing these patterns enables telecommunication companies to improve fraud detection and enhance network security.

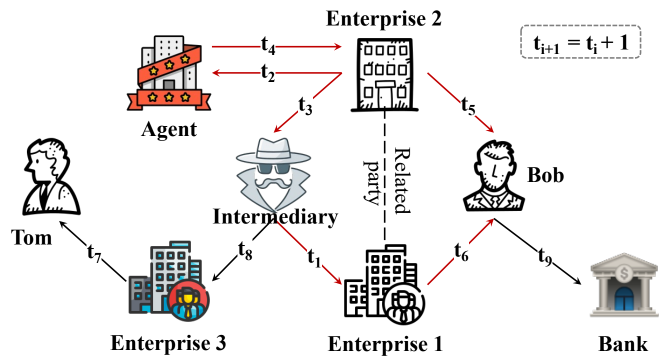


Fig. 1: A temporal bill circulation network in Ant Group.

A specific example is illustrated in Figure 1, depicting a financial bill circulation network frequently observed in Ant Group's data. In this network, each vertex represents an entity such as an enterprise, bank, intermediary, and individual, while each edge signifies a transaction behavior between these entities. Each edge is associated with a set of timestamps indicating when these interactions occurred, with t_i $(i \in N)$ following an arithmetic sequence characterized by a common difference of one day. A significant risk within the financial bill circulation network arises from the activities of bill intermediaries. These intermediaries purchase acceptance bills from companies at a discounted price using cash, subsequently transferring the bills to other enterprises or banks to earn an interest margin. For instance, the subgraph model red highlighted in Figure 1 represents a typical risk intermediary model [5]–[7], [11]. The suspected intermediary is depicted as purchasing acceptance bills with cash and rapidly transferring them. **Unlike traditional subgraph patterns, the *temporal-constraints* within these transactions are temporally linked, which means these transactions must occur within a designated time window of Δt.** Thus, *temporal-constraint* subgraph matching enhances the accuracy of detecting risky transactions within the financial bill circulation network.

Given the importance of matching subgraphs with *temporal constraints* in various applications, our work focuses on *temporal-constraint subgraph matching* ($TCSM$). However, subgraph matching is an NP-hard problem, and consequently, the *TCSM* problem is also NP-hard (as demonstrated in Section 2). To improve the efficiency of subgraph matching, researchers have employed various techniques, including candidate filtering, ordering methods, and enumeration methods. Despite the NP-hard nature of the problem, these advanced techniques can match subgraphs within graphs containing millions of vertices in milliseconds [1], [2], [12]. However, they struggle to accelerate the *TCSM* problem. As a result, **non-optimized techniques may require up to 22 hours to match a temporal-constraint subgraph within a large temporal graph** (as demonstrated in Section 5, where the baseline *RI-DS* algorithm requires 80,753 seconds to match query q_1 and temporal-constraint tc_2 on the WT dataset). This inefficiency arises because the *TCSM* process involves time-consuming vertex and edge ordering after each Depth-First Search (*DFS*) match to verify temporal-constraints.

To tackle this challenge, we designed efficient algorithms for the *TCSM* problem and conducted extensive experiments to validate their effectiveness. The contributions are as follows:

- We introduce the TCSM-V2V algorithm, which performs vertex expansion in a vertex-to-vertex manner. This algorithm merges the query and temporal-constraints graphs into a *Temporal-Constraint Query Graph* ($\mathcal{TCQ}$), which leverages temporal-constraints to prune final vertices and reduces unnecessary duplicate matches.
- We present the TCSM-E2E algorithm, which utilizes an edge-to-edge expansion approach. This algorithm integrates the query graph and *temporal constraints* graph into the $\mathcal{TCQ}+$ graph. By minimizing the vertex permutation process inherent in the TCSM-V2V algorithm, this method significantly reduces the matching time.
- We propose the TCSM-EVE algorithm, which employs an interactive edge-vertex-edge expansion strategy. This approach produces results **without the need for vertex and edge permutations**, minimizing duplicate matches.
- Experiments on 7 real-world temporal datasets demonstrate that our TCSM-EVE algorithm consistently outperforms other algorithms in nearly all scenarios. For instance, while the baseline RI-DS algorithm takes 22 hours to match query q_1 with temporal constraint tc_2 on the WT dataset, TCSM-EVE **completes the task in just 84 seconds**.
- The code is available at https://github.com/xiaoyu-ll/TSI.

II. PRELIMINARIES

In this section, we first introduce several fundamental concepts. A simple directed graph can be represented by $G = (V, E, \mathcal{L})$, where V is a set of vertices, E is a set of edges where each edge is a pair (u, v) and $u, v \in V$, and $\mathcal{L}$ is a label function that maps the vertex $u \in V$ to a label $\mathcal{L}(u)$. Note that, we only consider graphs with labeled vertices. However, if edges are also labeled, the algorithm can be easily generalized. Given a query graph $G_q = (V_q, E_q, \mathcal{L}_q)$ and a data graph $G = (V, E, \mathcal{L})$, a subgraph matching (isomorphism) is an injective function $f : V_q \to V$ that satisfies: $\forall u \in V_q$, $\mathcal{L}_q(u) = \mathcal{L}(f(u))$; and $\forall(u,v) \in E_q$, $(f(u), f(v)) \in E$.

In this paper, we focus on the problem of matching the subgraphs with *temporal constraints* in the temporal graph. We define the data temporal graph, the query graph, and the *temporal-constraint* graph as follows.

Definition 1 (Data Temporal Graph ($\mathbb{G}$)): A simple directed data temporal graph can be represented by $\mathbb{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}, \mathcal{T})$, where $\mathcal{V}$ is the set of vertices; $\mathcal{E}$ is a set of *temporal edges* where each *temporal edge* is (u, v, t), $u, v \in V$ and $t \in \mathcal{T}$ denotes the interaction time between u and v; $\mathcal{L}$ is a label function that maps the vertex $u \in V$ to a label $\mathcal{L}(u)$; $\mathcal{T}$ is a set of all the timestamps.

By Definition 1, we use $\mathcal{T}(u, v)$ to stand for the set of timestamps in which u and v and interacted, and $e.t$ to represent the interaction time of the edge $e \in \mathcal{E}$ in the temporal graph. For a temporal graph $\mathbb{G}$, the *de-temporal graph* of $\mathcal{G}$ denoted by $G = (V, E)$ is a graph that ignores all the timestamps associated with the temporal edges. More formally, for the de-temporal graph G of $\mathcal{G}$, we have $V = \mathcal{V}$ and $E = \{(u, v)|(u, v, t) \in \mathcal{E}\}$.

Definition 2 (Query Graph (G_q)): A query graph $G_q = (V_q, E_q, \mathcal{L}_q)$ is a labeled simple directed graph.

Assuming that set E_q in G_q adheres to a specific order $\{e_1, e_2 ... e_{|\mathcal{E}|}\}$, we define the *temporal-constraint* graph that adheres to the temporal constraints as follows.

Definition 3 (Temporal-Constraints ($\mathbb{TC}$)): The temporal-constraint $\mathbb{TC}$ is a set of triples, in which each triple (i, j, k) represents that the interaction time of e_j minus the interaction time of e_i is not larger than k, i.e. $0 \le e_j.t - e_i.t \le k$.

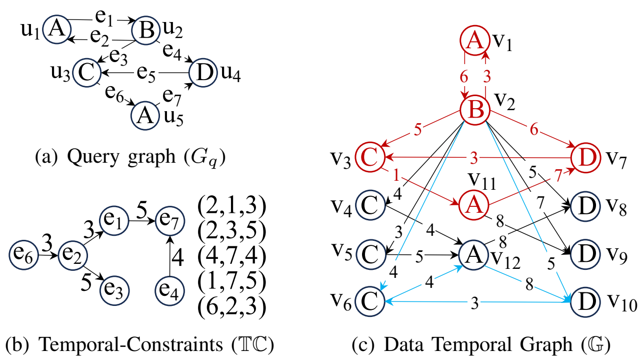
Based on Definition 3, we can observe that the $\mathbb{TC}$ is a simple directed edge-weighted graph, as shown in Figure 2(b). Moving forward, we present a formal definition of the problem, which involves the subgraph isomorphism considering *temporal constraints* on a temporal graph.

Definition 4 (Temporal-Constraint Subgraph Matching): Given a query graph $G_q = (V_q, E_q, \mathcal{L}_q)$, the temporal-constraints $\mathbb{TC} = \{(i, j, k)\}$ and a data temporal graph $\mathbb{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}, \mathcal{T})$, a *temporal-constraint subgraph matching* (abbreviated as *TCSM*) from G_q to $\mathbb{G}$ under the temporal-constraint $\mathbb{TC}$ is an injective function $f : E_q \to \mathcal{E}$ which satisfies:

1) Isomorphism: $\forall(u_q, v_q) \in E_q, \exists(u, v, t) \in \mathcal{E} \to f((u_q, v_q))=(u, v, t), \mathcal{L}_q(u_q)=\mathcal{L}(u), \mathcal{L}_q(v_q)=\mathcal{L}(v)$.
2) Temporal-constraint: Consider $E_q=\{e_1, e_2 ... e_{|E_q|}\}$, $\forall(i, j, k) \in \mathbb{TC} \to 0 \le f(e_j).t - f(e_i).t \le k$.

Problem of TCSM: Given a query graph G_q, the temporal-constraints $\mathbb{TC}$, and a data temporal graph $\mathbb{G}$, our task is to find all the subgraph matchings from G_q to $\mathbb{G}$ under $\mathbb{TC}$.

Example 1: Figure 2(a-c) show a toy example for query graph G_q, temporal-constraints $\mathbb{TC}$ and data temporal graph $\mathbb{G}$. Figure 2(a) shows that G_q contains 5 vertices and 7 directed edges, and each vertex in G_q has a label ($\mathcal{L}_q(u_1, u_5)$=A; $\mathcal{L}_q(u_2)$=B; ...). Figure 2(b) indicates that $\mathbb{TC}$ has 5 items ($0 \le e_1.t - e_2.t \le 3; 0 \le e_3.t - e_2.t \le 5; ...$), which can be assembled into a form of graph. In Figure 2(c), each vertices has a label, and each edge has a timestamp ($(v_1, v_2).t$=6; $(v_2, v_1).t$=3; ...). A *TCSM* from G_q to $\mathbb{G}$ under $\mathbb{TC}$ is highlighted in red, which is the matching



(a) Query graph (G_q)

(b) Temporal-Constraints ($\mathbb{TC}$)

(c) Data Temporal Graph ($\mathbb{G}$)

Fig. 2: A toy example.

$\{u_1, u_2, u_3, u_4, u_5 \rightarrow v_1, v_2, v_3, v_7, v_{11}\}$. We can observe that their labels are matching ($\mathcal{L}_q(u_1, u_5)$=A=$\mathcal{L}(v_1, v_{11})$; ...), and the temporal-constraints are satisfied ($e_1 \rightarrow (v_1, v_2), e_2 \rightarrow (v_2, v_1), 0 \leq (v_1, v_2).t - (v_2, v_1).t$=3 $\leq$ 3;...). However, if temporal information is disregarded, the subgraph marked in blue can be a traditional subgraph matching from G_q to G since their labels are matching, and it is not a *TCSM* since $e_2 \rightarrow (v_2, v_1), e_6 \rightarrow (v_6, v_{12}), (v_2, v_1).t - (v_6, v_{12}).t = -1$, conflict to $0 \leq e_2.t - e_6.t \leq 3$.

Theorem 1 (NP-hardness of TCSM): The Temporal-Constraint Subgraph Matching problem is NP-hard.

The proof of Theorem 1 can be established by considering a special case: if the time constraint *k* in each triplet of the temporal constraint is set to infinity, the Subgraph Matching problem can be reduced to the *TCSM* problem. Since Subgraph Matching is a well-known NP-hard problem, this reduction demonstrates the validity of the theorem.

Challenges: As noted, the *TCSM* problem is NP-hard. Traditional Subgraph Matching methods use filtering, verification, optimized matching order, and indexing (see Section 6), but these face challenges in the *TCSM* context. **First**, the temporal aspect adds complexities beyond static matching, requiring structural matches in temporal graphs while satisfying temporal constraints. **Second**, temporal constraints affect the matching order, increasing computational burden due to combinatorial factors. **Third**, using decomposition and indexing leads to unmanageable index sizes due to the added complexity of matching order. In summary, existing optimization techniques are not directly applicable to the *TCSM* problem.

III. BASIC ALGORITHM FOR TCSM

We propose an algorithm for $TCSM$ with vertex expansion in a Vertex-To-Vertex manner, abbreviated as TCSM-V2V. Before introducing the specific details of the algorithm, we need to consider the following key observations.

O1. Matching Orders. Traditional methods prioritize matching orders based on high-degree vertices, small candidate sets and the structure of query graph. In $TCSM$, the matching order must also consider the *temporal-constraint*.

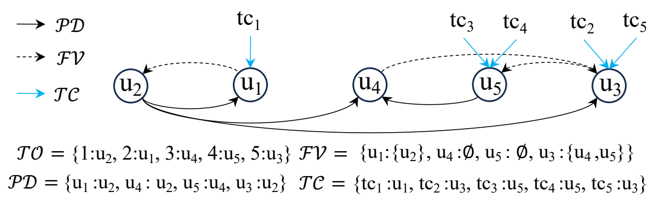


Fig. 3: Temporal-Constraint Query Graph $\mathcal{TCQ}$.

O2. Candidates Filtering. Filtering methods based on neighboring vertices, labels, and degrees are widely used to reduce the candidate data vertices. However, these methods often result in excessive candidates. Therefore, a more effective approach is needed to further minimize the candidates.

O3. Validity Checking. It is necessary to employ certain methods to determine whether a candidate data vertex can match a query vertex. This can be done by checking the edges between the query vertex and its matched neighbors, or by using matrix calculations with the state space method. However, both approaches are computationally demanding, with the latter being more suitable for smaller data graphs. Our goal is to develop a less resource-intensive and faster pruning method for validating candidate data vertices.

O4. Temporal-Constraint Checking. Our solution needs to find all subgraphs in the data graph that satisfy both the query and the *temporal-constraint* graph. This requires the algorithm to continuously check partial matches against both graphs.

Building on the four observations, we introduce the concept of a *temporal-constraint query graph*, denoted as $\mathcal{TCQ}$. The $\mathcal{TCQ}$ encompasses not only the structural information of the query graph and the temporal information of *temporal-constraint* graph but also the matching order of the vertices and the method by which each vertex candidate set is generated.

A. The construction of $\mathcal{TCQ}$

The $\mathcal{TCQ}$ is formed by four hash tables Temporal Order $\mathcal{TO}$, Prec Dictionary $\mathcal{PD}$, Forward Node $\mathcal{FV}$, and Time Constraint $\mathcal{TC}$. As shown in Figure 3, the vertices are aligned from left to right, indicating the order of the matching query vertices. For two vertices connected by a solid line, the preceding vertex is the *prec* (predecessor) of the subsequent one. For vertices connected by a dotted line, the preceding vertex is a member of the *forward vertex* of the latter vertex. The direction of the arrows shows the direction of the corresponding edges between the vertices in the query graph. The positions labeled tc_1 to tc_5 indicate where time-based pruning should be applied for each *temporal constraint*. The entire construction process is summarized in Algorithm 1 (details are below).

In the following, we show how to construct the four attached hash tables $\mathcal{TO}$, $\mathcal{PD}$, $\mathcal{FV}$, and $\mathcal{TC}$.

Definition 5 (Temporal-Constraint Support (tsup)): Given a query graph $G_q = (V_q, E_q, \mathcal{L}_q)$ and the *temporal constraints* graph, the *Temporal-Constraint Support* of a vertex $u \in V_q$ is the sum of the degrees of edges that contain u within the temporal constraint graph, i.e., $tsup(u) = \sum_{i=1}^{|E_q|} d(e_i), u \in e_i$.

Algorithm 1: $\mathcal{TCQ}(G_q, \mathbb{TC}, \mathbb{G})$

input : G_q, $\mathbb{TC}$ and $\mathbb{G}$.
output: four hash tables $\mathcal{TO}$, $\mathcal{PD}$, $\mathcal{FV}$, $\mathcal{TC}$ of $\mathcal{TCQ}$.
// compute $tsup$ for each query vertex
1 **for** each $u \in V_q$ **do** $tsup[u] = 0$;
2 **for** *each* $tc = (i, j, k) \in \mathbb{TC}$ **do**
3 $\quad tsup[e_i.u]$++; $tsup[e_i.v]$++; $tsup[e_j.u]$++; $tsup[e_j.v]$++;
4 $\mathcal{TO}[1] \leftarrow \arg\max_{u \in V_q}\{tsup[u]\}$;
5 $\mu \leftarrow 2$;
6 **while** $\mu \leq |V_q|$ **do**
7 $\quad V_q^{\mu} = \{u | u \in V_q \& u \notin \mathcal{TO}\}$, $N_\mu(u) = \{v | v \in \mathcal{TO} \& (u, v) \in E_q\}$;
8 $\quad \mathcal{TO}[\mu] \leftarrow \arg\max_{u \in V_q^{\mu}}\{|N_\mu(u)|\}$;
9 $\quad \mathcal{PD}[\mu] = \arg\min_{u \in N_\mu(\mathcal{TO}[\mu])}\{\mathcal{TO}'[u]\}$;
10 $\quad \mathcal{FV}(\mu) \leftarrow N_\mu(\mathcal{TO}[\mu]) \setminus \mathcal{PD}[\mu]$;
11 $\quad \mu$ ++;
12 **for** *each* $tc = (i, j, k) \in \mathbb{TC}$ **do**
13 $\quad \mathcal{TC}[tc] = \arg\max_{u \in \{e_i.u, e_i.v, e_j.u, e_j.v\}}\{\mathcal{TO}'[u]\}$;
14 **return** $(\mathcal{TO}, \mathcal{PD}, \mathcal{FV}, \mathcal{TC})$;

Temporal Order. The vertex ordering method is based on *tsup*, prioritizing query vertices that have the highest support and strongest connections with vertices already in $\mathcal{TO}$. First, *tsup* is calculated for each query vertex (Algorithm 1, Lines 1-3). The vertex with the highest *tsup* is selected as the starting vertex (Algorithm 1, Line 4). In case of a tie, the vertex with the fewest candidates is chosen; if the tie persists, selection is made randomly. Subsequent vertices are chosen based on their connections to vertices in $\mathcal{TO}$ (Algorithm 1, Lines 7-8).

Prec Dictionary. In the context of $\mathcal{TO}$, a query vertex is developed from another query vertex. Specifically, the latter becomes a candidate of $\mathcal{TCQ}$ because of the former, with the former referred to as the *prec* of the latter. The *Prec Dictionary* ($\mathcal{PD}$) stores the predecessor of each query vertex (Algorithm 1, Line 9). Beyond the initial vertex, candidate data vertices for matching are generated based on the partially matched subgraph. Since the $\mathcal{TCQ}$ graph stores structural relationships between vertices and their *prec*, candidate data vertices can be identified from the neighbors of already-matched predecessors. This approach reduces the size of candidate sets compared to other traditional candidate filtering methods.

Forward Vertex. To maintain the integrity of the query graph structure, *Forward Vertex* is created to track the neighboring vertices that already in $\mathcal{TCQ}$, excluding the *prec* (Algorithm 1, Line 10). This approach efficiently captures the structural relationships in the query graph while ensuring the accuracy of matched data vertices. When determining whether a data vertex can match a query vertex, it is essential to verify the presence of a corresponding data edge between the matched data vertex (the match of this query vertex's *forward vertex*) and this data vertex. If the edge does not exist, the data vertex cannot be considered a match for the query vertex.

Time Constraint. Once all query vertices within a *temporal constraint* have been matched, it's crucial to ensure that the current partial match adheres to the *temporal constraint*. To achieve this, $\mathcal{TC}$ is created to record the vertex that appears last in $\mathcal{TCQ}$ for each *temporal constraint*. (Algorithm 1, Lines 12-13). When determining whether a data vertex can match a query vertex, the temporal constraint must also be validated. If the current partial matching fails to satisfy the temporal constraints, then the data vertex cannot match the query vertex.

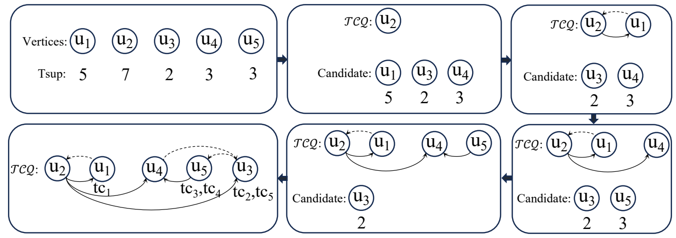


Fig. 4: The construction of $\mathcal{TCQ}$.

Theorem 2 (Complexity of building $\mathcal{TCQ}$): The time and space complexity of building the $\mathcal{TCQ}$ graph is $O(|V_q|^2 + |V_q| \cdot d_{\max} + |\mathcal{TC}|)$ and $O(|V_q| + |\mathcal{TC}|)$, respectively.

Proof of THEOREM 2: Initializing $tsup[u]$ for each $u \in V_q$ takes $O(|V_q|)$. Iterating over each temporal-constraint $tc \in \mathbb{TC}$ and updating $tsup$ takes $O(|\mathcal{TC}|)$. Finding the vertex with the maximum $tsup[u]$ takes $O(|V_q|)$. The main loop, which runs $|V_q| - 1$ times, involves constructing V_q^{μ} and finding the vertex with the maximum $|N_\mu(u)|$, each taking $O(|V_q|)$ per iteration. Additionally, finding the minimum in $N_\mu(\mathcal{TO}[\mu])$ takes $O(d_{\max})$ per iteration. Thus, the main loop has a time complexity of $O(|V_q|^2 + |V_q| \cdot d_{\max})$. Post-processing each $tc \in \mathbb{TC}$ takes $O(|\mathcal{TC}|)$ Therefore, the total time complexity is $O(|V_q|^2 + |V_q| \cdot d_{\max} + |\mathcal{TC}|)$.

The array $tsup$, hash tables $\mathcal{TO}$, $\mathcal{PD}$ and $\mathcal{FV}$ each require $O(|V_q|)$ space. Besides, the hash table $\mathcal{TC}$ require O($|\mathcal{TC}|$). Hence, the total space complexity is $O(|V_q| + |\mathcal{TC}|)$.

Example 2: Figure 4 shows the process of constructing $\mathcal{TCQ}$. The first step is to calculate the $tsup$ for each vertex to build the $\mathcal{TO}$ (recall Figure 2(b), $tsup(u_1)$=$d(e_1)$ + $d(e_2)$=5; $tsup(u_2)$=$d(e_1)+d(e_2)+d(e_3)+d(e_4)$=7; ...). Then, we choose the vertex u_2 with the highest $tsup$, and its neighboring vertices (u_1, u_3, u_4). Next, we choose the neighbor u_1 with the highest $tsup$, and build the solid line from u_2 to u_1 (which is stored in $\mathcal{PD}$), the dashed line from u_1 to u_2 (which is stored in $\mathcal{FV}$). Subsequently, we re-compute the neighboring vertices of u_2 and u_1, but the candidate are still u_3 and u_4. u_4 is added into the $\mathcal{TCQ}$, and (u_2, u_4) is added into $\mathcal{PD}$. Next, we re-compute the neighboring vertices and the candidate vertices become u_3 and u_5, u_5 is added into the $\mathcal{TCQ}$ and this process is iterated for the remaining vertices. In the final step, we build the $\mathcal{TC}$ to record the last appears in $\mathcal{TCQ}$ for all the vertices in each *temporal constraints*. $(\mathbb{TC}_1 = (2, 1, 3) \rightarrow e_2, e_1 \rightarrow u_2, u_1 \rightarrow \mathcal{TC}(tc_1) = u_1; \mathbb{TC}_2 = (2, 3, 5) \rightarrow e_2, e_3 \rightarrow u_1, u_2, u_3 \rightarrow \mathcal{TC}(tc_2) = u_3; ...)$.

B. *Algorithm* TCSM-V2V.

Given a temporal-constraint $\mathbb{TC}$, a query graph G_q and a data temporal graph $\mathbb{G}$, TCSM-V2V recursively expands

Algorithm 2: TCSM-V2V($G_q, \mathbb{TC}, \mathbb{G}$)

```
   input  : G_q, TC and G.
   output : all TCSMs M from G_q to G under TC.
   // generate initial candidate set (by def. 7)
1  for each u ∈ V_q do
2      for each v ∈ V do
3          if NLF(v, u) is true then u.C ← v;
   // generate TCQ graph
4  (TO,PD,FV,TC) ← TCQ(TC, G_q, G);
   // matching process.
5  λ ← 1, u ← TO[λ], M ← ∅;
6  for each v ∈ u.C do
7      M[u] ← v;
8      DFS (G, TO, PD, FV, TC, M, λ+1);
9  Procedure DFS (G,TO,PD,FV,TC,M,λ):
10     if λ = |TO| +1 then
11         print M;
12     else
13         u ← TO[λ], u' ← PD[u], u.C ← ∅;
           // generate candidate in current state
14         for each u_c ∈ N(M[u']) do
15             if L(u_c) = L(u) then
16                 u.C ← u_c;
17         for each v ∈ u.C do
18             if Validate(G, TC, FV, M, u, v, λ) is true then
19                 M[u] ← v;
20                 DFS (G, TO, PD, FV, TC, M, λ+1);
21 Function Validate(G, TC, FV, M, u, v, λ):
22     for each u' ∈ FN(u) do
23         if e(M[u'], v) ∉ E(G) then
24             return false;
25     for each tc ∈ TC do
26         if M do not satisfy time constraint tc then
27             return false;
28     return true;
```

partial matches by mapping query vertices to their respective candidates in accordance with $\mathcal{TO}$, and evaluates their structural and temporal information to determine whether they satisfy $\mathcal{FV}$ and $\mathcal{TC}$. The first step involves generating the initial candidates for each query vertex. This process aims to identify all potential data vertices while minimizing the candidate set. Inspired by the Neighborhood Label Frequency filtering technique [14], we introduce the *neighbor label filter* to effectively filter data vertices for each query vertex.

Definition 6 (Neighbor Label Filter (NLF)*):* Given a query graph $G_q = (V_q, E_q, \mathcal{L}_q)$ and a data temporal graph $\mathbb{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}, \mathcal{T})$ and the *de-temporal graph* $G = (V, E)$ of $\mathbb{G}$. Given $u \in V_q$ and $v \in V$, a *neighbor label filter* (v, u) is an injective function m: $v \to u$ that satisfies:

1) $L(v) = L(u)$.
2) $d.in(v) \geq d.in(u)$, $d.out(v) \geq d.out(u)$.
3) $\forall\ u' \in N(u)$, $\exists\ v' \in N(v)$, $L(v') = L(u')$.

Algorithm 2 presents our TCSM-V2V algorithm, which take $\mathbb{TC}$, G_q and $\mathbb{G}$ as input, and outputs all *temporal-constraint subgraph matchings* M from G_q to $\mathbb{G}$ under $\mathbb{TC}$. We first find all the possible initial candidates in $\mathbb{G}$ for each query vertex (Lines 1-3). Next, we generate $\mathcal{TO}$, $\mathcal{PD}$, $\mathcal{FV}$, and $\mathcal{TC}$ (Line 4). Then, we start the matching process by expanding the partial match in vertex-to-vertex manner recursively following

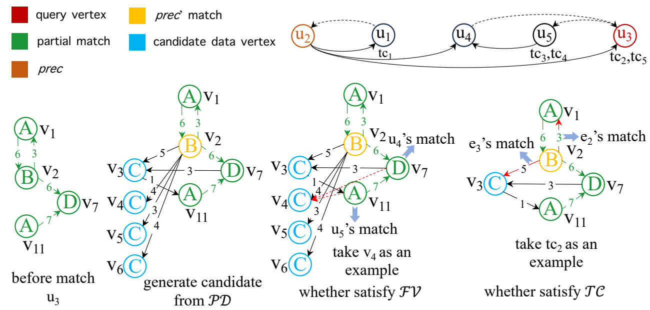


Fig. 5: A specific matching process of algorithm 2.

$\mathcal{TO}$ (Lines 5-8). If all query vertices have been matched, we output the result (Lines 10-11). Otherwise, we obtain the next query vertex u in $\mathcal{TO}$, the *prec* of u and set the candidate data vertices set of u to empty (Line 13). We extract the candidates data vertices of u based on the data vertex mapped to the *prec* of u and the structural correlation between u and the *prec* of u (Lines 14-16). For each candidate data vertex v of u, Line 18 checks whether there are edges between the candidate data vertex and the data vertices matched to the neighbors of u (Lines 22-24). If the query vertex u and the already matched query vertices can form a *temporal constraint* (tc) through their connecting edges, it is necessary to evaluate whether the temporal relationships of the partial match, after incorporating vertex v, satisfy the specified tc (Lines 25-27). If so, we expand the partial match by matching the next query vertex (Lines 19-20). Otherwise, the algorithm matches the next candidate vertex or backtracks to the last query vertex.

Example 3: Suppose the partial matching is $M = \{u_2 : v_2; u_1 : v_1; u_4 : v_7; u_5 : v_{11}\}$, we need to match vertex u_3. The first step involves generating candidate from $\mathcal{PD}$, so we match from the *prec* vertex of u_3 from $\mathcal{TCQ}$. The *prec* vertex is u_2 and $M(u_2) = v_2$. Among the neighbors of v_2, $\{v_3, v_4, v_5, v_6\}$ are generated , which have same label as u_3. Then, we check whether the candidate vertices satisfy $\mathcal{FV}$. Consider $M(u_3) = v_4$, since $\mathcal{FV}[u_3] = \{u_4,\ u_5\} \to v_4$ must have edge to $M(u_4) = v_7$ and $M(u_5) = v_{11}$, conflit to v_4 only has edges with v_2 and v_{12} in Figure 2(c). So v_4 will be removed from the candidates, and v_5, v_6 will also be removed in this step. Next, we check whether the candidate vertices satisfy $\mathcal{TC}$. Consider $M(u_3) = v_3$, since $\mathcal{TC}(tc_2, tc_5) = u_3$, $\mathbb{TC}_2, \mathbb{TC}_5 \to e_3.t - e_2.t \leq 5, e_2.t - e_6.t \leq 3$, and $e_2.t = (v_2, v_1).t = 3; e_3.t = (v_2, v_3).t = 5; e_6.t = (v_3, v_{11}).t = 1$. We can see that $0 \leq 5-3 \leq 5; 0 \leq 3-1 \leq 3$, so $M(u_3) = v_4$ is a correct matching.

Theorem 3 (Complexity of Algorithm TCSM-V2V*):* The time and space complexity of Algorithm TCSM-V2V is $O(2^{|V_q|} \times (|V| + d_{\max} + |\mathcal{TC}|))$ and $O(|V_q| + |\mathcal{TC}|)$, respectively.

Proof of THEOREM 3: The initialization (Lines 1-3) involves nested loops over $|V_q|$ and $|V|$ for checking $NLF(v, u)$, resulting in $O(|V_q| \times |V|)$. Generating the $\mathcal{TCQ}$ graph (Line 4) using the $\mathcal{TCQ}$ algorithm has a time complexity of $O(|V_q|^2 + |V_q| \cdot d_{\max} + |\mathcal{TC}|)$. The matching process (Lines 5-20), which includes initializing λ, u, and M, and performing DFS for each $v \in u.C$, involves exploring all possible subgraph matchings, leading to an exponential complexity of $O(2^{|V_q|})$. Specifically, each DFS call involves operations that take $O(|V_q| \times |V|)$ and validation steps that add $O(d_{\max})$ per edge check and $O(|\mathcal{TC}|)$ per time constraint check. Therefore, the overall time complexity is $O(2^{|V_q|} \times (|V| + d_{\max} + |\mathcal{TC}|))$.

The array $tsup$, hash tables $\mathcal{TO}$, $\mathcal{PD}$ and $\mathcal{FV}$ each require $O(|V_q|)$ space. Besides, the hash table $\mathcal{TC}$ require $O(|\mathcal{TC}|)$. The candidate set $u.C$ and the mapping M also require $O(|V_q|)$ space. Temporary variables used in loops and recursive calls consume additional $O(|V_q|)$ space. Hence, the total space complexity is $O(|V_q| + |\mathcal{TC}|)$.

IV. ADVANED ALGORITHM FOR TCSM

A. The construction of $\mathcal{TCQ}+$

In the previous section, we introduced a vertex-to-vertex matching approach. However, the necessity of identifying specific graph patterns with temporal constraints within temporal subgraphs leads to numerous time-consuming edge permutation checks when using vertex-based matching methods. Consequently, we are investigating an edge-based matching algorithm as an alternative. In the edge-to-edge matching method, the existing $\mathcal{TCQ}$ graph becomes inadequate, as the fundamental matching unit shifts from vertices to edges. To accommodate this transition, we propose the construction of a new graph, denoted as $\mathcal{TCQ}+$. This graph involves modifications to the Temporal Order ($\mathcal{TO}$), Precursor Dictionary ($\mathcal{PD}$), and Time Constraint ($\mathcal{TC}$), while also substituting the concept of Forward Vertex ($\mathcal{FV}$) with Forward Edge ($\mathcal{FE}$). The construction process for $\mathcal{TCQ}+$ is detailed below.

The edge-based matching order must take into account both temporal and structural constraints. Each edge to be matched should share at least a common vertex with already matched query edges in query graph, eliminating the need for post-matching connections and expediting the pruning process. By treating each edge as a vertex and connecting edges in a *temporal constraint* when they share a common vertex, we construct a Temporal-Constraint Forest (TCF). For each pair of edges within the *temporal-constraint*, if they are connected, we establish an edge between them. This forest, composed of interconnected trees, optimizes the matching process by capitalizing on the interconnected nature of edges.

We also define *temporal-constraint support (tsup)* for each query edge, which is the degrees of the edge in the *temporal constraint* graph, i.e., $tsup(e) = \sum_{i=1}^{|E_q|} d(e_i)$.

Forward Edge. To maintain the integrity of the query graph structure, we introduce the *forward edge*, is defined as a neighboring edge of e that already in $\mathcal{TO}$ and shares a common vertex with e distinct from the vertex shared between e and e's $prec$. Assuming that both two vertices of the query edge to be matched are part of the vertices in certain previously matched query edges, it is necessary to maintain the consistency of the candidate data edge's two vertices with the vertices of the data edges that correspond to these matched query edges.

Algorithm 3: $\mathcal{TCQ}+(G_q, \mathbb{TC}, \mathbb{G})$

input : G_q, $\mathbb{TC}$ and $\mathbb{G}$.
output: four hash tables $\mathcal{TO}$, $\mathcal{PD}$, $\mathcal{FE}$, $\mathcal{TC}$ of $\mathcal{TCQ}+$.
// creat TCF for *temporal-constraint* graph
1 $\mathscr{E} \leftarrow \emptyset$, $\mathscr{V} \leftarrow \{(N_e | e \in E_q)\}$;
2 **for** *each* $u \in V_q$ **do**
3   **for** *each* $e_i \in u.adje$ **do**
4   **for** *each* $e_j \in u.adje$ ($e_i \neq e_j$) **do**
5   **if** *(i, j, k)* $\in \mathbb{TC}$ **then**
6                 $\mathscr{E} \cup \{(N_{e_i}, N_{e_j})\}$;
7   **if** $(\mathscr{V}, \mathscr{E})$ *is cyclic* **then**
8                     $\mathscr{E} \setminus \{(N_{e_i}, N_{e_j})\}$;

// compute $tsup$ for each query edge
9 **for** each $e \in E_q$ **do** *tsup*$[e]$ = 0;
10 **for** *each* tc = *(i, j, k)* $\in \mathbb{TC}$ **do**
11   *tsup*$[e_i]$++; *tsup*$[e_j]$++;
12 $\mu \leftarrow 2$, $\mathcal{TO} \leftarrow \emptyset$, $\delta \leftarrow 0$;
13 $\mathcal{TO}[1] \leftarrow \arg\max_{e \in E_q}\{tsup[e]\}$;
14 **while** $\mu \leq |E_q|$ **do**
15   **for** each e', $(N_{e'}, N_{\mathcal{TO}[\mu-1]}) \in \mathscr{E}$ **do** $\delta \leftarrow \delta$ + 1;
16   **while** δ *>0* **do**
17         $N_\mu^{\mathscr{E}} = \{e | \exists e' \in \mathcal{TO}, (e, e') \in \mathscr{E}\}$;
18         $\delta \leftarrow \delta$ - 1, $\mathcal{TO}[\mu] \leftarrow \arg\max_{e \in N_\mu^{\mathscr{E}}}\{tsup[e]\}$;
19         $N_\mu = \{e | e \in \mathcal{TO} \& e \cap \mathcal{TO}[\mu] \neq \emptyset\}$;
20         $\mathcal{PD}[\mu] = \arg\min_{e \in N_\mu}\{\mathcal{TO}'[e]\}$;
21         $\mathcal{FE}[\mu] = \{e | e \in N_\mu \& e \cap \mathcal{TO}[\mu] \neq \mathcal{PD}[\mu] \cap \mathcal{TO}[\mu]\}$;
22   **for** each e', $(N_{e'}, N_{\mathcal{TO}[\mu]}) \in \mathscr{E}$ **do** $\delta \leftarrow \delta$ + 1;
23   μ++;
24     $N_\mu^E = \{e | \exists e' \in \mathcal{TO}, (e, e') \in E\}$;
25     $\mathcal{TO}[\mu] \leftarrow \arg\max_{e \in N_\mu^E}\{tsup[e]\}$;
26   N_μ = $\{e | e \in \mathcal{TO} \& e \cap \mathcal{TO}[\mu] \neq \emptyset\}$;
27     $\mathcal{PD}[\mu] = \arg\min_{e \in N_\mu}\{\mathcal{TO}'[e]\}$;
28     $\mathcal{FE}[\mu] = \{e | e \in N_\mu \& e \cap \mathcal{TO}[\mu] \neq \mathcal{PD}[\mu] \cap \mathcal{TO}[\mu]\}$;
29   μ++;
30 **for** each $tc = (i, j, k) \in \mathbb{TC}$ **do** $\mathcal{TC}[tc]$ = $\arg\max_{e \in \{e_i, e_j\}} \{\mathcal{TO}'[e]\}$;
31 **return** $(\mathcal{TO}, \mathcal{PD}, \mathcal{FE}, \mathcal{TC})$;

Algorithm 3 outlines the construction of $\mathcal{TCQ}+$. The process begins by building the TCF (Lines 1-8). Following this, the $tsup$ for each query edge is calculated (Lines 9-11). The first edge in $\mathcal{TO}$ is selected as the edge with the highest $tsup$ (Line 13). Subsequent edges within the same forest are selected based on their connection to a neighboring edge in $\mathcal{TO}$ and their $tsup$, with ties broken by choosing the edge with the smallest candidate set (Line 18). This process continues to organize all edges within each forest. When transitioning between forests, the selected edge shares at least one vertex with an edge already in $\mathcal{TO}$ and has the highest $tsup$ (Line 25). For each query edge, its $prec$ and *forward edge* is stored (Lines 20-21 and 27-28). For each *temporal-constraint*, the edge that appears last in $\mathcal{TO}$ is also recorded (Line 30).

Theorem 4 (Complexity of building $\mathcal{TCQ}+$): The time and space complexity of building the $\mathcal{TCQ}+$ graph is $O(|V_q|^2 \cdot d_{\max}^2 + |E_q|^2 + |\mathcal{TC}|)$ and $O(|E_q| + |\mathcal{TC}|))$, respectively.

Proof of THEOREM 4: Initializing the temporal-constraint edge $\mathcal{E}$ and checking cycles (Lines 1-8) takes $O(|V_q|^2 \cdot d_{\max}^2)$, where $d_{\max}$ is the maximum degree of the graph. Computing the support for each query edge (Lines 9-11) takes

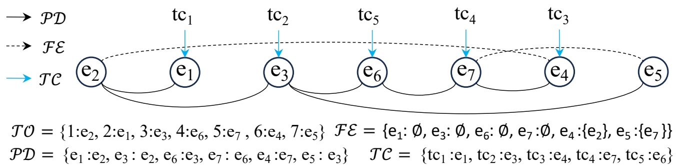


Fig. 6: Temporal-Constraint Query Graph $\mathcal{TCQ}+$

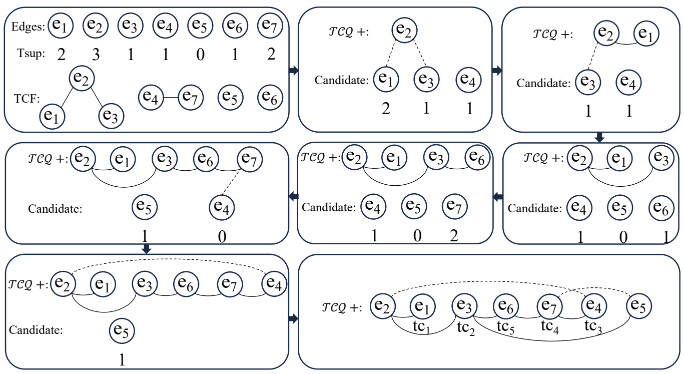


Fig. 7: The construction of $\mathcal{TCQ}+$

$O(|E_q|+|\mathcal{TC}|)$. The main loop (Lines 14-29), which involves selecting edges based on support and updating data structures, iterates $O(|E_q|)$ times with each iteration involving $O(|E_q|)$ operations for finding the maximum support and minimum operations, leading to $O(|E_q|^2)$. Therefore, the total time complexity is $O(|V_q|^2 \cdot d_{\max}^2 + |E_q|^2 + |\mathcal{TC}|)$.

Storage for array $tsup$, hash tables $\mathcal{TO}$, $\mathcal{PD}$ and $\mathcal{FE}$ requires $O(|E_q|)$ space. Besides, the hash table $\mathcal{TC}$ require O($|\mathcal{TC}|$). The temporary data structures and variables used during processing also require $O(|E_q|)$ space. Hence, the total space complexity is $O(|E_q| + |\mathcal{TC}|)$.

Example 4: Figure 7 shows the process of constructing $\mathcal{TCQ}$+. The first step involves constructing TCF and calculating the $tsup$ for each edge (recall Figure 2(b)). For instance, given the *temporal-constraint* tc = (2, 3, 5), it is obvious that edge e_2 and e_3 have common vertex u_2, so we build an edge between edge vertices N_{e_2} and N_{e_3}. And also build edges between N_{e_2} and N_{e_1}, between N_{e_4} and N_{e_7}. $tsup[e_1] = 2$; $tsup[e_2] = 3$; ... Then, we choose the edge e_2 with the highest $tsup$, and its neighboring edges (e_1, e_3, e_4) to *candidate*. Next, we choose the neighboring edge e_1 within the same forest that has the highest $tsup$, and build the solid line between e_2 and e_1 (which is stored in $\mathcal{PD}$). Subsequently, we re-compute the neighboring edges of e_2 and e_1, but the candidate are still e_3 and e_4. e_3 is added into the $\mathcal{TCQ}$+, and (e_2, e_3) is added into $\mathcal{PD}$. ... e_4 is added into the $\mathcal{TCQ}$+, and (e_7, e_4) is added into $\mathcal{PD}$ and build the dashed line between e_4 to e_2 (which is stored in $\mathcal{FE}$). This process is iterated for the remaining edges. In the final step, we build the $\mathcal{TC}$ to record the last appears in $\mathcal{TCQ}$+ for both edges in each *temporal constraint*. ($\mathbb{TC}_1 = (2, 1, 3) \rightarrow e_2, e_1 \rightarrow \mathcal{TC}(tc_1) = e_1; \mathbb{TC}_2 = (2, 3, 5) \rightarrow e_2, e_3 \rightarrow \mathcal{TC}(tc_2) = e_3; ...$).

Algorithm 4: TCSM-E2E($G_q, \mathbb{TC}, \mathbb{G}$)

input : G_q, $\mathbb{TC}$ and $\mathbb{G}$.
output: all *TCSMs* M from G_q to $\mathbb{G}$ under $\mathbb{TC}$.
// generate initial candidate set.
1 **for** *each* $e \in E_q$ **do**
2   **for** *each* $e_c \in \mathcal{E}$ **do**
3   **if** $LDF(e_c, e)$ is true **then** $e.C \leftarrow e_c$;
// generate $\mathcal{TCQ}$+ graph
4 ($\mathcal{TO}$,$\mathcal{PD}$,$\mathcal{FE}$,$\mathcal{TC}$) ← $\mathcal{TCQ}$+($\mathbb{TC}$, G_q, $\mathbb{G}$);
// matching process.
5 $\lambda \leftarrow 1$, $e \leftarrow \mathcal{TO}[\lambda]$, $M \leftarrow \emptyset$;
6 **for** *each* $e_c \in e.C$ **do**
7     $M[e] \leftarrow e_c$,;
8   *DFS* ($\mathbb{G}$, $\mathcal{TO}$, $\mathcal{PD}$, $\mathcal{FE}$,$\mathcal{TC}$,*M*,λ+1);
9 **Procedure** DFS($\mathbb{G}$, $\mathcal{TO}$, $\mathcal{PD}$, $\mathcal{FE}$, $\mathcal{TC}$, *M*, λ)**:**
10   **if** λ = $|\mathcal{TO}|$ +1 **then**
11   print M;
12   **else**
13         $e \leftarrow \mathcal{TO}[\lambda]$, $e' \leftarrow \mathcal{PD}[e]$, $e.C \leftarrow \emptyset$;
        // generate candidate in current state
14   **for** *each* $e_c \in M[e'].adje$ **do**
15   **if** $L(e_c.u)$=$L(e.u)$&$L(e_c.v)$=$L(e.v)$ **then**
16                 $e.C \leftarrow e_c$;
17   **for** *each* $e_c \in e.C$ **do**
18   **if** *Validate($\mathcal{FE}$, $\mathcal{TC}$,M, e,e_c,e') is true* **then**
19                 $M[e] \leftarrow e_c$;
20   *DFS* ($\mathbb{G}$, $\mathcal{TO}$, $\mathcal{PD}$, $\mathcal{FE}$,$\mathcal{TC}$,*M*,λ+1);
21 **Function** Validate($\mathcal{FE}$, $\mathcal{TC}$, *M*, *e*, e_c, e')**:**
22   **if** $M[\mathcal{FE}(e)] \cap e_c = \emptyset$ **then**
23   **return** false;
24   **for** *each* $tc \in \mathbb{TC}$ **do**
25   **if** *M do not satisfy time constraint tc* **then**
26   **return** false;
27   **return** true;

B. *Algorithm* TCSM-E2E

We also need to generate initial candidates for each query edge. To achieve this, we design a *label degree filter* to determine whether a data edge e_c can match a query edge e, thereby minimizing the size of the candidate set.

Definition 7 (Label Degree Filter(LDF)*):* Given a query graph $G_q = (V_q, E_q, \mathcal{L}_q)$ and a data temporal graph $\mathbb{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}, \mathcal{T})$. Given $e \in E_q$ and $e_c \in \mathcal{E}$, a *neighbor label filter* (e_c, e) is an injective function h: $e_c \rightarrow e$ that satisfies:

1) $L(e.u)$ = $L(e_c.u)$, $L(e.v)$ = $L(e_c.v)$.
2) $d.in(e_c.u) \geq d.in(e.u)$, $d.out(e_c.u) \geq d.out(e.u)$, $d.in(e_c.v) \geq d.in(e.v)$, $d.out(e_c.v) \geq d.out(e.v)$.

Algorithm 4 presents our TCSM-E2E algorithm, which take $\mathbb{TC}$, G_q and $\mathbb{G}$ as input, and outputs all *TCSM* from G_q to $\mathbb{G}$ under $\mathbb{TC}$. We first find all the possible candidates for each query edge (Lines 1-3). Next, we generate $\mathcal{TO}$, $\mathcal{PD}$, $\mathcal{FE}$, and $\mathcal{TC}$ (Line 4). Then, we start the matching process by expanding the partial match in edge-to-edge manner recursively following $\mathcal{TO}$ (Lines 5-8). If all query edges have been matched, we output the match (Lines 10-11). Otherwise, we obtain the next query edge e in $\mathcal{TO}$, the *prec* of e and set the candidate data edges set of e to empty (Line 13). We extract the candidates data edges of e based on the data edge matched to the *prec* of e and the structural correlation between e and the *prec* of e (Lines 14-16). For each candidate data edge e_c of

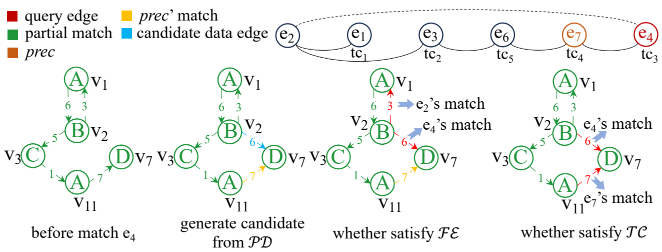


Fig. 8: A specific matching process of algorithm 4.

e, Line 18 checks whether there is an intersection between the candidate data edge and the match of $\mathcal{FE}(e)$. If the query edge e and the already matched query edge can form a temporal constraint (tc), it is necessary to evaluate whether the temporal relationships of the partial match, after incorporating data edge e_c, satisfy the specified tc (Lines 24-26). If so, we expand the partial match by matching the next query edge (Lines 19-20). Otherwise, the algorithm continue to matches the next candidate data edge or backtracks to the last query edge.

Example 5: Suppose the current partial match M = $\{e_2$:$(v_2, v_1, 3)$; e_1 :$(v_1, v_2, 6)$; e_3: $(v_2, v_3, 5)$; e_6:$(v_3, v_{11}, 1)$; e_7: $(v_{11}, v_7, 7)\}$, we need to match e_4. The first step involves generating candidate from $\mathcal{PD}$, so we match from the $prec$ edge of e_4 from $\mathcal{TCQ}+$. The $prec$ edge is e_7 and $M(e_7) = (v_{11}, v_7, 7)$. Among the neighboring edges of $(v_{11}, v_7, 7)$, $\{(v_2, v_7, 6)\}$ is generated, which is an edge between vertex with label same as B and v_7. Then, we check whether the candidate edge satisfy $\mathcal{FE}$. Consider $M(e_4) = (v_2, v_7, 6)$, since $\mathcal{FE}[e_4] = \{e_2\} \rightarrow (v_2, v_7, 6)$ must have common vertex with $M(e_2) = (v_2, v_1, 3)$ in Figure 2(c). There exist a common vertex between $(v_2, v_7, 6)$ and $(v_2, v_1, 3)$, so $M(e_4) = (v_2, v_7, 6)$ can satisfy $\mathcal{FE}$. Next, we check whether the candidate edge satisfy $\mathcal{TC}$. Consider $M(e_4) = (v_2, v_7, 6)$, since $\mathcal{TC}(tc_3) = e_4$, $\mathbb{TC}_3 \rightarrow e_7.t - e_4.t \leq 4$, and $e_7.t = (v_{11}, v_7, 7).t = 7; e_4.t = (v_2, v_7, 6).t = 6$. We can see that $0 \leq 7 - 6 \leq 4$, so $M(e_3) = (v_2, v_7, 6)$ is a correct matching.

Theorem 5 (Complexity of Algorithm TCSM-E2E*):* The time and space complexity of Algorithm TCSM-E2E is $O(2^{|E_q|} \times (|E| + d_{\max} + |\mathcal{TC}|))$ and $O(|E_q| + |\mathcal{TC}|)$, respectively.

Proof of THEOREM 5: The initialization (Lines 1-3) involves nested loops over $|E_q|$ and $|E|$ for checking $LDF(e_c, e)$, resulting in $O(|E_q| \times |E|)$. Generating the $\mathcal{TCQ}+$ graph (Line 4) has a time complexity of $O(|E_q|^2 + |V_q| \cdot d_{\max}^2 + |\mathcal{TC}|)$. The matching process (Lines 5-20), including DFS and validation, has a worst-case exponential time complexity due to the depth-first search exploring all possible subgraph matchings, leading to $O(2^{|E_q|})$. Each DFS call involves operations that take $O(|E_q| \times |E|)$ and validation steps that add $O(d_{\max})$ per edge check and $O(|TC|)$ per time constraint check. Therefore, the overall time complexity is $O(2^{|E_q|} \times (|E| + d_{\max} + |\mathcal{TC}|))$.

Storage for array $tsup$, hash tables $\mathcal{TO}$, $\mathcal{PD}$ and $\mathcal{FE}$ requires $O(|E_q|)$ space. Besides, the hash table $\mathcal{TC}$ require O($|\mathcal{TC}|$). The candidate set $e.C$ and the mapping M also require $O(|E_q|)$ space. Temporary variables used in loops and recursive calls consume additional $O(|E_q|)$ space. Hence, the total space complexity is $O(|E_q| + |\mathcal{TC}|)$.

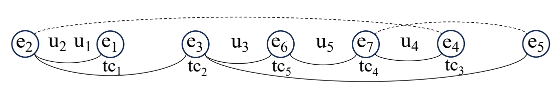


Fig. 9: $\mathcal{TCQ}+$ in algorithm TCSM-EVE.

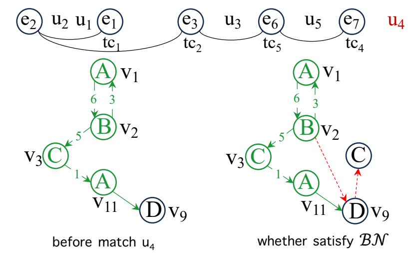


Fig. 10: A specific matching process of algorithm 5.

C. *Algorithm* TCSM-EVE

The key difference between algorithm TCSM-EVE and algorithm TCSM-E2E lies in whether vertex pre-matching occurs after each edge matching, depending on the addition of a new vertex. Vertex pre-matching helps prune invalid matches by ensuring that when a new query vertex u is added, its matched neighbors align with all *backward neighbors* of u.

Example 6: Figure 9 illustrates the construction of the $\mathbb{TCQ}+$ in the TCSM-EVE algorithm. After each query edge, we check for any newly introduced vertices; if any are present, they are marked accordingly. For instance, when edge e_2 is added, the corresponding vertices u_2 and u_1 are introduced and they are marked after e_2. When edge e_1 is incorporated, no new vertices are introduced and as a result, no vertices are marked. Similarly, when e_3 is added, u_3 is marked after it.

Definition 8 (Backward Neighbor): Given a query graph $G_q = (V_q, E_q, \mathcal{L}_q)$, $\mathcal{PD}$, and a vertex $u \in V_q$ which is added because of query edge e, a *backward neighbor* of u, denoted as $\mathcal{BN}(u)$, is a set that contains all neighbor vertices of u in V_q except the common vertex between e and $prec$ of e.

Algorithm 5 outlines our TCSM-EVE algorithm, which shares similarities with the TCSM-E2E; thus, certain portions are omitted for brevity. The matching process begins by recursively expanding the partial match in an edge-to-edge manner according to $\mathcal{TO}$ (Lines 1-3). Before matching the second query edge, we verify whether the two data vertices of the first edge can match the two query vertices of first query edge. This entails confirming that there are neighboring vertices of the data vertex that can match $\mathcal{BN}(u)$ of query vertex u. If so, we proceed to match the next edge (Lines 4-5). Upon matching all query edges, we output the match (Lines 7-8). If not, we retrieve the next query edge e, generating the candidate data edges set for e (Lines 10-13). For each candidate data edge e_c of e, Line 15 evaluates the structural and temporal constraints of the match between the candidate

Algorithm 5: TCSM-EVE($G_q, \mathbb{TC}, \mathbb{G}$)

```
   input : G_q, 𝕋ℂ and 𝔾.
   output: all TCSMs M from G_q to 𝔾 under 𝕋ℂ.
   // generate initial candidate set.
   // ...
   // generate 𝒯𝒞𝒬+ graph
   // ...
   // matching process.
1  λ ← 1, e ← 𝒯𝒪[λ], M ← ∅;
2  for each e_c ∈ e.C do
3      M[e] ← e_c;
4      if Vmatch(e.u, e_c.u) & Vmatch(e.v, e_c.v) then
5          DFS(𝔾, 𝒯𝒪, 𝒫𝒟, ℱℰ,𝒯𝒞,M, λ+1);

6  Procedure DFS(𝔾, 𝒯𝒪, 𝒫𝒟, ℱℰ,𝒯𝒞,M, λ):
7      if λ = |𝒯𝒪| +1 then
8          print M;
9      else
10         e ← 𝒯𝒪[λ], e′ ← 𝒫𝒟[e], e.C ← ∅;
           // generate candidate in current state
11         for each e_c ∈ M[e′].adje do
12             if L(e_c.u)=L(e.u)&L(e_c.v)=L(e.v) then
13                 e.C ← e_c;

14         for each e_c ∈ e.C do
15             if M[ℱℰ(e)] ∩ e_c ≠ ∅ & M ∪ e_c can satisfy 𝕋ℂ then
16                 M[e] ← e_c;
17                 if {e.u, e.v}−{e′.u, e′.v} ≠ ∅ then
18                     u ← {e.u, e.v}−{e′.u, e′.v};
19                     v←{e_c.u, e_c.v}−{M[e′].u, M[e′].v};
20                     if Vmatch(u, v) then
21                         DFS(𝔾, 𝒯𝒪, 𝒫𝒟, ℱℰ,𝒯𝒞,M, λ+1);
22                 else
23                     DFS(𝔾, 𝒯𝒪, 𝒫𝒟, ℱℰ,𝒯𝒞,M, λ+1);

24 Function Vmatch(u, v):
25     for each u′ ∈ BN(u) do
26         if ∄ v′ ∈ N(v), L(v′) = L(u′) then
27             return false;

28     return true;
```

edge and the query edge. If a match is feasible, we include it in the partial match (Line 16), and we check whether a new vertex emerges in this layer of edge matching (Line 17). If no not, we proceed to match the next query edge (Line 23). However, if a new vertex does emerge, we assess whether this data vertex can match the corresponding query vertex (Line 20). If the newly added vertex can be matched, we continue matching the next query edge (Lines 23).

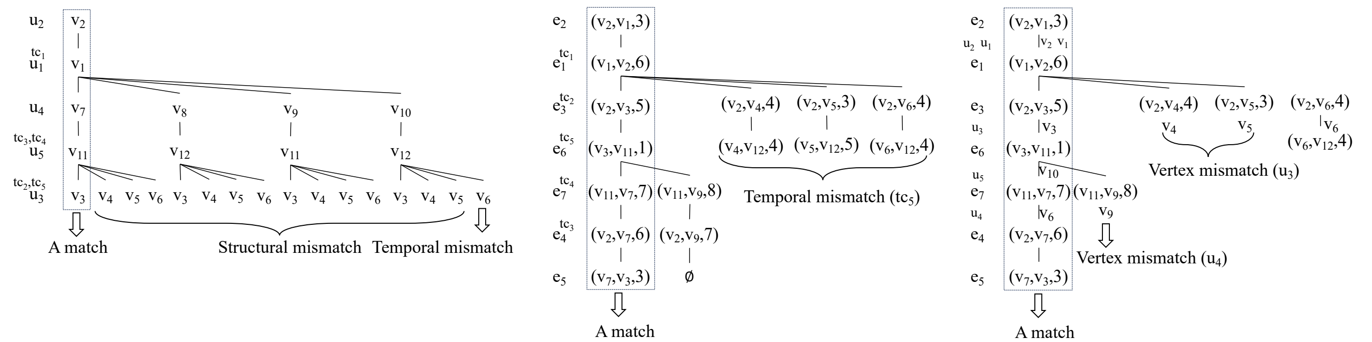
Example 7: To facilitate a better understanding of Algorithm 5 better, we provide the following illustrative example. When matching newly added vertex u_4 after edge e_7 is matched, we find that $\mathcal{BN}(u_4)$ = $\{u_2, u_3\}$. We need to check whether the neighbor vertex of v_9 (the match of u_4) can match both u_2 and u_3. For simplicity, we will assume that a match is feasible if the labels correspond. The label of u_2 and u_3 is B and C, respectively. In Figure 2(c), v_9 has neighboring vertices with labels B and A, allowing it to match u_2, but not u_3.

Theorem 6 (Complexity of Algorithm TCSM-EVE*):* The time and space complexity of Algorithm TCSM-EVE is $O(2^{|E_q|} \times (|E| + d_{\max} + |TC|))$ and $O(|E_q| + |\mathcal{TC}|)$, respectively.

Proof of THEOREM 6: The initialization and generating of the $\mathcal{TCQ}+$ graph (omitted details) involve operations similar to those in previous algorithms, which typically take $O(|E_q|^2 + |V_q|^2 \cdot d_{\max}^2 + |TC|)$. The matching process (Lines 1-6) involves nested loops over the candidate set $e.C$ and checking vertex matches, leading to $O(|E_q| \times |E|)$. The DFS procedure (Lines 7-25) explores all possible subgraph matchings, resulting in a worst-case exponential complexity of $O(2^{|E_q|})$. Each DFS call and the function $Vmatch$ (Lines 26-30) involve operations that add $O(d_{\max})$ per edge check and $O(|TC|)$ per time constraint check, thus the overall time complexity is $O(2^{|E_q|} \times (|E| + d_{\max} + |TC|))$.

Storage for array $tsup$, hash tables $\mathcal{TO}$, $\mathcal{PD}$ and $\mathcal{FE}$ requires $O(|E_q|)$ space. Besides, the hash table $\mathcal{TC}$ require O($|\mathcal{TC}|$). The candidate set $e.C$ and the mapping M also require $O(|E_q|)$ space. Temporary variables used in loops and recursive calls consume additional $O(|E_q|)$ space. Hence, the total space complexity is $O(|E_q| + |\mathcal{TC}|)$.

Example 8: Figure 11 illustrates the matching trees generated by our algorithms. As shown, edge-based matching outperforms vertex-based matching by reducing the size of the matching tree and significantly enhancing efficiency. Additionally, integrating vertex matching further minimizes the tree size by pruning unsuitable matches early in the process. Thus, the TCSM-EVE algorithm achieves superior pruning effectiveness compared to the other two algorithms, as further confirmed in subsequent experiments. Figure 11(a) presents the matching process obtained using the TCSM-V2V algorithm, as applied to Figure 2. The match $M = \{u_2 : v_2; u_1 : v_1; u_4 : v_7; u_5 : v_{11}; u_3 : v_3\}$ is identified as a valid match (highlighted within a dotted box). The remaining matches either fail to structurally align with the query graph, do not satisfy the temporal constraints, or are unable to generate the corresponding candidate vertices. For instance, consider the match $M = \{u_2 : v_2; u_1 : v_1; u_4 : v_7; u_5 : v_{11}; u_3 : v_4\}$, which does not structurally match the query graph. In this case, there should be edges between v_4 and v_{11}, as well as between v_4 and v_7 (corresponding to edges between u_3 and u_5 and between u_3 and u_4 in the query graph), but such edges are absent. Conversely, the match $M = \{(u_2 : v_2; u_1 : v_1; u_4 : v_{10}; u_5 : v_{12}; u_3 : v_6\}$ fails to satisfy the temporal constraints. Constraint tc_5 indicates that the edge between v_6 and v_{12} must correspond to the edge between v_2 and v_1 in terms of timing, with a time difference of less than 3 ($0 \le e_2.t - e_6.t \le 3 \to 0 \le (M[u_2], M[u_1]).t - (M[u_3], M[u_5]).t \le 3 \to 0 \le (v_2, v_1).t - (v_6, v_{12}).t \le 3 \to 0 \le 3 - 4 \le 3$). Figure 11(b) presents the matching process obtained using the TCSM-E2E algorithm, as applied to Figure 2. The match M = $\{e_2{:}(v_2, v_1, 3);\ e_1 :(v_1, v_2, 6);\ e_3{:}\ (v_2, v_3, 5);\ e_6{:}(v_3, v_{11}, 1);\ e_7{:}\ (v_{11}, v_7, 7);\ e_4{:}\ (v_2, v_7, 6);\ e_5{:}(v_7, v_3, 3)\}$ is identified as a valid match (highlighted within a dotted box). The remaining matches either fail to structurally align with the query graph, do not satisfy the temporal constraints, or are unable to generate the corresponding candidate data edges. For example, the match $M = \{e_2{:}(v_2, v_1, 3);\ e_1 :(v_1, v_2, 6);\ e_3{:}\ (v_2, v_4, 4);\ e_6{:}(v_4, v_{12}, 4)\}$ fails to satisfy the temporal constraints. Constraint tc_5 indicates that the time associated with the edge $(v_4, v_{12}, 4)$ must be less than that of the edge



(a) Algorithm **TCSM-V2V** (b) Algorithm **TCSM-E2E** (c) Algorithm **TCSM-EVE**

Fig. 11: The matching trees of our algorithms.

$(v_2, v_1, 3)$, with a difference of less than 3 ($0 \le e_2.t - e_6.t \le 3 \rightarrow 0 \le (v_2, v_1, 3).t - (v_4, v_{12}, 4).t \le 3 \rightarrow 0 \le 3 - 4 \le 3$). However, the matches for e_6 and e_2 clearly do not conform to this temporal relationship. In another partial match, $M = \{e_2{:}(v_2, v_1, 3);\ e_1\ {:}(v_1, v_2, 6);\ e_3{:}\ (v_2, v_3, 5);\ e_6{:}(v_3, v_{11}, 1);\ e_7{:}\ (v_{11}, v_9, 8);\ e_4{:}\ (v_2, v_9, 7)\}$, when attempting to match e_5, the match of e_5's $prec$ $(v_2, v_3, 5)$ fails to find a neighboring edge directed toward v_3 from a vertex labeled D. As a result, there are no corresponding candidate edges. Figure 11(c) show the matching process of using TCSM-EVE algorithm. The match $M = \{((v_3, v_1, 3),\ e_2),\ (v_3, u_2),\ (v_1, u_1),\ ((v_1, v_3, 6),\ e_1),\ ((v_3, v_{14}, 5),\ e_3),\ (v_{14}, u_3),\ (\ (v_{14}, v_{18}, 1),\ e_6),\ (v_{18}, u_5),\ ((v_{18}, v_7, 7),\ e_7),\ (v_7, u_4),\ ((v_3, v_7, 6),\ e_4),\ ((v_7, v_{14}, 3),\ e_5))\}$ is a correct match. Such as the candidate data vertex v_4 to match u_3, the partial match $M = \{(v_2, v_1, 3){:}\ e_2;\ u_2, v_2\ {:};\ u_1 : v_1;\ e_1{:}\ (v_1, v_2, 6);\ e_3{:}(v_2, v_4, 4)\}$. u_3 have *backford neighbor* u_4 and u_5 with label D, A, respectively. But, the data vertex v_4 only have two neighbors vertex with B, A. So, the data vertex v_4 can not match u_3.

V. EXPERIMENTS

A. Experimental setup

In this section, we evaluate the performance of the algorithms we proposed across various datasets and query configurations. We implement all our algorithms in C++. All experiments are conducted on a Linux machine equipped with a 2.9GHz AMD Ryzen 3990X CPU and 256GB RAM running CentOS 7.9.2 (64-bit). The experimental results are meticulously detailed in the subsequent parts of this section.

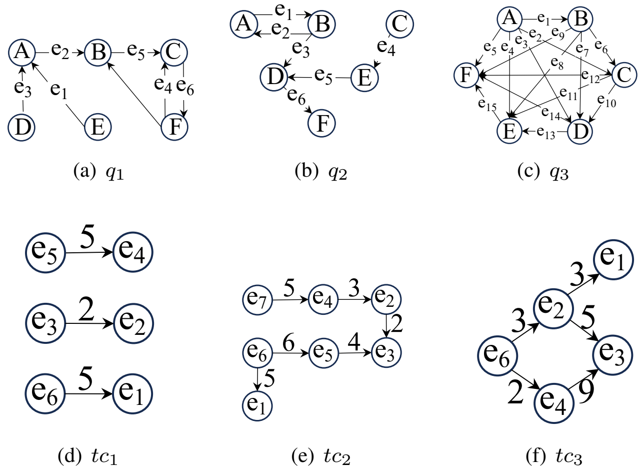
Datasets. We utilize 7 datasets that have been commonly used for evaluating previous temporal subgraph matching methods [15]. Table I presents their statistics. All the datasets are downloaded from https://snap.stanford.edu/data/ and are listed in increasing order of the number of edges. CM (*CollegeMsg*) is the datasets of messages on a facebook-like platform at UC-Irvine. EE (*email-Eu-core-temporak*) is the datasets of e-mails between users at a research institution. MO (*sx-mathoverflow*) is the datasets of comments, questions, and answers on Math Overflow. UB (*sx-askubuntu*) is the datasets of comments, questions, and answers on Ask Ubuntu. SU (*sx-superuser*) is the datasets of comments, questions, and answers on Super User. WT (*wiki-talk-temporal*) is the sets of users editing talk pages on Wikipedia. SO (*sx-stackoverflow*) is the datasets of comments, questions, and answers on Stack Overflow.

TABLE I: Data Temporal Graph $\mathcal{G}$

Dataset	$\|V\|$	$\|\mathcal{E}\|$	$\|E\|$	Time span
CM	1,899	59,835	20,296	193 days
EE	986	332,334	24,929	803 days
MO	24,818	506,550	239,978	2,350 days
UB	159,316	964,437	596,933	2,613 days
SU	194,085	1,443,339	924,886	2,773 days
WT	1,140,149	7,833,140	3,309,592	2,320 days
SO	2,601,977	63,497,050	36,233,450	2,774 days

Queries and Temporal-Constriants. We employed three labeled queries and three temporal constraints between edges, as illustrated in Figure 12. Each of these three queries comprises six nodes, while the temporal constraints vary in structure, being linear, tree-shaped, and graph-shaped, respectively. Furthermore, we conducted experiments with queries containing more vertices and temporal constraints featuring more edges, yielding similar conclusions. However, due to space limitations, we refrain from reporting on those cases here. Additionally, unless otherwise specified, comparisons are made between query graph q_1 and *temporal-constraint* tc_2.

Algorithms. The RI-DS algorithm is a classical method for subgraph isomorphism in static graphs, utilizing compatibility domains to filter candidate target vertices before matching. In our experiments, we establish a baseline with subgraphs matched by RI-DS under an additional temporal constraint. The experimental section evaluates four algorithms: RI-DS, TCSM-V2V, TCSM-E2E, and TCSM-EVE. We also modified state-of-the-art continuous subgraph matching algorithms (SymBi [16], Turboflux [17], Graphflow [18], SJ-Tree [19], and IEDyn [20]) to ensure them satisfy the temporal order, incorporating them into our comparisons. All algorithms are implemented in C++, with source codes obtained from [21].



(a) q_1 (b) q_2 (c) q_3

(d) tc_1 (e) tc_2 (f) tc_3

Fig. 12: Queries q and Temporal-Constriants tc.

TABLE II: Running time of various methods (second).

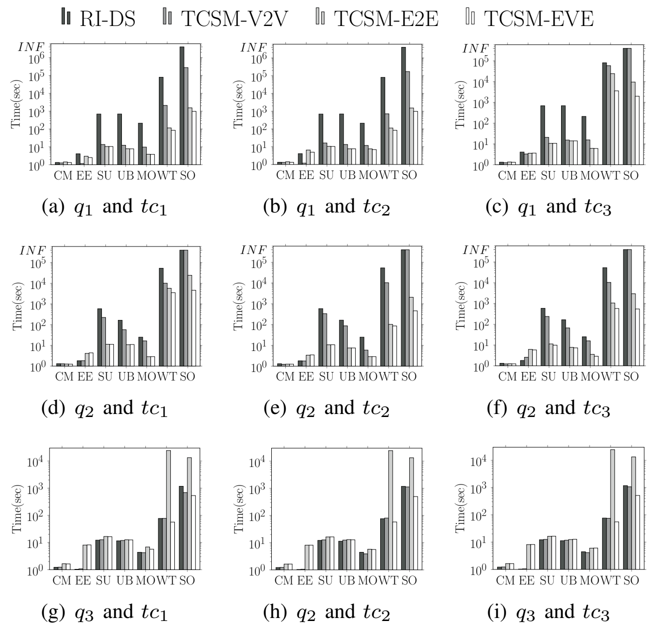
Methods	EE	MO	UB	SU	WT
Symbi	0.297	2.443	9.578	12.116	inf
Turboflux	8.268	153.648	166.463	1422.53	inf
Graphflow	5.797	411.795	269.438	2271.52	inf
SJ−Tree	6.244	4464.54	781.256	521.975	inf
IEDyn	2.573	11.640	94.889	100.663	inf
RI−DS	4.077	212.3	699.223	693.222	80753.1
TCSM−V2V	1.194	11.791	13.374	16.228	712.753
TCSM−E2E	6.506	7.581	7.629	10.528	113.242
TCSM−EvE	4.937	6.885	7.628	10.336	84.821

B. Experimental Results

Exp-1: Running time of various temporal methods. Table II presents a comparative analysis of the performance of six temporal algorithms alongside our proposed methods across five datasets. The designation "inf" denotes instances where the runtime exceeded three days. The results indicate that, while our algorithms exhibit performance comparable to existing methods on smaller datasets (such as EE), they demonstrate superior efficiency on medium-sized datasets relative to mainstream matching algorithms. Furthermore, on large-scale datasets comprising millions of vertices, our algorithms significantly outperform the existing approaches, underscoring their exceptional efficiency and scalability.

Exp-2: Runtime of our algorithms V.S. baselines. Figure 13 illustrates the running times for RI-DS, TCSM-V2V, TCSM-E2E, and TCSM-EVE across all datasets, consistently yielding identical subgraphs. The results reveal the following:

TCSM-V2V generally outperforms RI-DS in terms of speed, particularly with larger queries and denser datasets. This enhanced efficiency can be attributed to two key features of TCSM-V2V: 1) the ordering of query vertices based on structural and *temporal constraints*, and 2) the generation of candidate vertices from the preceding vertex. These strategies effectively reduce unnecessary computations through early filtering and the minimization of redundant candidates.

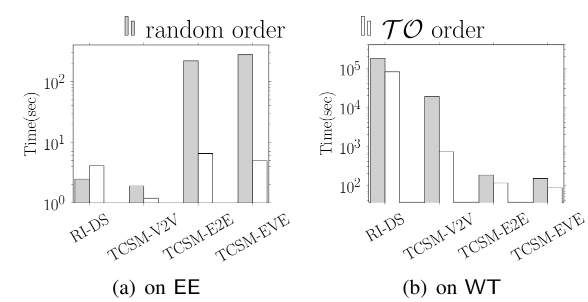


(a) q_1 and tc_1 (b) q_1 and tc_2 (c) q_1 and tc_3

(d) q_2 and tc_1 (e) q_2 and tc_2 (f) q_2 and tc_3

(g) q_3 and tc_1 (h) q_2 and tc_2 (i) q_3 and tc_3

Fig. 13: Runtime with different q and tc.



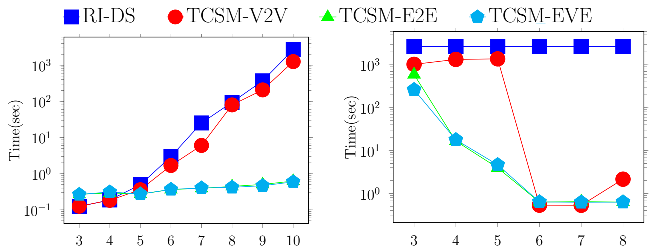
(a) on EE (b) on WT

Fig. 14: Runtime with different matching orders.

However, when the query graph is fully connected, RI-DS surpasses all other algorithms, with its runtime becoming independent of the *temporal-constraint* graph.

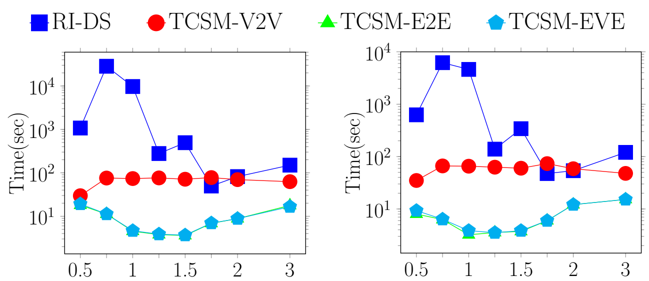
TCSM-EVE demonstrates superior than TCSM-E2E as it incorporates vertex pre-matching pruning after each match. Although their performance is comparable on smaller datasets, TCSM-EVE exhibits significantly faster execution on larger datasets. In the worst-case scenario, where the query graph is fully connected, TCSM-EVE maintains strong performance.

Exp-3: Ramdom matching order V.S. $\mathcal{TO}$. Figure 14 presents bar charts comparing the computational times of various algorithms using both random matching order and our proposed $\mathcal{TO}$, based on the EE and WT datasets. Similar conclusions were observed across other datasets. These results demonstrate that the matching order generated by our algorithms is highly effective and significantly enhances performance. The findings emphasize the importance of integrating both temporal information of *temporal-constraint* and structural constraints of the query graph when determining the matching order in temporal graphs. Notably, our proposed order was found to be more than ten times faster than a random



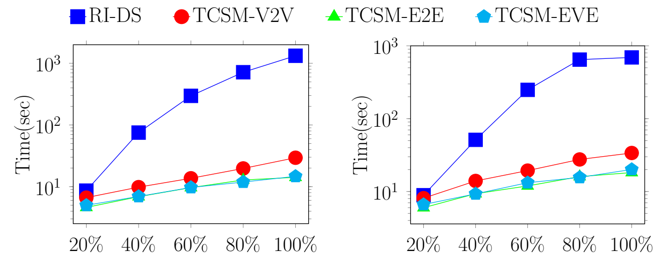
(a) vary $|q|$ on CM (b) vary $|tc|$ on CM

Fig. 15: Runtime with different $|q|$ or $|tc|$.



(a) vary $\frac{|E_q|}{|V_q|}$ on UB (b) vary $\frac{|E_q|}{|V_q|}$ on SU

Fig. 16: Runtime with queries of different density.



(a) vary $|\mathcal{E}|$ on UB (b) vary $|\mathcal{E}|$ on SU

Fig. 17: Runtime with different data temporal graph.

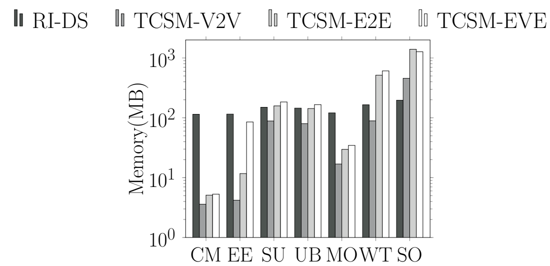


Fig. 18: The memory usages of the algorithms.

order, further validating the robustness of our algorithm.

Exp-4: Scalability with different size of queries and *temporal-constraints*. Figure 15 presents line charts showing the running times of various algorithms across a series of queries, with vertex counts ranging from three to ten and temporal constraints varying from three to eight on CM. The results reveal that running times generally increase for all algorithms as $|q|$ grows, reflecting the corresponding rise in computational complexity. Notably, the RI-DS and TCSM-V2V algorithms experience sharp increases in runtime, contrasting with the more gradual rise seen in TCSM-E2E and TCSM-EVE. Since the RI-DS algorithm is designed for static graphs, changes in $|tc|$ do not impact its performance. The runtime of the TCSM-V2V algorithm initially increases slowly as $|tc|$ grows, then drops sharply before stabilizing. In contrast, the TCSM-E2E and TCSM-EVE algorithms demonstrate a trend of gradually decreasing runtime as $|tc|$ increases. These trends suggest that the TCSM-E2E and TCSM-EVE algorithms offer better stability compared to TCSM-V2V.

Exp-5: Scalability with queries of different density $|E_q|/|V_q|$. Figure 16 illustrates line charts depicting the running times of various algorithms across the UB and SU datasets, with edge-to-vertex ratios ranging from 0.5 to 3. Despite differences in computational efficiency, all algorithms yielded identical subgraphs in these experiments. Notably, the TCSM-E2E and TCSM-EVE algorithms demonstrated optimal performance when the edge-to-vertex ratio approached 1, suggesting that a balanced ratio enhances their efficiency by simplifying graph topology. In contrast, the TCSM-V2V algorithm exhibited diminished performance when the ratio fell below 1, indicating that it may depend on a more complex graph structure for optimal operation, likely due to its specific methodology for handling vertex-to-vertex comparisons. The RI-DS and TCSM-V2V algorithms displayed an initial increase in runtime, followed by a decline as the ratio increased. Overall, their performance exhibited relatively large fluctuations and less stability compared to the more consistent performance of the TCSM-E2E and TCSM-EVE algorithms.

Exp-6: Scalability with different data temporal graphs. We conducted experiments to assess the scalability of our algorithms by varying $|\mathcal{E}|$ in the original graph, generating four subgraphs for each dataset, and comparing the running times of all algorithms on these subgraphs. The results for the large graphs UB and SU dataset are shown in Figure 17, with consistent findings across other datasets. As $|\mathcal{E}|$ changes, the runtimes of TCSM-V2V, TCSM-E2E, and TCSM-EVE increase smoothly, whereas the runtime of RI-DS fluctuates more sharply. Moreover, across all parameter settings, TCSM-V2V, TCSM-E2E, and TCSM-EVE are significantly faster than RI-DS, reaffirming the findings from Exp-2 again.

Exp-7: Memory usages. Figure 18 compares memory usage across all algorithms on multiple datasets. The results indicate that the memory usage of the RI-DS algorithm remains relatively consistent across all datasets. For smaller datasets, our proposed algorithms use less memory, whereas on larger datasets, their memory usage exceeds that of the baseline but stays within an acceptable range. The memory usage of the TCSM-E2E and TCSM-EVE algorithms is similar and slightly higher than that of the TCSM-V2V algorithm.

Exp-8: The effect of queries q with vary $|\mathcal{L}_q|$. Figure 19 illustrates the runtime performance of all algorithms when querying a dataset with varying label types and quantities.

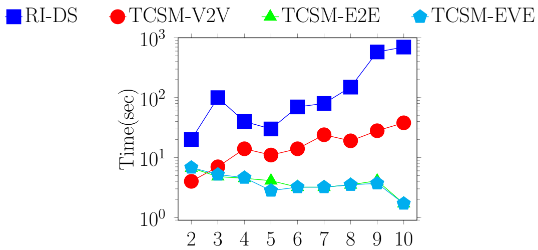
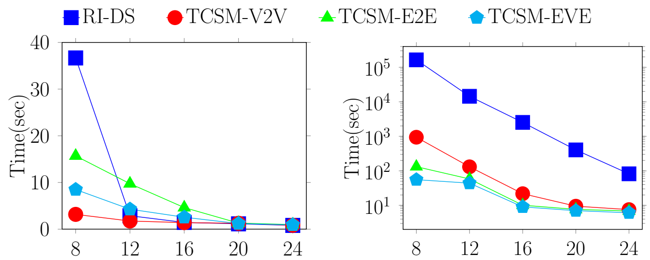


Fig. 19: Runtime with vary $|\mathcal{L}_q|$ on UB.



(a) Vary $|\mathcal{L}|$ on CM

(b) Vary $|\mathcal{L}|$ on UB

Fig. 20: Runtime with $\mathcal{G}$ of different number of labels.

The graph demonstrates how label diversity affects runtime efficiency. As the number of labels increases, the RI-DS and TCSM-V2V algorithms exhibit a general upward trend in runtime, while the TCSM-E2E and TCSM-EVE algorithms display a decreasing trend with relatively minor performance differences between them. Notably, RI-DS performs worse than the other three algorithms, reaching a local maximum in runtime when the label count is three.

Exp-9: The effect of data temporal graph $\mathbb{G}$ with vary $|\mathcal{L}|$. We generated five data graphs, each with a different number of distinct labels: 8, 12, 16, 20, and 24. Figure 20 illustrates the performance of all algorithms on these synthetic datasets as the number of labels ($|\mathcal{L}|$) increases. The results show that runtimes generally decrease and stabilize around 20 labels, suggesting that greater label diversity helps reduce the search space, thereby speeding up the matching process. Our algorithms consistently outperform RI-DS, likely due to their more sophisticated handling of labels and graph structures. For smaller datasets, TCSM-V2V demonstrates the best performance, indicating that its vertex-focused approach is particularly effective when dealing with fewer data vertices. Conversely, for larger datasets, TCSM-E2E and TCSM-EVE prove to be more efficient, as they better manage increased connectivity and complex edge data garph.

Exp-10: Observations on Failed Enumeration. We designed experiments to observe the total occurrences of failed enumerations and the specific layer in the matching tree where the first failed enumeration occurred, which can serve as an indicator of pruning efficiency. Figure 21 compares the occurrences of failed enumeration and the layer of first failed enumeration across all algorithms on UB, with consistent patterns observed on other datasets. The results indicate that the number of failed

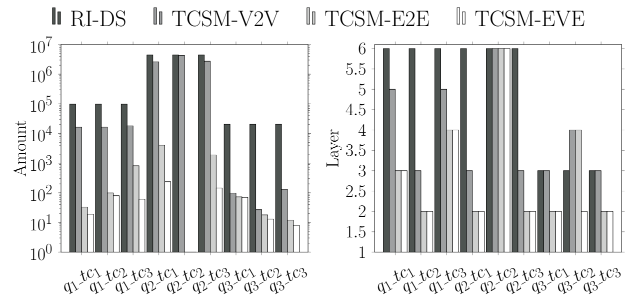


Fig. 21: Observations on failed enumeration in the UB.

enumerations in edge-based matching is lower than that in vertex-based matching, and the first failed enumeration occurs at a shallower layer in the matching tree. This suggests that pruning at higher levels of the matching tree reduces the frequency of failed enumerations, thereby enhancing matching efficiency. Additionally, the TCSM-EVE algorithm exhibits slightly fewer failed enumerations compared to TCSM-E2E, reflecting a higher level of algorithmic efficiency.

VI. Related Work

A. Subgraph isomorphism on static graphs

Filtering-Based Methods. Several filtering methods focus on local features. For instance, label-and-degree filtering [22]–[26] considers a data vertex v a candidate for query vertex u if v shares the same label as u and has a degree at least as high as u. Neighborhood label frequency filtering [14] checks if a candidate has sufficient neighbors with matching labels compared to u. Recent approaches combine these methods and incorporate pseudo-matching on nearby vertices or utilize a spanning tree or *DAG* from the query graph [1], [27]–[30].

Ordering-Based Methods. The size of the search space depends heavily on the order in which the query vertices are matched. This is because the target of the query vertex u must be selected from data vertices that are adjacent to the targets of all previously matched neighbors of u. Several efforts have been made to generate an optimal matching order by first determining the target for the query vertex with the fewest candidate vertices, thus keeping the search space for the remaining query vertices small [1], [30]–[33]. However, there is no universal method for generating an optimal ordering for arbitrary query graphs and data graphs [2], [34], [35]. Therefore, it is crucial to minimize the number of candidates.

Enumeration-Based Methods. These algorithms typically adopt a recursive enumeration procedure to find all matches and can be categorized into three types. The first, known as Direct Enumeration, directly explores the data graph to find all results, exemplified by *QuickSI* [36], *RI* [37], and *VF2++* [38]. The second type, Indexing Enumeration, constructs indexes on the data graph and uses these indices to answer queries, as seen in *GADDI* [39], *SUFF* [40], *HUGE* [41], *Circinus* [42] and *SGMatch* [43]. The third type, Preprocessing Enumeration, generates candidate vertex sets for each query at runtime and evaluates the query based on these sets. This method is widely used in recent database community algorithms, such as *GraphQL* [44], *TurboISO* [45], and *CECI* [29].

B. Subgraph isomorphism on temporal graphs

While subgraph isomorphism in static graphs is well-studied, the temporal variant has received less attention, with only a few algorithms addressing the Temporal Subgraph Isomorphism (TSI) problem [1], [10], [46]–[52]. For instance, [47] sorts edges by their time series before matching, while others [10], [46], [48], [49] focus on edge-by-edge matching under global time constraints. Earlier research [53] employed a vertex-by-vertex approach similar to the RI algorithm [37]. Additional studies explore temporal graph pattern matching using database query techniques [54]–[56]. Related to TSI is the identification of temporal motifs, which are small, recurrent subgraphs characterized by k vertices or m edges [57], [58]. The definition of a temporal motif, as introduced in [59], [60], involves matching a small query pattern with edges that follow a specific timestamp order, although these studies mainly focus on counting motifs of 2 or 3 edges. Other refinements include constraining timestamp offsets [61] and finding motifs with flow constraints [62]. Additionally, time-constrained graph pattern matching has been studied through graph simulation techniques [49], [63]–[65], highlighting different approaches to temporal graph challenges.

VII. CONCLUSION

We address the problem of Temporal-Constraint Subgraph Matching (*TCSM*), which is inherently *NP-hard*. Traditional techniques, such as candidate filtering, often struggle with the complexities of $TCSM$. To overcome these challenges, we developed three novel algorithms: TCSM-V2V, which utilizes vertex-to-vertex expansion and leverages temporal constraints to minimize duplicate matches; TCSM-E2E, which applies edge-to-edge expansion, significantly reducing matching time by minimizing vertex permutations; and TCSM-EVE, which adopts an interactive edge-vertex-edge expansion strategy, eliminating both vertex and edge permutations to further reduce duplicate matches. Extensive experiments on seven real-world temporal datasets which popularly used demonstrate that our algorithms consistently outperform existing methods, achieving substantial reductions in matching time.

REFERENCES

[1] S. Sun and Q. Luo, "Subgraph matching with effective matching order and indexing," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 1, pp. 491–505, 2022.

[2] S. Shixuan and L. Qiong, "In-memory subgraph matching: An in-depth study," in *SIGMOD*, 2020, pp. 1083–1098.

[3] J. Arai, Y. Fujiwara, and M. Onizuka, "Gup: Fast subgraph matching by guard-based pruning," *Proc. ACM Manag. Data*, vol. 1, no. 2, pp. 167:1–167:26, 2023.

[4] T. Jin, B. Li, Y. Li, Q. Zhou, Q. Ma, Y. Zhao, H. Chen, and J. Cheng, "Circinus: Fast redundancy-reduced subgraph matching," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 12:1–12:26, 2023.

[5] V. Koibichuk, N. Ostrovska, F. Kashiyeva, and A. Kwilinski, "Innovation technology and cyber frauds risks of neobanks: gravity model analysis," *Marketing i menedžment innovacij*, no. 1, pp. 253–265, 2021.

[6] J. M. Karpoff, "The future of financial fraud," *Journal of Corporate Finance*, vol. 66, p. 101694, 2021.

[7] P. O. Shoetan and B. T. Familoni, "Transforming fintech fraud detection with advanced artificial intelligence algorithms," *Finance & Accounting Research Journal*, vol. 6, no. 4, pp. 602–625, 2024.

[8] S. Gurukar, S. Ranu, and B. Ravindran, "COMMIT: A scalable approach to mining communication motifs from dynamic networks," in *SIGMOD*, pp. 475–489.

[9] S. Choudhury, L. B. Holder, G. C. Jr., K. Agarwal, and J. Feo, "A selectivity based approach to continuous pattern detection in streaming graphs," in *EDBT*, 2015, pp. 157–168.

[10] K. Semertzidis and E. Pitoura, "A hybrid approach to temporal pattern matching," in *ASONAM*, 2020, pp. 384–388.

[11] I. W. Widnyana and S. R. Widyawati, "Role of forensic accounting in the diamond model relationship to detect the financial statement fraud," *International Journal of Research in Business and Social Science (2147-4478)*, vol. 11, no. 6, pp. 402–409, 2022.

[12] L. Lai, Z. Qing, Z. Yang, X. Jin, Z. Lai, R. Wang, K. Hao, X. Lin, L. Qin, W. Zhang, Y. Zhang, Z. Qian, and J. Zhou, "Distributed subgraph matching on timely dataflow," *Proc. VLDB Endow.*, vol. 12, no. 10, pp. 1099–1112, 2019.

[13] X. Leng, G. Zeng, H. Qin, L. Lin, R.-H. Li, and G. Wang, "On temporal-constraint subgraph matching," 2025, full version available at: https://github.com/xiaoyu-ll/TSI/blob/master/tsi_main_fullversion.pdf.

[14] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *SIGMOD*, 2016, pp. 1199–1214.

[15] F. Li, Z. Zou, and J. Li, "Durable subgraph matching on temporal graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 5, pp. 4713–4726, 2023.

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

[17] K. Kim, I. Seo, W. Han, J. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong, "Turboflux: A fast continuous subgraph matching system for streaming graph data," in *SIGMOD*, 2018, pp. 411–426.

[18] C. Kankanamge, S. Sahu, A. Mhedhbi, J. Chen, and S. Salihoglu, "Graphflow: An active graph database," in *SIGMOD*, 2017, pp. 1695–1698.

[19] S. Choudhury, L. B. Holder, G. C. Jr., K. Agarwal, and J. Feo, "A selectivity based approach to continuous pattern detection in streaming graphs," in *EDBT*, 2015, pp. 157–168.

[20] M. Idris, M. Ugarte, S. Vansummeren, H. Voigt, and W. Lehner, "General dynamic yannakakis: conjunctive queries with theta joins under updates," *VLDB J.*, vol. 29, no. 2-3, pp. 619–653, 2020.

[21] X. Sun, S. Sun, Q. Luo, and B. He, "An in-depth study of continuous subgraph matching," *Proc. VLDB Endow.*, vol. 15, no. 7, pp. 1403–1416.

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

[23] H. Kim, J. Lee, S. S. Bhowmick, W. Han, J. Lee, S. Ko, and M. H. A. Jarrah, "DUALSIM: parallel subgraph enumeration in a massive graph on a single machine," in *SIGMOD*, 2016, pp. 1231–1245.

[24] L. Lai, L. Qin, X. Lin, and L. Chang, "Scalable subgraph enumeration in mapreduce," *Proc. VLDB Endow.*, vol. 8, no. 10, pp. 974–985, 2015.

[25] F. N. Afrati, D. Fotakis, and J. D. Ullman, "Enumerating subgraph instances using map-reduce," in *ICDE*, 2013, pp. 62–73.

[26] L. Lai, L. Qin, X. Lin, Y. Zhang, and L. Chang, "Scalable distributed subgraph enumeration," *Proc. VLDB Endow.*, vol. 10, no. 3, pp. 217–228, 2016.
[27] H. Kim, Y. Choi, K. Park, X. Lin, S. Hong, and W. Han, "Fast subgraph query processing and subgraph matching via static and dynamic equivalences," *VLDB J.*, vol. 32, no. 2, pp. 343–368, 2023.
[28] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K. Tan, "Gpu-accelerated subgraph enumeration on partitioned graphs," in *SIGMOD*, 2020, pp. 1067–1082.
[29] B. Bhattarai, H. Liu, and H. H. Huang, "CECI: compact embedding cluster index for scalable subgraph matching," in *SIGMOD*, 2019, pp. 1447–1462.
[30] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *SIGMOD*, 2016, pp. 1199–1214.
[31] M. Han, H. Kim, G. Gu, K. Park, and W. Han, "Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together," in *SIGMOD*, 2019, pp. 1429–1446.
[32] V. Bonnici and R. Giugno, "On the variable ordering in subgraph isomorphism algorithms," *IEEE ACM Trans. Comput. Biol. Bioinform.*, vol. 14, no. 1, pp. 193–203, 2017.
[33] A. Mhedhbi and S. Salihoglu, "Optimizing subgraph queries by combining binary and worst-case optimal joins," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1692–1704, 2019.
[34] S. Bouhenni, S. Yahiaoui, N. Nouali-Taboudjemat, and H. Kheddouci, "A survey on distributed graph pattern matching in massive graphs," *ACM Comput. Surv.*, vol. 54, no. 2, pp. 36:1–36:35, 2022.
[35] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu, "Triplebit: a fast and compact system for large scale RDF data," *Proc. VLDB Endow.*, vol. 6, no. 7, pp. 517–528, 2013.
[36] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 364–375, 2008.
[37] V. Bonnici, R. Giugno, A. Pulvirenti, D. E. Shasha, and A. Ferro, "A subgraph isomorphism algorithm and its application to biochemical data," *BMC Bioinform.*, vol. 14, no. S-7, p. S13, 2013.
[38] A. Jüttner and P. Madarasi, "VF2++ - an improved subgraph isomorphism algorithm," *Discret. Appl. Math.*, vol. 242, pp. 69–81, 2018.
[39] S. Zhang, S. Li, and J. Yang, "GADDI: distance index based subgraph matching in biological networks," in *EDBT*, 2009, pp. 192–203.
[40] X. Jian, Z. Li, and L. Chen, "SUFF: accelerating subgraph matching with historical data," *Proc. VLDB Endow.*, vol. 16, no. 7, pp. 1699–1711, 2023.
[41] Z. Yang, L. Lai, X. Lin, K. Hao, and W. Zhang, "HUGE: an efficient and scalable subgraph enumeration system," in *SIGMOD*, 2021, pp. 2049–2062.
[42] T. Jin, B. Li, Y. Li, Q. Zhou, Q. Ma, Y. Zhao, H. Chen, and J. Cheng, "Circinus: Fast redundancy-reduced subgraph matching," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 12:1–12:26, 2023.
[43] C. R. Rivero and H. M. Jamil, "Efficient and scalable labeled subgraph matching using sgmatch," *Knowl. Inf. Syst.*, vol. 51, no. 1, pp. 61–87, 2017.
[44] H. He and A. K. Singh, "Graphs-at-a-time: query language and access methods for graph databases," in *SIGMOD*, 2008, pp. 405–418.
[45] W. Han, J. Lee, and J. Lee, "Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *SIGMOD*, 2013, pp. 337–348.
[46] G. Micale, G. Locicero, A. Pulvirenti, and A. Ferro, "Temporalri: subgraph isomorphism in temporal networks with multiple contacts," *Appl. Netw. Sci.*, vol. 6, no. 1, p. 55, 2021.
[47] P. Mackey, K. Porterfield, E. Fitzhenry, S. Choudhury, and G. C. Jr., "A chronological edge-driven approach to temporal subgraph isomorphism," in *IEEE*, 2018, pp. 3972–3979.
[48] K. Kim, I. Seo, W. Han, J. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong, "Turboflux: A fast continuous subgraph matching system for streaming graph data," in *SIGMOD*, 2018, pp. 411–426.
[49] 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.
[50] F. Li and Z. Zou, "Subgraph matching on temporal graphs," *Inf. Sci.*, vol. 578, pp. 539–558, 2021.
[51] F. Li, Z. Zou, J. Li, X. Yang, and B. Wang, "Evolving subgraph matching on temporal graphs," *Knowl. Based Syst.*, vol. 258, p. 109961, 2022.
[52] N. Masuda and P. Holme, "Small inter-event times govern epidemic spreading on networks," *Physical Review Research*, vol. 2, no. 2, p. 023163, 2020.
[53] G. Locicero, G. Micale, A. Pulvirenti, and A. Ferro, "Temporalri: A subgraph isomorphism algorithm for temporal networks," in *Complex Networks*, 2020, pp. 675–687.
[54] A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels, F. Murlak, S. Plantikow, P. Selmer, O. van Rest, H. Voigt, D. Vrgoc, M. Wu, and F. Zemke, "Graph pattern matching in GQL and SQL/PGQ," in *SIGMOD*, 2022, pp. 2246–2258.
[55] A. Aghasadeghi, J. V. den Bussche, and J. Stoyanovich, "Temporal graph patterns by timed automata," *VLDB J.*, vol. 33, no. 1, pp. 25–47, 2024.
[56] J. Giavitto and J. Echeveste, "Real-time matching of antescofo temporal patterns," in *International Symposium on Principles and Practice of Declarative Programming*, 2014, pp. 93–104.
[57] P. Liu, A. R. Benson, and M. Charikar, "Sampling methods for counting temporal motifs," in *WSDM*, 2019, pp. 294–302.
[58] 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.
[59] A. Paranjape, A. R. Benson, and J. Leskovec, "Motifs in temporal networks," *CoRR*, vol. abs/1612.09259, 2016.
[60] S. Sun, X. Sun, Y. Che, Q. Luo, and B. He, "Rapidmatch: A holistic approach to subgraph query processing," *Proceedings of the VLDB Endowment*, vol. 14, no. 2, pp. 176–188, 2020.
[61] A. Züfle, M. Renz, T. Emrich, and M. Franzke, "Pattern search in temporal social networks," in *EDBT*, 2018, pp. 289–300.
[62] C. Kosyfaki, N. Mamoulis, E. Pitoura, and P. Tsaparas, "Flow motifs in interaction networks," in *EDBT*, 2019, pp. 241–252.
[63] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke, "Computing simulations on finite and infinite graphs," in *Annual Symposium on Foundations of Computer Science*, 1995, pp. 453–462.
[64] S. Gillani, G. Picard, and F. Laforest, "Continuous graph pattern matching over knowledge graph streams," in *DEBS*, 2016, pp. 214–225.
[65] T. Zhang, Y. Gao, L. Qiu, L. Chen, Q. Linghu, and S. Pu, "Distributed time-respecting flow graph pattern matching on temporal graphs," *World Wide Web*, vol. 23, no. 1, pp. 609–630, 2020.