Proof of THEOREM 1: We can present an algorithm that verifies a solution in polynomial time:

1) Iterate through all possible subgraphs in the temporal graph, considering all combinations of vertices and edges.
2) For each potential subgraph, use a graph isomorphism algorithm to check if there exists a one-to-one correspondence between vertices and edges, and if their timestamps match.
3) If a subgraph is found to be isomorphic to the pattern graph, add it to the set of candidate solutions.
4) Finally, verify if the set of candidate solutions satisfies all constraints of the subgraph matching problem, such as the size of the pattern graph.

Proof of THEOREM 2: Initializing `tsup[u]` for each $u \in V_q$ takes $O(|V_q|)$. Iterating over each temporal-constraint $tc \in TC$ and updating `tsup` takes $O(|TC|)$. 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 $tc \in TC$ takes $O(|TC|)$. Therefore, the total time complexity is $O(|V_q|^2 + |V_q| \cdot d_{\max} + |TC|)$.

The arrays `tsup`, `TO`, `PD`, `FN`, and `TP` each require $O(|V_q|)$ space. Temporary variables used in loops and recursive calls also take $O(|V_q|)$ space. Hence, the total space complexity is $O(|V_q|)$.

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 $\mathbb{TCQ}$ graph (Line 4) using the $\mathbb{TCQ}$ algorithm has a time complexity of $O(|V_q|^2 + |V_q| \cdot d_{\max} + |TC|)$. 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 $O(|TC|)$ per time constraint check. Therefore, the overall time complexity is $O(2^{|V_q|} \times (|V| + d_{\max} + |TC|))$.

Arrays `tsup`, $\mathcal{TO}$, `PD`, `FN`, and `TP` each require $O(|V_q|)$ space. 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 $O(|V_q|)$.

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 $O(|E_q| + |TC|)$. The main loop (Lines 12-25), which involves selecting edges based on support and updating data structures, iterates $O(|E_q|)$ times with each iteration involving $O(|E|)$ operations for finding the maximum support and minimum operations, leading to $O(|E_q| \cdot |E|)$. Therefore, the total time complexity is $O(|V_q|^2 \cdot d_{\max}^2 + |E_q| \cdot |E| + |TC|)$. Storage for arrays `support`, `TO`, `PD`, `FE`, and `TP` requires $O(|E_q|)$ space. The temporary data structures and variables used during processing also require $O(|E_q|)$ space. Hence, the total space complexity is $O(|E_q|)$.

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 $\mathbb{TCQ}$+ graph (Line 4) has a time complexity of $O(|E_q|^2 + |E_q| \cdot d_{\max}^2 + |TC|)$. 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 $O(2^{|E_q|} \times (|E| + d_{\max} + |TC|))$.

Arrays `support`, `TO`, `PD`, `FE`, and `TP` each require $O(|E_q|)$ space. 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 $O(|E_q|)$.

Proof of THEOREM 6: The initialization and generating of the $\mathbb{TCQ}$+ graph (omitted details) involve operations similar to those in previous algorithms, which typically take $O(|E_q|^2 + |E_q| \cdot d_{\max}^2 + |TC|)$. 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 $O(2^{|E_q|} \times (|E| + d_{\max} + |TC|))$.

Arrays `support`, `TO`, `PD`, `FE`, and `TP` each require $O(|E_q|)$ space. 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 $O(|E_q|)$.

On Temporal-Constraint Subgraph Matching

Xiaoyu Leng
Beijing Institute of Technology, China
3120230929@bit.edu.cn

Guang Zeng
Ant Group
zengguang_77@qq.com

Hongchao Qin
Beijing Institute of Technology, China
hcqin@bit.edu.cn

Longlong Lin
Southwest University, China
longlonglin@swu.edu.cn

Rong-Hua Li
Beijing Institute of Technology, China
rhli@bit.edu.cn

Guoren Wang
Beijing Institute of Technology, China
wanggr@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 communication networks. However, the problem of temporal-constraint subgraph matching is NP-hard. Furthermore, because each temporal sequence 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 with a temporal constraint tc on the 8-million-scale WT dataset. 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 with 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.

PVLDB Reference Format:
Xiaoyu Leng, Guang Zeng, Hongchao Qin, Longlong Lin, Rong-Hua Li, and Guoren Wang. On Temporal-Constraint Subgraph Matching. PVLDB, 17(1): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Subgraph matching is a fundamental problem in graph theory. Given a data graph d and a query graph q, the objective is to identify all subgraphs of d that are isomorphic to q. While most recent research focuses on matching subgraphs in static graphs [3, 23, 54, 56], real-world subgraph patterns are often associated with temporal information. The following scenarios highlight the importance of temporal subgraph matching in practical applications.

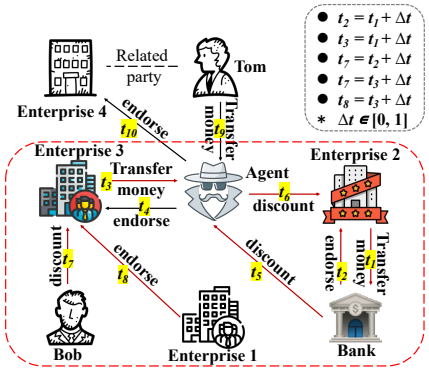


Figure 1: A temporal bill circulation network in Ant Group.

- 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 [27, 32, 55]. 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 [10, 16, 52]. Analyzing these patterns enables telecommunication companies to improve fraud detection and enhance network security.

A more specific example is illustrated in Figure 1, which shows 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, agent, or individual, and each edge signifies a transaction, discount, or endorsement behavior between vertices. Each edge is associated with a set of timestamps that indicate when these interactions occurred, with t_i ($i \in N$) following an arithmetic sequence with a common difference of one day. A significant risk in the financial bill circulation network is the activity of bill intermediaries. These intermediaries purchase acceptance bills from companies at a discounted price using cash and then transfer the bills to other enterprises or banks to earn an interest margin. Such intermediary activities destabilize the transaction

Proceedings of the VLDB Endowment, Vol. 17, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

system, increasing the risk of bill fraud, money laundering, and other criminal behaviors. By using temporal-constraint subgraph matching techniques, it's possible to more accurately identify bill intermediaries. For example, the subgraph model highlighted by the red dashed circle in Figure 1 represents a typical risk intermediary model in financial bill transactions[27, 32, 55, 61]. The suspected intermediary at the center of the red circle is shown buying acceptance bills with cash and quickly transferring them. **Unlike traditional subgraph patterns, the temporal constraints in the upper right corner show that each transaction is temporally linked to the previous one, satisfying the condition that the transactions occur within a time window of Δt.** Therefore, temporal-constraint subgraph matching can more accurately detect 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 3). 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 [36, 54, 56]. 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 address this challenge, we have designed efficient algorithms to solve the *TCSM* problem and conducted extensive experiments to verify 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 into a Temporal-Constraint Query Graph ($\mathbb{TCQ}$), which leverages temporal constraints to prune final vertexs and reduces unnecessary duplicate matches.
- We present the TCSM-E2E algorithm, which employs an edge-to-edge expansion approach. This algorithm merges the query and temporal-constraints into the $\mathbb{TCQ}$+ graph. This method minimizes the vertex permutation process found in the TCSM-V2V algorithm, significantly reducing matching time.
- We propose the TCSM-EVE algorithm, which utilizes an interactive edge-vertex-edge expansion strategy. This approach generates results **without requiring vertex and edge permutations**, thereby 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, **the** TCSM-EVE **algorithm completes the task in just 84 seconds**. We also analyze the performance of these algorithms concerning failed backtracking metrics, showing that our algorithm accelerates matching by pruning non-conforming matches early.
- **The code are available at https://github.com/xiaoyu-ll/TSI**.

2 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 $f : V_q \rightarrow V$ 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 ($\mathcal{G}$)). *A simple directed data temporal graph can be represented by $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}, \mathcal{T})$, 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.

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

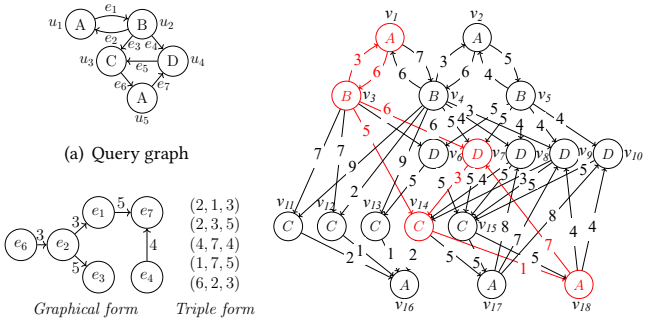
Assuming that set E_q in G_q adheres to a specific order $\{e_1, e_2 ... e_{|\mathcal{E}|}\}$, we define the temporal-constraint graph that adheres to the temporal constraints as follows.

Definition 3 (Temporal-Constraints ($\mathcal{TC}$)). *The temporal-constraint $\mathcal{TC}$ 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. $0 \leq e_j.t - e_i.t \leq k$.*

Based on Definition 3, we can observe that the $\mathcal{TC}$ 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 $\mathcal{TC} = \{(i, j, k)\}$, and a data temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}, \mathcal{T})$, a temporal-constraint subgraph matching (abbreviated as TCSM) from G_q to $\mathcal{G}$ under the temporal-constraint $\mathcal{TC}$ is an injective function $f : E_q \rightarrow \mathcal{E}$ which satisfies:*

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



(a) Query graph

(b) Temporal-Constraints

(c) Data Temporal Graph

Figure 2: A running example.

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

Example 1. *Given G_q, $\mathcal{TC}$ and $\mathcal{G}$ as shown in Figure 2, we identify a temporal-constraint subgraph matching from G_q to $\mathcal{G}$, with the matching subgraphs highlighted in red in Figure 2(c). When temporal information is not considered, the red-highlighted subgraph satisfies the structural requirements of graph 2(a). Next, we incorporate the temporal constraints from graph 2(b), which defines five distinct sets of constraints that are evaluated individually. For instance, the temporal constraint tc_2 = (2,3,5) dictates that the event on the edge from the vertex labeled 'B' to the vertex labeled 'A' must occur before the event on the edge from the vertex labeled 'B' to the vertex labeled 'C', and this must happen within a time interval of 5. In the red-highlighted subgraph, we verify whether the event on the edge v_3 to v_1 happens before the event on the edge v_3 to v_{14}, and if the time interval between them is within 5. Observing that the event on the edge v_3 to v_1 occurs at time 3, which precedes the event on the edge v_3 to v_{14} at time 5, and that the interval between them is 2 (which is within 5), we confirm that this temporal constraint is satisfied. The same holds true for the other sets of constraints. In this example, there is only one match that satisfies both the query graph and the temporal-constraint graph.*

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

The proof of Theorem 1 can be demonstrated by constructing 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 known NP-hard problem, this proves the theorem.

Challenges: As mentioned above, the *TCSM* problem is NP-hard. The traditional Subgraph Matching problems are typically accelerated through techniques like filtering and verification, optimizing the matching order, and using decomposition and combination indexes (Section 6). However, these methods are challenging to apply to the *TCSM*. **First**, the temporal aspect of the Temporal-Constraint Subgraph Matching problem introduces additional complexity beyond static subgraph matching. It requires not only finding structural matches in temporal graphs but also satisfying temporal constraints, which means any filtering and verification framework must account for these temporal conditions. **Second**, due to the inclusion of temporal constraints, the matching order must adhere to temporal requirements. This introduces a combinatorial factor into the matching sequence, significantly increasing the computational burden. **Third**, if a decomposition and indexing approach is taken, the added combinatorial factor in the matching order makes the index size unmanageable, rendering this optimization method impractical. In summary, existing optimization techniques are difficult to directly apply for the *TCSM*.

3 BASIC ALGORITHM FOR *TCSM*

Traditional subgraph isomorphism methods generally follow the basic approach of starting from a vertex in the query graph, finding a matching order, and then matching along the vertices of the query graph on the data graph while pruning during the matching process. Based on this principle, we propose an algorithm for temporal-constraint subgraph matching by Vertex-To-Vertex, abbreviated as TCSM-V2V. Unlike the traditional methods, we need to consider the following observations while matching in the temporal graphs.

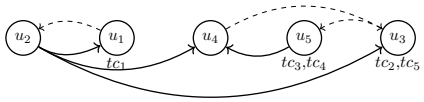
O1. Matching Orders. Traditional methods prioritize matching orders by focusing on high-degree vertices in the query graph and minimizing candidate set sizes. However, when temporal constraints are introduced, the matching order must also account for the temporal aspects of the graph. Therefore, a new ordering rule is required—one that simultaneously considers the structure of the query graph, the size of candidate sets, and the temporal constraints.

O2. Candidate Filtering. Traditional methods, like the NLDF (Neighbor Label Degree Filter), are used to identify candidate sets for each query vertex, often applying special filters to remove unsuitable vertices. Despite this, the candidate sets can remain large. After the initial vertex is matched, the number of valid candidates under structural constraints is typically much smaller than the original set. To address this, we propose a method for dynamically generating candidate sets based on the currently matched vertices, thereby reducing unnecessary computational overhead.

O3. Validity Checking. When a candidate data vertex for a query vertex appears, its validity can be checked in two ways: by verifying the edges between the query vertex and its matched neighbors, and by using matrix calculations of the state space method. Both are computationally intensive, and the latter is only suitable for small data graphs. We aim to find a less computationally intensive and faster pruning method to validate candidate data vertices.

O4. Temporal-Constraint Checking. To effectively find all subgraphs in the data graph that satisfy both the query graph and the temporal-constraint graph, the algorithm must continuously check partial matches against both graphs. We propose a novel approach that combines the query graph and the temporal-constraint graph into a single, unified structure. This integration aims to reduce the time spent on cross-referencing between the two graphs.

Following the above four observations, we propose the concept of temporal-constraint query graph, denoted as $\mathbb{TCQ}$, which can combine the query graph and temporal-constraints (Definitions 2, 3) into one graph. The $\mathbb{TCQ}$ not only contains the structure information of the query graph and the structure information of the *temporal-constraint* graph, but also the matching order of the vertices, and the way in which each vertex candidate set is generated.



$\mathcal{TO} = \{1 : u_2, 2 : u_1, 3 : u_4, 4 : u_5, 5 : u_3\}$, $\mathcal{FN} = \{u_1$: $\{u_2\}$, u_4: $\emptyset$, u_5: $\emptyset$, u_3: $\{u_4, u_5\}\}$

$\mathcal{PD} = \{u_1 : u_2, u_4 : u_2, u_5 : u_4, u_3 : u_2\}$, $\mathcal{TP} = \{tc_1$: u_1, tc_2: u_3, tc_3: u_5, tc_4: u_5, tc_5: $u_3\}$

Figure 3: Temporal-Constraint Query Graph $\mathbb{TCQ}$

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

input : a temporal-constraint $\mathcal{TC}$, a query graph G_q and a data temporal graph $\mathcal{G}$
output: four hash tables $\mathcal{TO}$, $\mathcal{PD}$, $\mathcal{FN}$, $\mathcal{TP}$ of $\mathbb{TCQ}$.
// compute $tsup$ for each query vertex
1 **for** each $u \in V_q$ **do** $tsup[u] = 0$;
2 **for** *each* $tc = (i, j, k) \in \mathcal{TC}$ **do**
3 $tsup[e_i.u]$++; $tsup[e_i.v]$++; $tsup[e_j.u]$++; $tsup[e_j.v]$++;
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 V_q^μ = {$u \in V_q$ and not in $\mathcal{TO}$};
8 $\mathcal{TO}[\mu] \leftarrow \arg\max_{u \in V_q^\mu}\{|N_\mu(u)|\}$;
9 $\mathcal{PD}[\mu] = \arg\min_{u' \in N_\mu(\mathcal{TO}[\mu])}\{\mathcal{TO}'[u']\}$;
10 $\mathcal{FN}(\mu) \leftarrow N_\mu(u)$;
11 μ ++;
12 **for** *each* $tc \in \mathcal{TC}$ **do**
13 $\mathcal{TP}[tc] = \arg\max_{u \in S}\{\mathcal{TO}'[u]\}$;
14 **return** ($\mathcal{TO}$, $\mathcal{PD}$, $\mathcal{FN}$, $\mathcal{TP}$);

3.1 The construction of $\mathbb{TCQ}$

The $\mathbb{TCQ}$ is constructed using four key hash tables: Temporal Order ($\mathcal{TO}$), Prec Dictionary ($\mathcal{PD}$), Forward Neighbor ($\mathcal{FN}$), and Time Peeling ($\mathcal{TP}$). As shown in Figure 3, vertices are arranged from left to right, indicating the matching order. Solid lines between vertices represent predecessor relationships, where the preceding vertex is the *prec* of the following vertex. Dotted lines indicate forward relationships, where the preceding vertex is part of the *forward vertex* set of the subsequent vertex. Arrows show the direction of the corresponding edges between vertices in the query graph. Labels 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, with further details provided below.

In the following, we describe the process of constructing the four associated hash tables: Temporal Order ($\mathcal{TO}$), Prec Dictionary ($\mathcal{PD}$), Forward Node ($\mathcal{FN}$), and Time Peeling ($\mathcal{TP}$).

Temporal Order. In temporal graph matching, the matching order must consider both structural constraints in the query graph and temporal constraint graph. The vertex ordering method is based on *Temporal-Constraint Support*, prioritizing query vertices that have the highest support and strong connections with vertices already placed in the sequence. First, *Temporal-Constraint Support* is calculated for each query vertex (Algorithm 1, Lines 1-3). The vertex with the highest support 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 previously ordered vertices (Algorithm 1, Lines 7-8). In the event of a tie, the vertex with the maximum *temporal-constraint support* is chosen. Should there still be a tie, the vertex is chosen at random.

Definition 5 (Temporal-Constraint Support ($tsup$)). *Given a tempogstraint TC = {(i, j, k)}, and a query graph $G_q = (V_q, E_q, \mathcal{L}_q)$, the* Temporal-Constraint Support *of a vertex $u \in V_q$ is the total sum of the in-degrees and out-degrees of edges that contain u within the temporal constraint graph.*

Prec Dictionary. The *Prec Dictionary* ($\mathcal{PD}$) stores the predecessor of each query vertex, defined as the neighbor vertex that appears earliest in the matching sequence (Algorithm 1, Line 9). Beyond the initial vertex, candidate vertices for matching are generated based on the partially matched subgraph at each stage. Since the $\mathbb{TCQ}$ graph stores structural relationships between vertices and their *prec*, candidate vertices can be identified by examining the neighbors of already-matched predecessors, ensuring they satisfy the required structural constraints. This approach reduces the size of candidate sets, minimizing unnecessary computations. If a candidate set is empty, the matching process fails, and the algorithm backtracks.

Definition 6 (Prec Dictionary). *Given a query graph $G_q = (V_q, E_q, \mathcal{L}_q)$ and a matching order $\mathcal{TO}$, the* prec *of a query vertex u is a vertex which is a neighbor of u and appears earliest in $\mathcal{TO}$. The* prec dictionary *to maintain the relationship of query vertices and $\mathcal{TO}$, denotes as $\mathcal{PD}$, records the* prec *of each query vertex, and $\mathcal{PD}[u]$ denotes the* prec *of u. $\mathcal{TO}[1]$ has no* prec*, since $\mathcal{TO}[1]$ is the first vertex in $\mathcal{TO}$ and has no neighbor that appear earlier than it.*

Forward Neighbor. To maintain the integrity of the query graph structure, *Forward Neighbor* sets are created to track the neighboring vertices that precede the current vertex in the $\mathcal{TO}$ sequence, excluding the *prec* (Algorithm 1, Line 10). These forward neighbors are stored for each query vertex, except the initial one, within the $\mathbb{TCQ}$. This approach efficiently captures the structural relationships in the query graph while ensuring the accuracy of matched data vertices. When matching a query vertex, it is essential to verify the presence of a corresponding data edge between the matched data vertex and its candidate data vertex. If the edge is missing, the algorithm backtracks, enhancing pruning efficiency.

Definition 7 (Forward Node). *Given a matching order $\mathcal{TO}$, a query graph $G_q = (V_q, E_q, \mathcal{L}_q)$ and a vertex $u \in V_q$, a forward node of u, denoted as $\mathcal{FN}(u)$, is a neighbor vertex of u in V_q which appear earlier than u in $\mathcal{TO}$ except the* prec *of u.*

Time Peeling. Once all 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, a dictionary is created to record the vertex within the *temporal constraint* set that appears last in the $\mathcal{TO}$ sequence for each constraint (Algorithm 1, Lines 12-13). When matching this vertex, both structural compatibility and compliance with the *temporal constraint* must be verified. If the constraint is not satisfied, the match is illegal.

Definition 8 (Temporal-Constraint Dictionary). *Given a* temporal-constraint *tc = (i, j, k), and a matching order $\mathcal{TO}$, the* temporal-constraint dictionary *to maintain the relationship of tc and $\mathcal{TO}$, denotes as $\mathcal{TP}[tc]$, records the vertex in vertices set ($e_i.u$, $e_i.v$, $e_j.u$, $e_j.v$) that appear latest in $\mathcal{TO}$.*

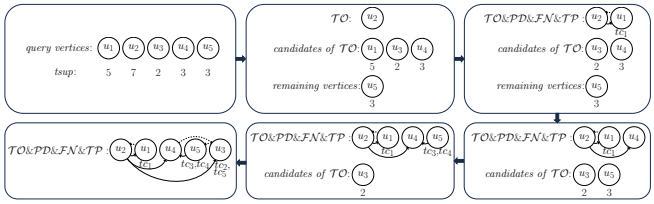


Figure 4: The construction of $\mathbb{TCQ}$.

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

The proof of Theorem 2 can get access at the full version [37].

Example 2. *When constructing a $\mathbb{TCQ}$ graph for the query graph, the* temporal-constraints *graph, and the data graph shown in Figure 2, the first step involves calculating the temporal-constraint support (tsup) for each vertex. The vertex with the highest tsup is then selected and added to $\mathcal{TO}$. Subsequently, its neighboring vertices are included in the* candidates of $\mathcal{TO}$. *Among these candidates, the vertex with the highest tsup is chosen next, and its preceding vertex in $\mathcal{TO}$ is designated as its prec, connecting them with a solid line. The neighboring vertices of this newly added vertex are also incorporated into the* candidates of $\mathcal{TO}$. *We then examine for edges between this vertex and other vertices already in $\mathcal{TO}$, except for prec, and if such edges exist, they are connected with dashed lines. Furthermore, we verify whether this vertex forms any pairs of constraint edges with other vertices in $\mathcal{TO}$, according to the* temporal constraints. *This process is iterated for the remaining vertices and edges.*

Therefore, the $\mathbb{TCQ}$ can consider the four observations such that:

1) All query vertices are put in sequence according to the structural and temporal constraints. **(Inferred from $\mathcal{TO}$)**
2) The precursor of a vertex can be directly identified from the $\mathbb{TCQ}$ graph, which aids in generating candidate data vertices for the query vertices. **(Inferred from $\mathcal{PD}$)**
3) The $\mathbb{TCQ}$ graph should encode sufficient structural information from the query graph to ensure accurate results during the matching process. **(Inferred from $\mathcal{FN}$)**
4) Once all vertices within a temporal constraint have been matched, the current partial match must be evaluated to ascertain its compliance with the temporal constraint. Thus, it is evident from the $\mathbb{TCQ}$ graph that each temporal constraint is assessed at a specific vertex. **(Inferred from $\mathcal{TP}$)**

3.2 Algorithm TCSM-V2V.

Given a temporal-constraint $\mathcal{TC}$, a query graph G_q and a data temporal graph $\mathcal{G}$, TCSM-V2V recursively expands partial match by mapping query vertices to their candidates following their order and judge their temporal information whether satisfy $\mathcal{TC}$ when all vertices in a *temporal-constraint vertices* set has been matched.

At first, we need to generate the initial candidates for each vertex in the data temporal graph $\mathcal{G}$. Given G_q and $\mathcal{G}$, the generation of initial candidates process is to obtain all possible data vertices set which can match u for each query vertex u. Inspired by the Neighborhood Label Frequency filtering [5], we design *neighbor label filter* to judge a data vertex v whether can match a query vertex u to make the candidate set as small as possible.

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

```
   input  : a temporal-constraint TC, a query graph G_q and a data temporal graph G
   output : all temporal-constraint subgraph matchings M from G_q to G under TC
   // generate initial candidate set (by def. 11)
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,FN,TP) ← 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, FN, TP, M, λ+1);
9  Procedure DFS(G,TO,PD,FN,TP,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) & the direction of edge(u_c, M[u']), (u, u') is same
                 then
16                 u.C ← u_c;
17         for each v ∈ u.C do
18             if Validate(G, TP, FN, M, u, v, λ) is true then
19                 M[u] ← v;
20                 DFS (G, TO, PD, FN, TP, M, λ+1);
21 Function Validate(G, TP, FN, M, u, v, λ):
22     for each u' ∈ FN(u) with TO'[u'] < λ 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;
```

Definition 9 (Neighbor Label Filter (NLF)). *Given a query graph $G_q = (V_q, E_q, \mathcal{L}_q)$ and a data temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}, \mathcal{T})$ and the* de-temporal graph $G = (V, E)$ *of $\mathcal{G}$. 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) *$\forall\, u' \in N(u), \exists\, v' \in N(v), L(v') = L(u')$ & the direction of edge (u', u), (v', v) is same.*

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

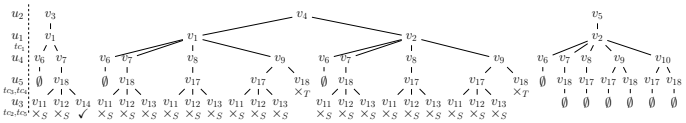


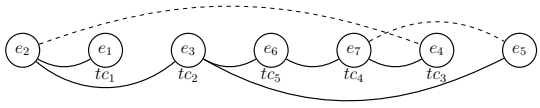
Figure 5: A running example of TCSM-V2V on Figure 2.

vertices matched to the neighbors of u. And it also needs to check whether the temporal information of the data vertices matched to the *temporal-constraint vertices* satisfy *temporal-constraint* if there exist a *temporal-constraint* which the largest order number of its *temporal-constraint vertices* is λ. If so, we expand the partial match by matching the next query vertex (Lines 19-20). Otherwise, the algorithm backtracks to match the next data candidate vertex.

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

The proof of Theorem 3 can get access at the full version [37].

Example 3. *Figure 5 illustrates the matching results using the* TCSM-V2V *algorithm on the query graph* G_q *and the data graph*$\mathcal{G}$ *under the temporal constraint* $\mathcal{TC}$*. In the figure, the symbol* ✓ *indicates that the path match is correct. The matching set* $M = \{(v_3, u_2), (v_1, u_1), (v_7, u_4), (v_{18}, u_5), (v_{14}, u_3)\}$ *satisfies the matching conditions. The symbol* ✗*(S)denotes that a candidate data vertex cannot match the query vertex due to a structural mismatch. This occurs when there is an edge between a vertex* u *and its neighbor vertex* u'*, which appears earlier in the matching order* $\mathcal{TO}$*, but no corresponding edge exists between the candidate data vertex* v *and the match of* u'*. The symbol* ✗*(T) indicates that a candidate data vertex fails to match the query vertex due to temporal inconsistencies, meaning that the temporal information along the matched data vertices does not satisfy the given temporal constraint. The symbol* $\emptyset$*signifies that there is no candidate vertex available for matching with query vertex* u*. If a candidate data vertex cannot match the query vertex or if the candidate set is empty, the algorithm backtracks. For example, when matching* u_5*, with the partial match* $M = \{(u_2, v_4), (u_1, v_1), (u_4, v_7)\}$*, the match for* u_4 *(the predecessor of* u_5*) is* v_7*. The structural relationship between* u_4 *and* u_5 *includes an edge from* u_5 *to* u_4*. Therefore, when identifying candidate vertices for* u_5*, only the neighboring vertices of* v_7 *that share an edge with* v_7 *and have the same label as* u_5 *need to be considered. This reduces the candidate vertex set for* u_5 *to* $\{v_{18}\}$*, which is significantly smaller than the initial set. Similarly, when matching* u_3*, with the partial match* $M = \{(u_2, v_4), (u_1, v_1), (u_4, v_7), (u_5, v_{18})\}$*, the candidate vertex set for* u_3 *is* $\{v_{11}, v_{12}, v_{13}\}$*. The forward neighbors for* u_3 *are* u_4 *and* u_5*. To determine if candidate vertex* v_{11} *can structurally match* u_3*, it is necessary to verify the existence of corresponding edges between* v_{11} *and the matches of* u_4 *(denoted as* v_7*) and* u_5 *(denoted as* v_{18}*). In the data graph, no edges exist from* v_{11} *to* v_7 *or from* v_{11} *to* v_{18}*. Consequently,* v_{11} *cannot structurally match* u_3*. Lastly, when matching* u_5 *with the partial match*$M = \{(u_2, v_4), (u_1, v_1), (u_4, v_9)\}$*, the candidate data vertex* v_{18} *structurally matches* u_5*. According to the temporal-constraint query graph* $\mathbb{TCQ}$*,* u_5 *is the latest vertex in the matching sequence for both temporal constraints* tc_3 *and* tc_4*. Therefore, at* u_5*, it is necessary to verify whether the interactions of all matched vertices satisfy these constraints. In the data graph, the partial match at this stage satisfies* tc_3 *but fails to satisfy* tc_4*, indicating that* v_{18} *cannot match* u_5*.*



$\mathcal{TO} = \{1{:}\ e_2, 2 : e_1, 3 : e_3, 4 : e_6, 5 : e_7, 6 : e_4, 7 : e_5\}$, $\mathcal{FE} = \{e_1{:}\ \emptyset, e_3{:}\ \emptyset, e_6{:}\ \emptyset, e_7{:}\ \emptyset, e_4{:}\ \{e_2\}, e_5{:}\ \{e_7\}\}$
$\mathcal{PD} = \{e_1 : e_2, e_3 : e_2, e_6 : e_3, e_7 : e_6, e_4 : e_7, e_5 : e_3\}$, $\mathcal{TP} = \{tc_1{:}\ e_1, tc_2{:}\ e_3, tc_3{:}\ e_4, tc_4{:}\ e_7, tc_5{:}\ e_6\}$

Figure 6: Temporal-Constraint Query Graph $\mathbb{TCQ}$+

4 ADVANED ALGORITHM FOR *TCSM*

4.1 The construction of $\mathbb{TCQ}$+

In the previous section, we introduced a vertex-to-vertex matching approach. However, due to the requirement of identifying specific graph patterns with temporal constraints within temporal subgraphs, vertex-based matching methods involve numerous time-consuming edge permutation checks. Consequently, we are exploring an edge-based matching algorithm as an alternative.

For edge-to-edge matching method, the $\mathbb{TCQ}$ graph becomes unsuitable since the basic matching unit shifts from vertices to edges. To accommodate this shift, we propose constructing a new $\mathbb{TCQ}$+ graph, which involves modifying the Temporal Order ($\mathcal{TO}$), Precursor Dictionary ($\mathcal{PD}$), and Time Peeling ($\mathcal{TP}$), while also replacing the concept of Forward Node ($\mathcal{FN}$) with Forward Edge ($\mathcal{FE}$). The construction process for $\mathbb{TCQ}$+ is detailed below.

To determine the matching order of edges, we start from Figure 2(b) and consider the following criteria: (i) the degrees of the edges in Figure 2(b), with higher-degree edges given higher priority; (ii) if two adjacent nodes e in Figure 2(b) can be linked in Figure 2(a), then these associated edges are prioritized in the matching order.

The edge-based matching order must take into account both temporal and structural constraints. Each edge to be matched should share a vertex with previously matched 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 that share a vertex and have temporal constraints, we construct a Temporal-Constraint Forest (*TCF*). This forest, composed of interconnected trees, optimizes the matching process by capitalizing on the interconnected nature of edges, thereby improving efficiency and effectiveness in handling complex graph queries.

To further optimize the matching order, edges with more temporal and structural constraints should be prioritized. We also define *temporal-constraint support*, which quantifies the extent of each edge's temporal constraints by summing its in-degrees and out-degrees in the temporal constraint graph. This metric helps manage and optimize query processing in temporally constrained graph.

Forward Edge. To maintain the integrity of the query graph, we introduce the concept of the *forward edge*. This approach minimizes the storage of query graph structural information while ensuring the validity of matched edges. Assuming that the source and target vertices of the matching edge are both vertices matched in some partial match, and the candidate match is generated by $\mathcal{PD}$, ensuring that one vertex is in the partial match, we also need to ensure that the other vertex is in the partial match. We use $\mathcal{FE}$ to check whether the other vertex is in the partial match; this vertex is the common vertex between the candidate edge and the match with $\mathcal{FE}(e)$. If the other vertex is in the partial match, then the candidate edge and $\mathcal{FE}(e)$ match must have a common vertex.

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

input : a temporal-constraint $\mathcal{TC}$, a query graph G_q and a data temporal graph $\mathcal{G}$
output: four hash tables $\mathcal{TO}$, $\mathcal{PD}$, $\mathcal{FE}$, $\mathcal{TP}$ of $\mathbb{TCQ}+$.

```
// creat TCF for temporal-constraint graph
1  ℰ ← ∅, 𝒱 ← {(N_e | e ∈ E_q)};
2  for each u ∈ V_q do
3      for each e_i ∈ u.adje do
4          for each e_j ∈ u.adje (e_i ≠ e_j) do
5              if (i, j, k) ∈ TC then
6                  ℰ ∪ {(N_{e_i}, N_{e_j})};
7                  if (𝒱, ℰ) is cyclic then
8                      ℰ \ {(N_{e_i}, N_{e_j})};
// compute support for each query edge
9  for each e ∈ E_q do support[e] = 0;
10 for each tc = (i, j, k) ∈ TC do
11     support[e_i]++; support[e_j]++;
12 μ ← 2, TO ← ∅, δ ← 0;
13 TO[1] ← arg max_{e∈E_q} {support[e]};
14 while μ ≤ |E_q| do
15     for each e', (N_{e'}, N_{TO[μ−1]}) ∈ ℰ do
16         δ ← δ + 1, PD[e'] = TO[μ − 1];
17     while δ > 0 do
18         δ ← δ - 1, TO[μ] ← arg max_{e∈N^ℰ_μ} {support[e]};
19         FE[μ] = arg min_{e∈TO, e∩TO[μ]≠TO[μ]∩PD[μ]} {TO'[e]};
20         for each e', (N_{e'}, N_{TO[μ]}) ∈ ℰ do
21             δ ← δ + 1, PD[e'] = TO[μ];
22         μ++;
23     TO[μ] ← arg max_{e∈N^E_μ} {support[e]};
24     FE[μ] = arg min_{e∈TO, e∩TO[μ]≠TO[μ]∩PD[μ]} {TO'[e]};
25     μ++;
26 for each tc ∈ TC do TP[tc] = arg max_{e∈tc} {TO'[e]};
27 return (TO, PD, FE, TP);
```

Definition 10 (Forward Edge). *Given a matching order $\mathcal{TO}$, a query graph $G_q = (V_q, E_q, \mathcal{L}_q)$ and a edge $e \in E_q$, a forward edge of e, denoted as $\mathcal{FE}(e)$, is an edge of in E_q which share common vertex with e and appear earliest than e in $\mathcal{TO}$ and this common vertex is not the common vertex of e and prec of e share.*

Algorithm 3 presents the construction of $\mathbb{TCQ}+$ graph. We first construct Temporal-Constraint Forest (Lines 1-8). Then we compute the *tsup* for each edge in query graph (Lines 9-11). The process starts by selecting the edge with the highest count of *temporal constraints* (Line 13). Subsequent edges in the same forest are selected based on their shortest path length to the initial edge, with ties again resolved by the smallest candidate set (Line 18). This method systematically organizes edges within each forest. When advancing to edges from different forests, the next edge is chosen from a forest that shares at least one vertex with the previously ordered edge and has the highest number of *temporal constraints* (Line 23). We store the *forward edge* of each query edge, except for the initial edge (Line 19 & 24) in the $\mathbb{TCQ}+$ graph. For each *temporal-constraint*, we record the edge that appears latest in the matching sequence (Line 26).

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

The proof of Theorem 4 can get access at the full version [37].

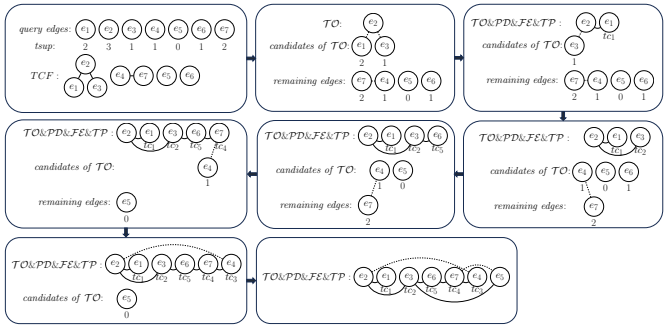


Figure 7: The construction of $\mathbb{TCQ}+$

Example 4. *When constructing a $\mathbb{TCQ}+$ graph for the query graph,* temporal-constraints *graph, and data graph in Figure 2, we first build a TCF. For each pair of edges with* temporal-constraint*s, if they are connected, we link these edge nodes. Such as the* temporal-constraint *tc = (2, 3, 5), so build an edge between edge vertex N_{e_2} and N_{e_3}. Similarly, we calculate the tsup for each edge and select the edge with the highest tsup to add to $\mathcal{TO}$. Simultaneously, we add the neighboring edge nodes in TCF to* candidates of *$\mathcal{TO}$. We then choose the edge in* candidates of *$\mathcal{TO}$ with the highest tsup to add to $\mathcal{TO}$. The preceding edge in $\mathcal{TO}$ is its prec, and we connect them with a solid line. We also add the neighboring edge nodes in TCF to* candidates of *$\mathcal{TO}$. We check if this edge shares a common vertex with any other edge in $\mathcal{TO}$, excluding the common vertex it shares with prec; if so, we connect them with corresponding dashed lines. We also check if this edge forms any pairs of constraint edges with other edges in $\mathcal{TO}$ according to the* temporal constraint*s. If all edges that belong to the same forest as the preceding edge in $\mathcal{TO}$ are already in $\mathcal{TO}$, we select the remaining edge that is connected to $\mathcal{TO}$ and has the highest tsup to add to $\mathcal{TO}$, following the same process for the remaining edges.*

4.2 Algorithm TCSM-E2E

At first, we need to generate the initial candidates for each edge in the data temporal graph $\mathcal{G}$. Given G_q and $\mathcal{G}$, the generation of initial candidates process is to obtain all possible data edges set which can match e for each query edge e. We design *label degree filter* to judge a data vertex e_c whether can match a query edge e to make the candidate set as small as possible.

Definition 11 (Label Degree Filter(LDF)). *Given a query graph $G_q = (V_q, E_q, \mathcal{L}_q)$ and a data temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L}, \mathcal{T})$. Given $e \in E_q$ and $e_c \in \mathcal{E}$, a* neighbor label filter *(e_c, e) is an injective function h: $e_c \to e$ 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 $\mathcal{TC}$, G_q and $\mathcal{G}$ as input, and outputs all *temporal-constraint subgraph matching* from G_q to $\mathcal{G}$ under the temporal-constraint $\mathcal{TC}$. We first find all the possible candidates in $\mathcal{G}$ for each query edge (Lines 1-3). Next, we generate the matching order $\mathcal{TO}$, the *prec dictionary* $\mathcal{PD}$, and the *forward edge* set $\mathcal{FE}$, and the *temporal-constraint dictionary* $\mathcal{TP}$ (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

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

input : a temporal-constraint $\mathcal{TC}$, a query graph G_q and a data temporal graph $\mathcal{G}$
output: all *temporal-constraint subgraph matching* from G_q to $\mathcal{G}$ under the temporal-constraint $\mathcal{TC}$

// 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 $\mathbb{TCQ}$+ graph
4 $(\mathcal{TO}, \mathcal{PD}, \mathcal{FE}, \mathcal{TP}) \leftarrow \mathbb{TCQ}+(\mathcal{TC}, G_q, \mathcal{G})$;
// 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* ($\mathcal{G}, \mathcal{TO}, \mathcal{PD}, \mathcal{FE}, \mathcal{TP}, M, \lambda$+1);
9 **Procedure** DFS($\mathcal{G}, \mathcal{TO}, \mathcal{PD}, \mathcal{FE}, \mathcal{TP}, M, \lambda$):
10 **if** $\lambda = |\mathcal{TO}|$ +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*($\mathcal{FE}, \mathcal{TP}, M, e, e_c, e'$) *is true* **then**
19 $M[e] \leftarrow e_c$;
20 *DFS* ($\mathcal{G}, \mathcal{TO}, \mathcal{PD}, \mathcal{FE}, \mathcal{TP}, M, \lambda$+1);
21 **Function** Validate($\mathcal{FE}, \mathcal{TP}, M, e, e_c, e'$):
22 **if** $M[\mathcal{FE}(e)] \cap e_c = \emptyset$ **then**
23 **return** false;
24 **for** *each* $tc \in \mathcal{TC}$ **do**
25 **if** M *do not satisfy time constraint* tc **then**
26 **return** false;
27 **return** true;

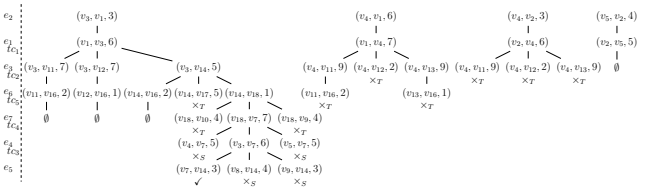


Figure 8: A running example of TCSM-E2E **on Figure 2**

result (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 mapped 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 e, Line 18 checks whether e has a common vertex with the match of e' if the query edge e has a common vertex with a query edge e' that is already matched and not its *prec* edge. And it also needs to check whether the temporal information of the data edges satisfy *temporal-constraint* if there exist a *temporal-constraint* which the largest order number of its edge is λ. If so, we expand the partial match by matching the next query edge (Lines 19-20). Otherwise, the algorithm backtracks to match the next data candidate edge.

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

The proof of Theorem 5 can get access at the full version [37].

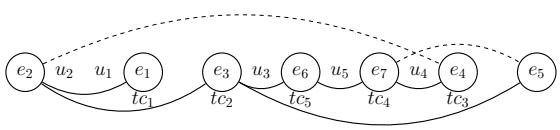
Example 5. *Figure 8 show the matching result of using* TCSM-E2E *in query graph* G_q *and data graph* $\mathcal{G}$ *under the temporal-constraint* $\mathcal{TC}$. *The symbol* ✓ *represents the match of this path is a correct match. The match* $M = \{((v_3, v_1, 3), e_2), ((v_1, v_3, 6), e_1), ((v_3, v_{14}, 5), e_3), ((v_{14}, v_{18}, 1), e_6), ((v_{18}, v_7, 7), e_7), ((v_3, v_7, 6), e_4), ((v_7, v_{14}, 3), e_5)\}$ *is a match satisfy condition. The symbol* ✗*(S) represents there exist a vertex in of a candidate edge to match e cannot map to the vertex in M. The symbol* ✗*(T) represents the candidate data edge cannot match the query edge with temporal error, which means the temporal information along the matched data edges cannot satisfy some* temporal-constraint. *The symbol* ∅ *represents there is no candidate edge of e when match query edge e. When a candidate data edge can not match the query edge or the candidate set is empty, the algorithm backtracks. When matching* e_3, *and the partial match* M=$\{(e_2, (v_3, v_1, 3)), (e_1, (v_1, v_3, 6))\}$, *the match of* e_2 *(the Prec of* e_3*) is* $(v_3, v_1, 3)$ *and the structural relationship between* e_2 *and* e_3 *is that the source vertex of* e_2 *is also the source vertex of* e_3. *So, when we search for candidate edges for* e_3, *we only need to examine the edges formed by the source vertex* v_3 *of* e_2 *and its neighboring vertices, identifying those that point from* v_3 *to a neighbor with a label matching the target vertex of query edge* e_3. *The candidate edges set of* e_3 *is* $\{(v_3, v_{11}, 7), (v_3, v_{12}, 7), (v_3, v_{14}, 5)\}$, *is considerably smaller than the initial candidate set. When matching* e_4, *and the partial match* M=$\{(e_2, (v_3, v_1, 3)), (e_1, (v_1, v_3, 6)), (e_3, (v_3, v_{14}, 5)), (e_6, (v_{14}, v_{18}, 1)), (e_7, (v_{18}, v_7, 7))\}$, *the candidate edges set of* e_4 *is* $\{(v_4, v_7, 5), (v_3, v_7, 6), (v_5, v_7, 5)\}$. *The* forward edge *of* e_4 *is* e_2. *When determining whether the candidate data edge* $(v_4, v_7, 5)$ *can structurally match edge* e_4, *we need to verify whether there exist common vertex between edge* $(v_3, v_1, 3)$ *(the match of* e_2*) and* $(v_4, v_7, 5)$. *It is evident that there is no common vertex between* $(v_3, v_1, 3)$ *and* $(v_4, v_7, 5)$*; therefore, data edge* $(v_4, v_7, 5)$ *cannot be structurally matched with edge* e_4. *The same applies to dat edge* $(v_5, v_7, 5)$. *When matching* e_6, *and the partial match* M=$\{(e_2, (v_3, v_1, 3)), (e_1, (v_1, v_3, 6)), (e_3, (v_3, v_{14}, 5))\}$. *The candidate data edge* $(v_{14}, v_{17}, 5)$ *can structurally match* e_6. *From the* $\mathbb{TCQ}$+ *graph, we can see that the latest edge in the matching sequence for the* temporal-constraint tc_5 *is* e_6. *Therefore, at* e_6, *we need to check whether the interactions of matched edges within* tc_5 *satisfy these constraints. In the data graph, it's evident that the partial match at this point does not satisfy* tc_5, *so* $(v_{14}, v_{17}, 5)$ *cannot match* e_6.

4.3 Algorithm TCSM-EVE

The primary distinction between TCSM-EVE and TCSM-E2E algorithm lies in whether vertex matching is performed after each layer of edge matching, depending on whether a new vertex is added to the partial match or not. The inclusion of vertex matching between edge matches helps prune invalid candidate edges. Specifically, when a new query vertex u is added, the vertex matching process ensures that there is a match between the neighbors of u's match and all of u's *backward neighbor* vertices (Function *Vmatch*).

Definition 12 (Backward Neighbor). *Given a matching order* $\mathcal{TO}$, *a query graph* $G_q = (V_q, E_q, \mathcal{L}_q)$ *and a vertex* $u \in V_q$, *a backward neighbor of u, denoted as BN(u), is a set that contains all neighbor vertices of u in* V_q *which appear later than u in* $\mathcal{TO}$.

Example 6. *Figure 9 illustrates the construction of the temporal-constraint query graph* $\mathbb{TCQ}$+ *in the* TCSM-EVE *algorithm. Unlike*



$\mathcal{TO} = \{1{:}\ e_2, 2:e_1, 3:e_3, 4:e_6, 5:e_7, 6:e_4, 7:e_5\}$, $\mathcal{PD} = \{e_1:e_2,\ e_3:e_2,\ e_6:e_3,\ e_7:e_6,\ e_4:e_7,\ e_5:e_3\}$

$\mathcal{FE} = \{e_1{:}\ \emptyset,\ e_3{:}\ \emptyset,\ e_6{:}\ \emptyset,\ e_7{:}\ \emptyset,\ e_4{:}\ \{e_2\},\ e_5{:}\ \{e_7\}\}$, $\mathcal{TP} = \{tc_1{:}\ e_1,\ tc_2{:}\ e_3,\ tc_3{:}\ e_4,\ tc_4{:}\ e_7,\ tc_5{:}\ e_6\}$

Figure 9: $\mathbb{TCQ}$+ in algorithm TCSM-EVE.

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

input : a temporal-constraint $\mathcal{TC}$, a query graph G_q and a data temporal graph $\mathcal{G}$
output: all *temporal-constraint subgraph matching* from G_q to $\mathcal{G}$ under the temporal-constraint $\mathcal{TC}$

```
   // 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          M[e.u]=e_c.u, M[e.v]=e_c.v;
6          DFS(G, TO, PD, FE,TP,M, λ+1);
7  Procedure DFS(G, TO, PD, FE,TP,M, λ):
8      if λ = |TO| +1 then
9          print M;
10     else
11         e ← TO[λ], e' ← PD[e], e.C ← ∅;
           // generate candidate in current state
12         for each e_c ∈ M[e'].adje do
13             if L(e_c.u)=L(e.u)&L(e_c.v)=L(e.v) then
14                 e.C ← e_c;
15         for each e_c ∈ e.C do
16             if M[FE(e)] ∩ e_c ≠ ∅ & e_c can satisfy TC with the edge in M then
17                 M[e] ← e_c;
18                 if {e.u, e.v}−{e'.u, e'.v} ≠ ∅ then
19                     u ← {e.u, e.v}−{e'.u, e'.v};
20                     v←{e_c.u, e_c.v}−{M[e'].u, M[e'].v};
21                     if Vmatch(u, v) then
22                         M[u]=v;
23                         DFS(G, TO, PD, FE,TP,M, λ+1);
24                 else
25                     DFS(G, TO, PD, FE,TP,M, λ+1);
26 Function Vmatch(u, v):
27     for each u' ∈ BN(u) do
28         if ∄ v' ∈ N(v), L(v') = L(u') then
29             return false;
30     return true;
```

the TCSM-E2E *algorithm, which focuses solely on edge sequences,* TCSM-EVE *simultaneously generates matching sequences for both edges and vertices. For instance, when edge e_2 is added in the specified edge order, the corresponding vertices u_2 and u_1 are introduced, and they are added in the vertex order. When edge e_1 is incorporated, no new vertices are introduced, and as a result, no vertices are added in the vertex order. Similarly, when e_3 is added, the additional vertex u_4 is included in the vertex order.*

Algorithm 5 outlines our TCSM-EVE algorithm, which shares similarities with the TCSM-E2E algorithm and thus omitted portions. The matching process begins by recursively expanding the partial match in an edge-to-edge manner following $\mathcal{TO}$ (Lines 1-3). Before matching the second edge, we ascertain whether the endpoints of the two vertices of the first edge can be matched. If so, we incorporate the match of these two vertices into the partial match

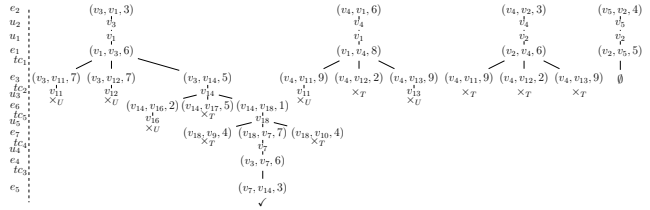


Figure 10: A running example of TCSM-EVE **on Figure 2.**

M and proceed to match the next edge (Lines 4-6). Upon matching all query edges, we output the result (Lines 8-9). Otherwise, we retrieve the next query edge e in $\mathcal{TO}$ along with its *prec*, and initialize the candidate data edges set of e to empty (Line 11). We derive the candidate data edges of e based on the data edge mapped to the *prec* of e and the structural correlation between e and its *prec* (Lines 12-14). For each candidate data edge e_c of e, Line 16 evaluates the structural and temporal constraints of the match between the candidate edge and the query edge. If a match is feasible, we include it in the partial match (Line 17) and verify whether a new vertex emerges in this layer of edge matching (Line 18). If not, we proceed to match the next edge (Line 25). However, if a new vertex emerges, we assess whether this vertex can match the corresponding query vertex using backward neighbor matching (Line 21). If the newly added vertex can be matched, we incorporate it into the partial match and continue matching the subsequent edge (Lines 22-23).

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 $O(|E_q|)$, respectively.*

The proof of Theorem 6 can get access at the full version [37].

Example 7. *Figure 10 illustrates the matching result of using* TCSM-EVE *on a query graph G_q and data graph $\mathcal{G}$ under the temporal constraint $\mathcal{TC}$. The symbol ✓ indicates that the match is correct. The match $M = \{((v_3, v_1, 3), e_2), (v_3, u_2), (v_1, u_1), ((v_1, v_3, 6), e_1), ((v_3, v_{14}, 5), e_3), (v_{14}, u_3), ((v_{14}, v_{18}, 1), e_6), (v_{18}, u_5), ((v_{18}, v_7, 7), e_7), (v_7, u_4), ((v_3, v_7, 6), e_4), ((v_7, v_{14}, 3), e_5)\}$ represents a valid match. The symbol ✗(U) indicates that an additional data vertex in the edge-matching process cannot match the corresponding query vertex. For example, consider the candidate data vertex v_{12} attempting to match u_3 in the partial match $M = ((v_3, v_1, 3), e_2), (v_3, u_2), (v_1, u_1), ((v_1, v_3, 6), e_1), ((v_3, v_{14}, 5), e_3)$. The data vertex v_{12} cannot match u_3 because query vertex u_3 has* backward neighbors *u_4 and u_5 with labels D and A, respectively. However, the data vertex v_{12} only has two neighboring vertices with labels B and A, and thus v_{12} cannot match u_3.*

5 EXPERIMENTS

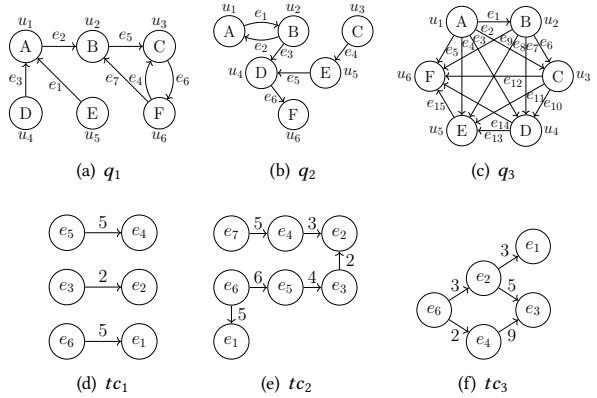
5.1 Experimental setup

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).

Datasets. We use 7 datasets that have been popularly used for the evaluation of previous temporal subgraph matching method [39]. Table 1 lists their statistics. All the datasets are downloaded from https://snap.stanford.edu/data/, and they are listed in an increasing

Table 1: Data Temporal Graph $\mathcal{G}$

Dataset	$\|V\|$	$\|\mathcal{E}\|$	$\|E\|$	Time span	d
CM	1,899	59,835	20,296	193 days	15.75
EE	986	332,334	24,929	803 days	168
MO	24,818	506,550	239,978	2,350 days	10.21
UB	159,316	964,437	596,933	2,613 days	3.03
SU	194,085	1,443,339	924,886	2,773 days	3.72
WT	1,140,149	7,833,140	3,309,592	2,320 days	3.44
SO	2,601,977	63,497,050	36,233,450	2,774 days	12.20



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

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

Figure 11: Queries q and Temporal-Constriants tc.

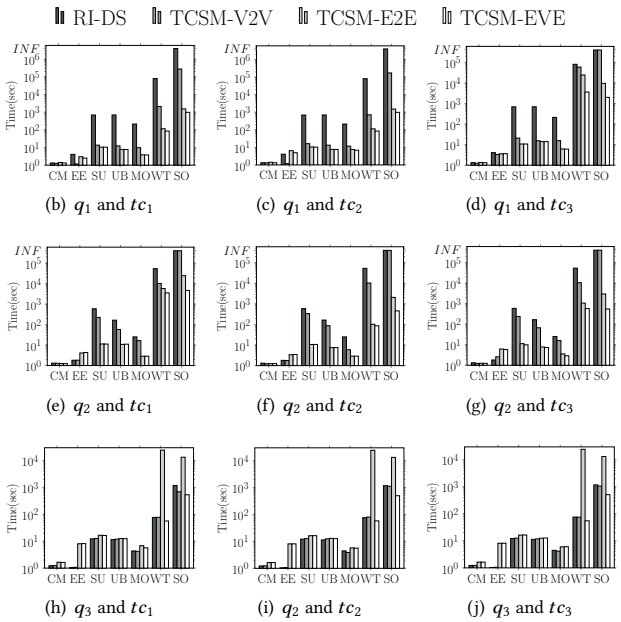
order of number of edges. CM (*CollegeMsg*) is the datasets of messages on a facebook-like platform at UC-Irvine. EE (*email-Eu-core-temporak*) is the datasets of e-mails between users at a research institution. MO (*sx-mathoverflow*) is the datasets of comments, questions, and answers on Math Overflow. 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.

Queries and Temporal-Constriants. We employed three labeled queries and three temporal constraints between edges, as illustrated in Figure 11. 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 subgraph isomorphism method designed for graphs without temporal information. Proposed for static graphs [7, 8], RI-DS introduces the concept of compatibility domains to filter candidate target vertices for query vertices before starting the matching process. In our experiments, we use the subgraphs matched by RI-DS with an additional temporal constraint filtering as our baseline. Therefore, the experimental section employs the following four algorithms: RI-DS, TCSM-V2V, TCSM-E2E, and TCSM-EVE. Furthermore, we modified the state-of-the-art continuous subgraph matching algorithms (SymBi [49], Turboflux [31], Graphflow [26], SJ-Tree [11] and IEDyn [21]) to solve our problem by additionally checking whether the embeddings found by these methods satisfy the temporal order, and included them in our comparisons. The source codes of these algorithms are obtained from [58] and all methods are implemented in C++.

Table 2: Running time of various temporal methods (second).

Temporal 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) q_1 and tc_1 (c) q_1 and tc_2 (d) q_1 and tc_3

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

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

Figure 12: Runtime with different q and tc.

5.2 Experimental Results

Exp-1: Running time of various temporal methods. Table 2 compares the performance of six temporal algorithms with our proposed algorithms across five datasets. The term "inf" indicates that the runtime exceeded three days. The results show that while our algorithms perform comparably to existing methods on smaller datasets (such as EE), they demonstrate superior performance on medium-sized datasets compared to mainstream matching algorithms. Moreover, on large-scale datasets with millions of nodes, our algorithms significantly outperform the existing approaches, highlighting their exceptional efficiency and scalability.

Exp-2: Runtime of our algorithms V.S. baselines. Figure 12 displays the running times for RI-DS, TCSM-V2V, TCSM-E2E, and TCSM-EVE across all datasets, consistently returning identical subgraphs. The results indicate:

TCSM-V2V generally outperforms RI-DS in speed, especially with larger queries and denser datasets. This efficiency is due to two key features of TCSM-V2V: (1) ordering query vertices by structural and

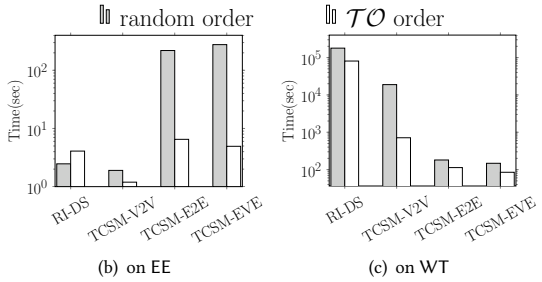


Figure 13: Runtime with different matching orders.

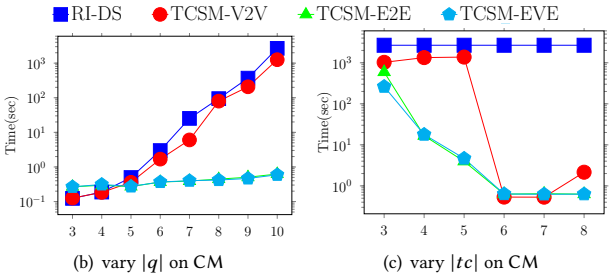


Figure 14: Runtime with different $|q|$ or $|tc|$.

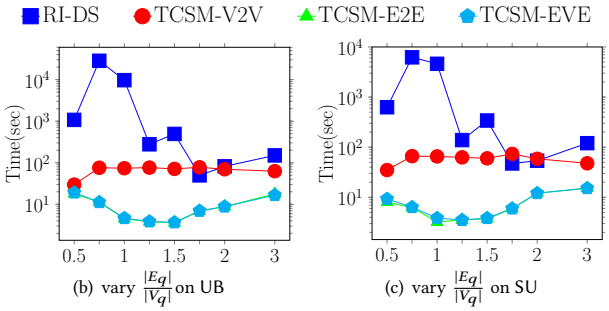


Figure 15: Runtime with queries of different density.

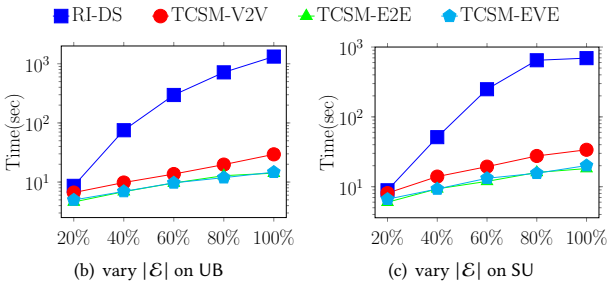


Figure 16: Runtime with different data temporal graph.

temporal constraints, and (2) generating candidate vertices from the preceding vertex. These methods reduce unnecessary computations by early filtering and minimizing redundant candidate selection. Conventional methods involve intersecting vertex sets, which is computationally heavy in dense graphs, while TCSM-V2V refines the candidate set more efficiently. However, when the query graph is fully connected, RI-DS outperforms all other algorithms, and its runtime becomes independent of the time-constrained graph.

TCSM-EVE outperforms TCSM-E2E due to it incorporates vertex matching pruning after each match. While their performance is similar on small datasets, TCSM-EVE is significantly faster on large datasets. In the worst-case scenario, when the query graph is fully connected, TCSM-EVE still performs well. It shows the best performance on medium and large datasets.

Exp-3: Ramdom matching order V.S. TO. Figure 13 presents bar charts comparing the computational times of various algorithms using both random matching order and our proposed TO order, based on the EE and WT datasets. Similar conclusions were observed across other datasets, which are omitted here for conciseness. 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 for subgraph matching in temporal graphs. Notably, our proposed order was found to be more than ten times faster than a random order, further validating the robustness of our algorithm.

Exp-4: Scalability with different size of queries and *temporal-constraints*. Figure 14 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 the size of the query graph ($|q|$) 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 temporal constraints ($|tc|$) 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 15 presents 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 returned identical subgraphs in these tests. Notably, the TCSM-E2E and TCSM-EVE algorithms performed best when the edge-to-vertex ratio was close to 1, suggesting that a balanced ratio optimizes their performance by simplifying graph topology. In contrast, the TCSM-V2V algorithm performed worse when the ratio was less than 1, indicating that it may require a more complex graph structure for optimal performance, likely due to its specific handling of vertex-to-vertex comparisons. The RI-DS and TCSM-V2V algorithms exhibit an initial increase in runtime, followed by a decrease as the ratio grows. Overall, their fluctuations are relatively large and less stable compared to the more consistent performance of TCSM-E2E and TCSM-EVE algorithm.

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 16, with consistent findings across other

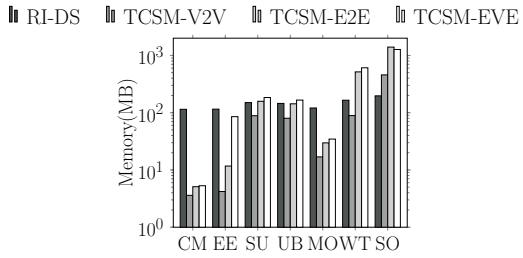


Figure 17: The memory usages of the algorithms.

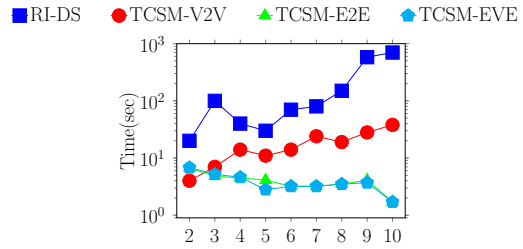


Figure 18: Runtime with vary $|\mathcal{L}_q|$ on UB.

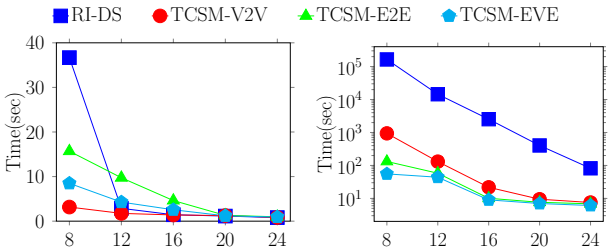


Figure 19: Runtime with $\mathcal{G}$ of different number of labels.

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 17 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 different number of labels. Figure 18 illustrates the runtime performance of all algorithms when querying a dataset with varying label types and quantities. 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.

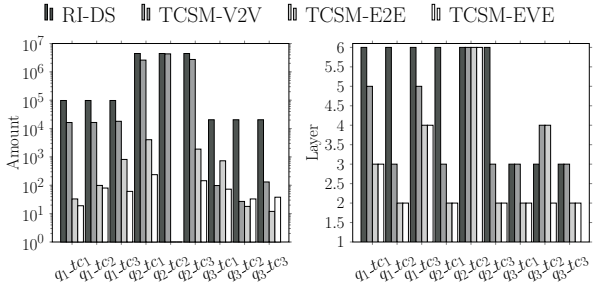


Figure 20: Number of failed backtracking matches on UB.

Exp-9: The effect of data temporal graph $\mathcal{G}$ with different number of labels. We generated five data graphs, each with a different number of distinct labels: 8, 12, 16, 20, and 24. Figure 19 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 (TCSM-V2V, TCSM-E2E, and TCSM-EVE) 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. These findings emphasize the importance of selecting the appropriate matching algorithm based on the dataset's structure, optimizing performance across various graph-based applications.

Exp-10: The effect of failed backtracking matches. Figure 20 compares the occurrences of backtracking and the layer of first backtracking across all algorithms on UB dataset, with consistent patterns observed on other datasets. The results indicate that TCSM-V2V algorithm experiences fewer total backtracking instances than RI-DS algorithm, suggesting more effective initial vertex matching or better preliminary checks. Additionally, TCSM-EVE shows fewer backtracks compared to TCSM-E2E, pointing to a more robust matching process. While RI-DS typically backtracks at the final layer, indicating a full-depth search approach, TCSM-V2V tends to backtrack one or two layers before the final layer, enabling earlier identification of mismatches. Moreover, TCSM-E2E and TCSM-EVE backtrack even earlier, employing proactive strategies that help avoid deep backtracking. These trends suggest that TCSM-V2V, TCSM-E2E, and TCSM-EVE implement more efficient initial matching strategies, reducing the need for costly late-stage backtracking and improving performance in complex queries. This insight is crucial for optimizing algorithm efficiency.

6 RELATED WORK

6.1 Subgraph isomorphism on static graphs

The main technical approaches for subgraph isomorphism include candidate filtering, ordering methods, and enumeration methods.

Filtering-Based Methods. Several filtering methods focus on local features. For example, label-and-degree filtering, as introduced in [1, 29, 34, 35, 60], considers a data vertex v as a candidate for query

vertex u if v has the same label as u and a degree no less than that of u. Neighborhood label frequency filtering [5] checks whether a candidate vertex of u has enough neighbors with matching labels compared to u. Recent methods combine label-and-degree filtering with neighborhood label frequency filtering, and they also use pseudo-matching on nearby vertices of candidates and query vertices [56], or match using a spanning tree or a *DAG* derived from the query graph [4, 6, 15, 28]. Unlike previous approaches, which establish a candidate set before matching and do not modify it, our methods dynamically adjust candidate vertices or edges based on the matching status of the *prec* vertex or edge.

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 [6, 7, 17, 46, 56]. However, there is no universal method for generating an optimal ordering for arbitrary query graphs and data graphs [9, 54, 63]. 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* [53], *RI* [8], and *VF2++* [25]. The second type, Indexing Enumeration, constructs indexes on the data graph and uses these indices to answer queries, as seen in *GADDI* [64], *GuP* [3], *SUFF* [22], *HUGE* [62], *Circinus* [24] and *SGMatch* [51]. 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* [19], *TurboISO* [18], and *CECI* [4].

6.2 Subgraph isomorphism on temporal graphs

Subgraph isomorphism in static graphs has been extensively studied, leading to a well-established body of research. However, subgraph isomorphism in temporal graphs—a class of graphs where edges have associated time information—has received comparatively less attention. Only a few algorithms have been proposed to tackle the problem of Temporal Subgraph Isomorphism (TSI) [30, 38, 40, 43–45, 47, 52, 56]. For instance, the work presented in [44] proposes an algorithm that sorts edges based on their time series before proceeding with the matching process. Other algorithms, such as those in [30, 43, 47, 52], focus on matching subgraphs under global time constraints, employing an edge-by-edge matching strategy. Earlier research, such as [42], utilized a vertex-by-vertex approach, which is similar to the classical RI algorithm [8]. In parallel, other studies explore different aspects of temporal graph pattern matching. For example, works like [2, 12, 13] investigate pattern matching in temporal graphs using database query techniques, showing a different perspective on handling temporal data. Closely related to TSI is the problem of searching for temporal motifs, which involves identifying small, recurrent temporal subgraphs (or motifs) characterized by k vertices or m edges [41, 59]. The concept of a temporal motif, as introduced in [50, 57], is defined as a match of a small query pattern where the edge timestamps follow a specified temporal order. However, these studies primarily address the problem of counting such temporal motifs when they consist of 2 or 3 edges. Further refinements of the temporal motif matching problem can be seen in works like [66], where the offset between the timestamps of query graph edges and their corresponding matching edges is constrained to be constant. Another study, [33], approaches the problem by focusing on finding temporal motifs that satisfy a flow constraint, where the flow on each edge represents a specific quantity, such as money or messages. Additionally, time-constrained graph pattern matching has also been explored through graph simulation techniques [20], as seen in studies like [14, 43, 65]. This approach offers a different definition of graph pattern matching compared to the standard subgraph isomorphism problem, highlighting the diversity of methods available for tackling challenges in temporal graphs.

While subgraph isomorphism in static graphs has been extensively studied, subgraph isomorphism in temporal graphs has received less attention. Only a few algorithms tackle the problem of Temporal Subgraph Isomorphism (*TSI*) [30, 38, 40, 43–45, 47, 52, 56]. For instance, [44] proposes an algorithm that sorts edges by their time series before matching. Algorithms like [30, 43, 47, 52] focus on matching subgraphs with global time constraints, using an edge-by-edge matching approach, while earlier versions of this work [42] applied a vertex-by-vertex approach similar to the original *RI* algorithm [8]. Other studies, such as [12] and [2, 13], implement pattern matching on temporal graphs using database query techniques. A closely related problem to *TSI* is the search for temporal motifs, which involves finding all small, recurrent temporal subgraphs of interactions, known as motifs, with k vertices or m edges [41, 59]. [50, 57] introduced the concept of a temporal motif, defined as a match of a small query pattern where the timestamps on its edges follow a specified order. However, they only address the counting problem for temporal motifs with 2 or 3 edges. [66] constrains the offset between the timestamps on query graph edges and their matching edges, requiring the offset to be constant. [33] focuses on finding temporal motifs with a limited sum of flows on the edge set, where the flow on each edge represents a quantity such as money or messages. Other studies, like [14, 43, 65], explore time-constrained graph pattern matching based on graph simulation [20], which defines graph pattern matching differently from our approach.

7 CONCLUSION

We address the problem of Temporal-Constraint Subgraph Matching (*TCSM*), an inherently *NP-hard* challenge. Traditional techniques, such as candidate filtering, often struggle to manage the complexities involved in TCSM. To tackle these challenges, we propose three innovative algorithms: TCSM-V2V, which employs vertex-to-vertex expansion and leverages temporal constraints to efficiently reduce duplicate matches; TCSM-E2E, which uses edge-to-edge expansion, significantly speeding up the matching process by minimizing vertex permutations; and TCSM-EVE, which utilizes an interactive edge-vertex-edge expansion strategy, eliminating both vertex and edge permutations to further decrease duplicate matches. Extensive experiments conducted on seven real-world

datasets demonstrate that our algorithms consistently outperform existing methods, achieving dramatic reductions in matching time.

REFERENCES

[1] Foto N. Afrati, Dimitris Fotakis, and Jeffrey D. Ullman. 2013. Enumerating subgraph instances using map-reduce. In *ICDE*. 62–73.

[2] Amir Aghasadeghi, Jan Van den Bussche, and Julia Stoyanovich. 2024. Temporal graph patterns by timed automata. *VLDB J.* 33, 1 (2024), 25–47.

[3] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. GuP: Fast Subgraph Matching by Guard-based Pruning. *Proc. ACM Manag. Data* 1, 2 (2023), 167:1–167:26.

[4] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *SIGMOD*. 1447–1462.

[5] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *SIGMOD*. 1199–1214.

[6] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *SIGMOD*. 1199–1214.

[7] Vincenzo Bonnici and Rosalba Giugno. 2017. On the Variable Ordering in Subgraph Isomorphism Algorithms. *IEEE ACM Trans. Comput. Biol. Bioinform.* 14, 1 (2017), 193–203.

[8] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis E. Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinform.* 14, S-7 (2013), S13.

[9] Sarra Bouhenni, Saïd Yahiaoui, Nadia Nouali-Taboudjemat, and Hamamache Kheddouci. 2022. A Survey on Distributed Graph Pattern Matching in Massive Graphs. *ACM Comput. Surv.* 54, 2 (2022), 36:1–36:35.

[10] Sutanay Choudhury, Lawrence B. Holder, George Chin Jr., Khushbu Agarwal, and John Feo. 2015. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. In *EDBT*. 157–168.

[11] Sutanay Choudhury, Lawrence B. Holder, George Chin Jr., Khushbu Agarwal, and John Feo. 2015. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. In *EDBT*. 157–168.

[12] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD*. 2246–2258.

[13] Jean-Louis Giavitto and José Echeveste. 2014. Real-Time Matching of Antescofo Temporal Patterns. In *International Symposium on Principles and Practice of Declarative Programming*. 93–104.

[14] Syed Gillani, Gauthier Picard, and Frédérique Laforest. 2016. Continuous graph pattern matching over knowledge graph streams. In *DEBS*. 214–225.

[15] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *SIGMOD*. 1067–1082.

[16] Saket Gurukar, Sayan Ranu, and Balaraman Ravindran. [n.d.]. COMMIT: A Scalable Approach to Mining Communication Motifs from Dynamic Networks. In *SIGMOD*. 475–489.

[17] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *SIGMOD*. 1429–1446.

[18] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*. 337–348.

[19] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*. 405–418.

[20] Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. 1995. Computing Simulations on Finite and Infinite Graphs. In *Annual Symposium on Foundations of Computer Science*. 453–462.

[21] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2020. General dynamic Yannakakis: conjunctive queries with theta joins under updates. *VLDB J.* 29, 2-3 (2020), 619–653.

[22] Xun Jian, Zhiyuan Li, and Lei Chen. 2023. SUFF: Accelerating Subgraph Matching with Historical Data. *Proc. VLDB Endow.* 16, 7 (2023), 1699–1711.

[23] Tatiana Jin, Boyang Li, Yichao Li, Qihui Zhou, Qianli Ma, Yunjian Zhao, Hongzhi Chen, and James Cheng. 2023. Circinus: Fast Redundancy-Reduced Subgraph Matching. *Proc. ACM Manag. Data* 1, 1 (2023), 12:1–12:26.

[24] Tatiana Jin, Boyang Li, Yichao Li, Qihui Zhou, Qianli Ma, Yunjian Zhao, Hongzhi Chen, and James Cheng. 2023. Circinus: Fast Redundancy-Reduced Subgraph Matching. *Proc. ACM Manag. Data* 1, 1 (2023), 12:1–12:26.

[25] Alpár Jüttner and Péter Madarasi. 2018. VF2++ - An improved subgraph isomorphism algorithm. *Discret. Appl. Math.* 242 (2018), 69–81.

[26] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *SIGMOD*. 1695–1698.

[27] Jonathan M Karpoff. 2021. The future of financial fraud. *Journal of Corporate Finance* 66 (2021), 101694.

[28] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2023. Fast subgraph query processing and subgraph matching via static and dynamic equivalences. *VLDB J.* 32, 2 (2023), 343–368.

[29] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, Jeong-Hoon Lee, Seongyun Ko, and Moath H. A. Jarrah. 2016. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *SIGMOD*. 1231–1245.

[30] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *SIGMOD*. 411–426.

[31] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *SIGMOD*. 411–426.

[32] Vitaliia Koibichuk, Natalia Ostrovska, Flora Kashiyeva, and Aleksy Kwilinski. 2021. Innovation technology and cyber frauds risks of neobanks: gravity model analysis. *Marketing i menedžment innovacij* 1 (2021), 253–265.

[33] Chrysanthi Kosyfaki, Nikos Mamoulis, Evaggelia Pitoura, and Panayiotis Tsaparas. 2019. Flow Motifs in Interaction Networks. In *EDBT*. 241–252.

[34] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable Subgraph Enumeration in MapReduce. *Proc. VLDB Endow.* 8, 10 (2015), 974–985.

[35] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2016. Scalable Distributed Subgraph Enumeration. *Proc. VLDB Endow.* 10, 3 (2016), 217–228.

[36] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2019. Distributed Subgraph Matching on Timely Dataflow. *Proc. VLDB Endow.* 12, 10 (2019), 1099–1112.

[37] Xiaoyu Leng, Guang Zeng, Hongchao Qin, Longlong Lin, Rong-Hua Li, and Guoren Wang. 2025. On Temporal-Constraint Subgraph Matching. (2025). full version available at: https://github.com/xiaoyu-ll/TSI/MPCfull.pdf.

[38] Faming Li and Zhaonian Zou. 2021. Subgraph matching on temporal graphs. *Inf. Sci.* 578 (2021), 539–558.

[39] Faming Li, Zhaonian Zou, and Jianzhong Li. 2023. Durable Subgraph Matching on Temporal Graphs. *IEEE Trans. Knowl. Data Eng.* 35, 5 (2023), 4713–4726.

[40] Faming Li, Zhaonian Zou, Jianzhong Li, Xiaochun Yang, and Bin Wang. 2022. Evolving subgraph matching on temporal graphs. *Knowl. Based Syst.* 258 (2022), 109961.

[41] Paul Liu, Austin R. Benson, and Moses Charikar. 2019. Sampling Methods for Counting Temporal Motifs. In *WSDM*. 294–302.

[42] Giorgio Locicero, Giovanni Micale, Alfredo Pulvirenti, and Alfredo Ferro. 2020. TemporalRI: A Subgraph Isomorphism Algorithm for Temporal Networks. In *Complex Networks*. 675–687.

[43] Yuliang Ma, Ye Yuan, Meng Liu, Guoren Wang, and Yishu Wang. 2020. Graph simulation on large scale temporal graphs. *GeoInformatica* 24, 1 (2020), 199–220.

[44] Patrick Mackey, Katherine Porterfield, Erin Fitzhenry, Sutanay Choudhury, and George Chin Jr. 2018. A Chronological Edge-Driven Approach to Temporal Subgraph Isomorphism. In *IEEE*. 3972–3979.

[45] Naoki Masuda and Petter Holme. 2020. Small inter-event times govern epidemic spreading on networks. *Physical Review Research* 2, 2 (2020), 023163.

[46] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (2019), 1692–1704.

[47] Giovanni Micale, Giorgio Locicero, Alfredo Pulvirenti, and Alfredo Ferro. 2021. TemporalRI: subgraph isomorphism in temporal networks with multiple contacts. *Appl. Netw. Sci.* 6, 1 (2021), 55.

[48] Seunghwan Min, Jihoon Jang, Kunsoo Park, Dora Giammarresi, Giuseppe F Italiano, and Wook-Shin Han. 2024. Time-Constrained Continuous Subgraph Matching Using Temporal Information for Filtering and Backtracking. In *ICDE*. 3257–3269.

[49] Seunghwan Min, Sung Gwan Park, Kunsoo Park, Dora Giammarresi, Giuseppe F. Italiano, and Wook-Shin Han. 2021. Symmetric Continuous Subgraph Matching with Bidirectional Dynamic Programming. *Proc. VLDB Endow.* 14, 8 (2021), 1298–1310.

[50] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. 2016. Motifs in Temporal Networks. *CoRR* abs/1612.09259 (2016).

[51] Carlos R. Rivero and Hasan M. Jamil. 2017. Efficient and scalable labeled subgraph matching using SGMatch. *Knowl. Inf. Syst.* 51, 1 (2017), 61–87.

[52] Konstantinos Semertzidis and Evaggelia Pitoura. 2020. A Hybrid Approach to Temporal Pattern Matching. In *ASONAM*. 384–388.

[53] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.* 1, 1 (2008), 364–375.

[54] Sun Shixuan and Luo Qiong. 2020. In-memory subgraph matching: An in-depth study. In *SIGMOD*. 1083–1098.

[55] Philip Olaseni Shoetan and Babajide Tolulope Familoni. 2024. Transforming fintech fraud detection with advanced artificial intelligence algorithms. *Finance*

& Accounting Research Journal 6, 4 (2024), 602–625.

[56] Shixuan Sun and Qiong Luo. 2022. Subgraph Matching With Effective Matching Order and Indexing. *IEEE Trans. Knowl. Data Eng.* 34, 1 (2022), 491–505.

[57] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: A holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment* 14, 2 (2020), 176–188.

[58] Xibo Sun, Shixuan Sun, Qiong Luo, and Bingsheng He. [n.d.]. An In-Depth Study of Continuous Subgraph Matching. *Proc. VLDB Endow.* 15, 7 ([n. d.]), 1403–1416.

[59] Xiaoli Sun, Yusong Tan, Qingbo Wu, Baozi Chen, and Changxiang Shen. 2019. TM-Miner: TFS-Based Algorithm for Mining Temporal Motifs in Large Temporal Network. *IEEE Access* 7 (2019), 49778–49789.

[60] Julian R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (1976), 31–42.

[61] I Wayan Widnyana and Sapta Rini Widyawati. 2022. 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)* 11, 6 (2022), 402–409.

[62] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. 2021. HUGE: An Efficient and Scalable Subgraph Enumeration System. In *SIGMOD*. 2049–2062.

[63] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. 2013. TripleBit: a Fast and Compact System for Large Scale RDF Data. *Proc. VLDB Endow.* 6, 7 (2013), 517–528.

[64] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: distance index based subgraph matching in biological networks. In *EDBT*. 192–203.

[65] Tianming Zhang, Yunjun Gao, Linshan Qiu, Lu Chen, Qingyuan Linghu, and Shiliang Pu. 2020. Distributed time-respecting flow graph pattern matching on temporal graphs. *World Wide Web* 23, 1 (2020), 609–630.

[66] Andreas Züfle, Matthias Renz, Tobias Emrich, and Maximilian Franzke. 2018. Pattern Search in Temporal Social Networks. In *EDBT*. 289–300.