TC-Match: Fast Time-constrained Continuous Subgraph Matching

Jianye Yang*
Guangzhou University
PengCheng Laboratory
jyyang@gzhu.edu.cn

Sheng Fang*
Guangzhou University
sheng@e.gzhu.edu.cn

Zhaoquan Gu
Harbin Institute of Technology (Shenzhen)
PengCheng Laboratory
guzhaoquan@hit.edu.cn

Ziyi Ma
Hebei University of Technology
zyma@hebut.edu.cn

Xuemin Lin
Shanghai Jiao Tong University
xuemin.lin@gmail.com

Zhihong Tian
Guangzhou University
tianzhihong@gzhu.edu.cn

ABSTRACT

Continuously monitoring structural patterns in streaming graphs is a critical task in many real-time graph-based applications. In this paper, we study the problem of time-constrained continuous subgraph matching (shorted as TCSM) over streaming graphs. Given a query graph Q with timing order constraint and a data graph stream G, TCSM aims to report all incremental matches of Q in G for each update of G, where a match should obey both structure constraint (i.e., isomorphism) and timing order constraint of Q. Although TCSM has a wide range of applications, such as cyber-attack detection and credit card fraud detection, we note that this problem has not been well addressed. The state-of-the-art bears the limitations of high index space cost and intermediate result maintenance cost. In this paper, we propose TC-Match, an effective approach to TCSM. First, we design a space and time cost-effective index CSS, which is essentially a k-partite graph structure where a node corresponds to an edge in G. By carefully creating links between nodes, we can encapsulate into CSS the partial embedding and timing order information between edges in G. We theoretically show that CSS has polynomial space and construction time complexities. Second, based on the property of CSS, we develop an efficient incremental matching algorithm with an effective node merging optimization. Extensive experiments show that TC-Match can achieve up to 3 orders of magnitude query performance improvement over the baseline methods, and meanwhile the memory consumption is reduced by 48.7%-86.7%.

PVLDB Reference Format:
Jianye Yang*, Sheng Fang*, Zhaoquan Gu, Ziyi Ma, Xuemin Lin, and Zhihong Tian. TC-Match: Fast Time-constrained Continuous Subgraph Matching. PVLDB, 17(11): XXX-XXX, 2024.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:

*Jianye Yang and Sheng Fang contribute equally to this work.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:XX.XX/XXX.XX

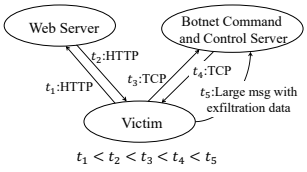


Figure 1: Attack pattern in network traffic [12].

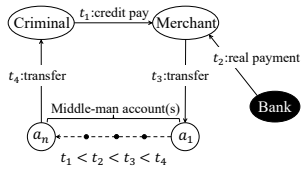


Figure 2: Credit card fraud in transactions [37].

The source code, data, and/or other artifacts have been made available at https://github.com/Sh-Fang/TCMatch.

1 INTRODUCTION

Graph is a widely used data structure to model complex systems, such as social networks, internet topologies, financial networks, and citation networks, etc. In real-world applications, graphs are highly dynamic due to the addition or deletion of edges. For example, in a social network, creating or cancelling the friendship between two users can be regarded as an edge addition or deletion, respectively. An important task on analysing such dynamic graphs is continuous subgraph matching (shorted as CSM), which has recently received significant research interests [12, 14, 15, 21, 22, 24, 26, 36, 46, 47, 53]. Given a pattern graph Q and a streaming graph G, CSM aims to efficiently report all incremental matches of Q in G for each update of G. CSM has many applications, such as merchant fraud detection [37], cyber-attack hunting [12], and rumor detection [51].

Motivation. In the past decade, a string of CSM algorithms have been proposed [12, 14, 15, 24, 26, 36, 46, 53]. All these algorithms find matches for the query graph by only considering the structure constraint, i.e., a result match is an isomorphic embedding of the query graph. However, in many applications, the timing constraint is also important. Specifically, there might exist some timing order constraints in the query graph, specifying that one edge is required to come before another one in the match. We call this variant problem as *time-constrained* continuous subgraph matching (shorted as TCSM). The problem of TCSM has many applications. In the following, we introduce a couple of examples.

Example 1 (**Cyber-attack detection**). *Figure 1 presents the pipeline of a typical cyber-attack pattern [12]. A victim browses a compromised website at time t_1, leading to downloading a malicious*

script at time t_2. After successfully invading the victim device, the threat actor establishes communication with the malicious C&C server. The infected victim host registers itselt at the C&C server at time t_3 and receives the command from the C&C server at time t_4. Once these commands are executed, further malware is installed, which gives the thread actor full control over the compromised machine. Finally, the exfiltrated data is sent back to C&C server at time t_5. Clearly, the time stamps in the above example follow a strict timing order, i.e., $t_1 < t_2 < t_3 < t_4 < t_5$. Therefore, this cyber-attack pattern can be modelled as a time-constrained query graph. By applying TCSM, we can quickly detect such cyber-attacks on network traffic data. As a matter of fact, subgraph matching semantic based method is drawing increasing attention to detect complex cyber-attacks, e.g., advanced persistent threat (APT) [11, 35].

Example 2 (**Credit card fraud detection**). *Figure 2 describes a typical credit card fraud pattern over a series of transactions [37]. A criminal tries to illegally cash out money from the bank with the help of a merchant and a middle man. In specific, the criminal conducts a fake purchase with the merchant at time t_1. Once the merchant receives the real payment from the bank at time t_2, he will transfer the money back to the criminal via one or more middle men, at times t_3 or t_4, respectively. Obviously, this pattern has a strict timing order $t_1 < t_2 < t_3 < t_4$. Using TCSM, the system can detect such transaction cycle in real time. Thus, the bank may reduce the loss by stopping such fraud in time.*

Existing solutions and limitations. To the best of our knowledge, Timing [30, 31] is the state-of-the-art to systematically study the problem of TCSM. It utilizes an intermediate result materialization method to report the matching results for each graph update. In specific, it decomposes the query graph into a set of subqueries, and maintains partial matches for each subquery. When the graph is updated, it first updates the partial matches, and obtains the overall matches by joining matches of each subquery. Although this method can speed up the online processing, its overall performance is not yet satisfactory due to the following two limitations. First, the space cost of Timing is very high due to the worst-case exponential number of partial matches as well as some intermediate join results. Second, the large size of intermediate results is also expensive to maintain for Timing, especially when the streaming graph has many query relevant edges. A more detailed discussion about Timing is given in Section 2.2.

Another method to deal with TCSM is first applying a CSM algorithm to list all embeddings of the query graph, and then verifying them posteriorly by considering the timing order constraints. However, the verification step is often cost-expensive, especially for datasets with a lot of false positive embeddings, i.e., a large number of embeddings do not satisfy the time constraints.

It is worth mentioning that the existing CSM algorithms [26, 36, 46, 53] all adopt a vertex-centric based index structure and an exploration-based incremental matching method. However, the timing order constraint in TCSM is defined on edges, which makes it difficult to directly apply the techniques of existing CSM algorithms to TCSM. In this paper, we aim to develop novel techniques that not only take advantage of exploration-based method, but can also naturally handle the temporal relationship between edges.

Our approach and contributions. In this paper, we propose an efficient approach, called TC-Match, following the commonly used incremental computation paradigm [30, 36, 46, 47, 53]. To avoid the limitations of existing solutions, we propose two techniques, including a novel index structure and an efficient incremental subgraph matching algorithm.

First, we propose a space and time cost-effective index structure, called *candidate storage structure* (shorted as CSS), to store the partial matchings as intermediate results to reduce the search space of backtracking. CSS is essentially a k-partite graph structure induced by the query graph Q and data graph G, where k is the number of edges in Q. Each group of nodes in CSS are corresponding to an edge ϵ in Q, containing edges in G matching to ϵ. A link is created between two nodes if (i) the corresponding edges are adjacent and satisfy the time constraint, and (ii) their mapping edges in Q are adjacent. Clearly, a node in CSS denotes a candidate matching edge for an edge in Q, and a link in CSS maintains the partial matching relationship between two edges in G regarding both structure and time constraints.

Second, we develop a CSS-based incremental subgraph matching algorithm, which follows the widely used *depth-first paradigm* [13, 42, 50]. We introduce a simple yet efficient candidate computation method based on the property of CSS. To further improve the computation performance, we propose a node merging optimization. Observe that nodes in CSS might have exactly the same neighborhood structure, which implies that we can process them in the same way. Thus, by merging such nodes together as a single node, we only need to process them once. After the matching process, we can restore the matching result by simply replacing this node with all merged nodes.

Extensive experimental results demonstrate that TC-Match can achieve up to 3 orders of magnitude query performance improvement over the baseline methods. In the meantime, the memory consumption of TC-Match is reduced by 48.7%-86.7% compared to the baseline methods under all experiment settings. Our principle contributions are summarized as follows.

- We propose a space and time cost-effective index structure CSS that maintains tight partial matchings by properly encapsulating the structure and temporal information, which in turn reduces the search space of backtracking.
- We propose a CSS-based incremental subgraph matching algorithm, which is optimized by effective node merging techniques.
- Extensive experimental results on real graphs demonstrate that TC-Match significantly outperforms the baseline algorithms by up to 3 orders of magnitude speedup, and meanwhile consumes much less amount of memory.

Roadmap. The rest of this paper is organized as follows. In Section 2, we introduce the problem definition and existing solution. In Section 3, we give the overview of our method. We introduce an novel index structure CSS and an efficient incremental subgraph matching algorithm in Section 4 and 5, respectively. We conduct extensive experiments in Section 6. Section 7 reviews the related work and Section 8 concludes the paper.

Table 1: Frequently used notations.

Notation	Meaning
g, Q, and G	graph, query graph, and data graph
$V(g)$ and $E(g)$	vertex set and edge set of g
ΔG	graph update stream
$L(v)$	label of vertex v
$L(e)$, $N(e)$, and $d(e)$	label, neighbors, and degree of edge e
$d_{\max}(g)$	maximum vertex degree in g
$\prec$	partial order between two edges
$\mathcal{O}$	timing order of a query graph
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	edge-centric index structure CSS
$e = (u, v, l, t)$	an edge in data graph or a node in CSS
$\mathcal{V}(\epsilon)$	a group of nodes with mapping edge ϵ
$\mathcal{P}(\epsilon)$	preceding set of edge ϵ

2 PRELIMINARIES

In this section, we introduce the problem definitions and state-of-the-art algorithm. Table 1 summarizes the mathematical notations frequently used in this paper.

2.1 Problem Definition

We focus on a directed, connected, and labeled graph $g = (V, E, L, \Sigma)$. Here, V is a set of vertices, $E \subseteq V \times V$ is a set of edges, Σ is a set of labels, and L is a labelling function that assigns each vertex $v \in V$ or edge $e \in E$ a label in Σ, denoted as $L(v, g)$ or $L(e, g)$. For two edges $e, e' \in E(g)$, we say e and e' are adjacent if they share a common endpoint. For each edge $e \in E(g)$, the neighbors of e, denoted as $N(e, g)$, are all adjacent edges of e, i.e., $N(e, g) = \{e' \in E(g) \setminus \{e\} | e \cap e' \neq \emptyset\}$. The degree of e, denoted as $d(e, g)$, is the number of neighbors of e, i.e., $d(e, g) = |N(e, g)|$. In the rest of the paper, we simplify $L(v, g)$, $L(e, g)$, $N(e, g)$, and $d(e, g)$ as $L(v)$, $L(e)$, $N(e)$, and $d(e)$, respectively, when the context is clear. We also simply refer a directed, connected, and labeled graph as a graph.

Definition 1 (Subgraph Isomorphism). *Given a query graph $Q = (V(Q), E(Q), L, \Sigma)$ and a data graph $G = (V(G), E(G), L, \Sigma)$, Q is subgraph isomorphic to G if and only if there exists an injective mapping m from $V(Q)$ to $V(G)$ such that $\forall u \in V(Q), L(u, Q) = L(m(u), G)$ and $\forall (u, u') \in E(Q), (m(u), m(u')) \in E(G)$ and $L((u, u'), Q) = L((m(u), m(u')), G)$.*

Note that, although we focus on subgraph isomorphism in this paper, subgraph homomorphism can be easily obtained by omitting the injective constraint in Definition 1. We call an injective mapping from vertices in Q to vertices in G as a *subgraph isomorphic embedding* of Q in G. Given two edges $\epsilon = (u, u')$ and $e = (v, v')$ in Q and G, respectively, we say ϵ and e are matching edges to each other, if $L(u, u') = L(v, v')$, and $L(u) = L(v)$ and $L(u') = L(v')$ (or $L(u) = L(v')$ and $L(u') = L(v)$), which is simply represented as $L(\epsilon) = L(e)$. we use the term "embedding" to simply refer to "subgraph isomorphic embedding" when the context is clear, and we may use *embedding* and *match*, interchangeably.

Definition 2 (Streaming Graph). *A streaming graph G is a constantly growing graph consisting of a sequence of incoming edges $\langle e_1, e_2, ..., e_n \rangle$ where $e = (u, v, l, t)$ denotes a directed edge from u to v with label l arriving at time t.*

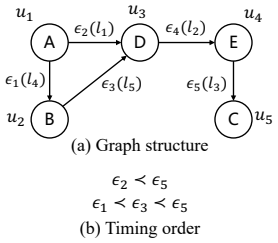


Figure 3: A time-constrained query graph Q.

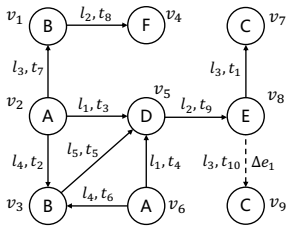


Figure 4: A data graph G with an edge insertion Δe_1.

Given two edges e and e', we say e precedes e', denoted as $e \prec e'$, if $e.t < e'.t$. For presentation simplicity, in this paper, we focus on edge insertion only for a streaming graph. Note that, out techniques can be readily extended to handle edge deletion to conform the time-based sliding window model [30]. We briefly discuss how to handle the case of edge deletion in Section 3.

Definition 3 (Timing Order). *Given a query graph $Q = (V(Q), E(Q), L, \Sigma)$, a timing order of Q, denoted as $\mathcal{O}(Q)$, is a strict partial order relation over edges in $E(Q)$. For two edges $\epsilon, \epsilon' \in E(Q)$, we use $\epsilon \prec \epsilon'$ to denote the partial order relation, which means that the matching edges of ϵ and ϵ' in the data graph should also obey this partial order relation regarding the arrival time.*

If an edge $\epsilon \in E(Q)$ is not defined in $\mathcal{O}$, we say ϵ is order-free (e.g., ϵ_4 in Figure 3), which is denoted as $\epsilon \notin \mathcal{O}$. An example time-constrained query graph is shown in Figure 3, which is described together by a graph structure and a timing order. For presentation simplicity, we use $Q = (V(Q), E(Q), \mathcal{O})$ to denote a query graph with timing order $\mathcal{O}$ in the rest of the paper.

Definition 4 (Time-Constrained Match). *Given a query graph $Q = (V(Q), E(Q), \mathcal{O})$ and a subgraph $S = (V(S), E(S))$ in a data graph G, S is a time-constrained match of Q if and only if the following conditions hold:*
(1) Structure Constraint (Isomorphism). *S is a embedding of Q in G on structure.*
(2) Timing Order Constraint. *For any two edges $\epsilon, \epsilon' \in E(Q)$: $\epsilon \prec \epsilon' \Rightarrow e \prec e'$ where $e \in E(S)$ and $e' \in E(S)$ are the matching edges of ϵ and ϵ' in S, respectively.*

Problem definition (*Time-constrained Continuous Subgraph Matching* (*TCSM*)). Given a query graph $Q = (V(Q), E(Q), \mathcal{O})$, a data graph G, and a graph update stream ΔG, the problem of time-constrained continuous subgraph matching is to find all *time-constrained matches* for Q under each graph update operation in ΔG.

Example 3. *Consider the query graph in Figure 3 and the data graph G with an edge insertion Δe_1 in Figure 4. Originally, there is no match in G. After inserting $\Delta e_1 = (v_8, v_9, l_3, t_{10})$, a new match is generated, namely $\{\langle (v_8, v_9, l_3, t_{10}), \epsilon_5 \rangle, \langle (v_5, v_8, l_2, t_9), \epsilon_4 \rangle, \langle (v_2, v_5, l_1, t_3), \epsilon_2 \rangle, \langle (v_2, v_3, l_4, t_2), \epsilon_1 \rangle, \langle (v_3, v_5, l_5, t_5), \epsilon_3 \rangle\}$.*

2.2 Existing Solution

The state-of-the-art Timing. The state-of-the-art algorithm for time-constrained continuous subgraph matching is Timing [30, 31]. In general, Timing utilizes an intermediate result materialization method to report the matching results for each graph update. Specifically, Timing decomposes the query graph Q into a set of subqueries $\{Q_1, \ldots, Q_k\}$, and maintains partial matches for each subquery. When the graph is updated, it first updates the partial matches, and then obtains the matches of Q by joining matches of each subquery. To speed up online processing, it also materializes some intermediate join results.

More specifically, for each subquery Q_i with edges $\{\epsilon_1, \ldots, \epsilon_n\}$, Timing maintains a set of expansion lists $\{L_i^1, \ldots, L_i^n\}$ where each list L_i^j corresponds to the first j edges in Q_i, denoted as $Prec(\epsilon_j) = \bigcup_{p=1}^{j} \epsilon_p$, and records a set of partial matches of $Pref(\epsilon_j)$. When a new edge e arrives, Timing locates the list L_i^j assuming e matches ϵ_j. If $j = 1$, e is simply inserted into L_i^1. Otherwise, it visits all matches in L_i^{j-1} to check if e can extend them as new matches for $Pref(\epsilon_j)$, which are inserted into L_i^j. After processing the subquery, Timing obtains the matches of Q by joining the matches of each Q_i following a precomputed join order. The intermediate join results are also materialized to speed up the online processing.

Drawbacks of Timing. Although Timing can correctly solve TCSM, it has the following two drawbacks.

• *Drawback 1: Large space cost.* First, Timing records all partial matches for each prefix of the edge sequence of each subquery. Second, it materializes some intermediate join results between subqueries. While a tree data structure is utilized to reduce the space cost, Timing is still space cost-expensive due to the exponential number of partial matches in the worst case as well as repeated matches of prefix edge sequence.

• *Drawback 2: Expensive intermediate result updating.* The large size of intermediate results is also expensive to maintain. When a new edge comes, Timing first needs to go through all matches in the expansion list of the corresponding prefix sequence. If new matches of the subquery are generated, it needs to further conduct join operation for the new matches and the existing matches of other subqueries. This process is repeated until no new partial matches are created or the new partial matches are exactly answers of the entire query Q.

3 AN OVERVIEW OF TC-MATCH

To solve TCSM, we propose a novel and efficient approach, called TC-Match, following the commonly used incremental computation paradigm [30, 36, 46, 47, 53]. Algorithm 1 summarizes the overall query processing, which consists of the following two phases.

• In the offline phase, we build an initial index structure based on the query graph Q and the data graph G (Line 1), where the details are presented in Section 4.3.

• In the online pahse, we find the incremental matches and maintain the index structure for each graph update operation. In specific, for edge insertion, we add the edge to G and update the index structure to keep it consistent with G (Lines 4-5), and then find the positive matches that contain the inserted edge (Line 6), where the details are discussed in Section 4.4 and Section 5, respectively. In contrast, for edge deletion, we first find the negative matches that contain the deleted edge, and then update the index structure to keep its consistency (Lines 8-10).

Discussion. The online processing is light-weight since both index updating and incremental matching are conducted in a neighborhood of the updated edge. Besides, it is clear that the operation for edge insertion and deletion is symmetric, i.e., the logic is exactly the same, which has also been discussed in [26, 36, 46]. Therefore,we focus on the edge insertion in the rest of the paper for brevity.

Algorithm 1: TC-Match(Q, G, ΔG)

```
Input  : Q: a query graph
         G: a data graph
         ΔG: a graph update stream
Output : incremental matches for each Δe ∈ ΔG
/* The offline phase.                                   */
1  G ← BuildCSS(Q, G);
/* The online phase.                                    */
2  for each Δe = (op, e) ∈ ΔG do
3      if op is + then
4          G ← G ⊕ e;
5          UpdateCSS-Ins(G, e);
6          FindIncrementalMatch(G, Q, e);
7      else
8          FindIncrementalMatch(G, Q, e);
9          G ← G ⊖ e;
10         UpdateCSS-Del(G, e);
```

4 AN EDGE-CENTRIC INDEX STRUCTURE

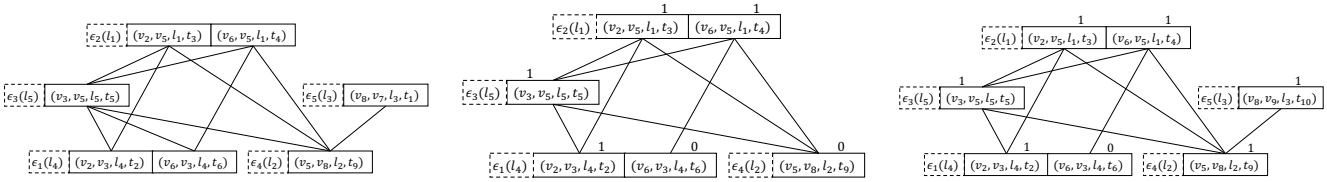
In this section, we introduce a novel index structure to efficiently maintain the intermediate results of our problem.

4.1 Main Idea of CSS

We introduce a novel edge-centric index structure, called *candidate storage structure* (CSS), to store the partial matchings as intermediate results to reduce the search space of backtracking since a partial matching is a necessary condition for a matching result.

In general, CSS is a k-partite graph structure induced by Q and G. Vertices in CSS are divided into $|E(Q)|$ groups, where each group is corresponding to an edge $\epsilon \in E(Q)$, containing all edges in G matching to ϵ. An edge is created between two vertices in CSS if (i) their corresponding edges in G are adjacent and satisfy the time constraint, and (ii) their mapping edges in Q are adjacent. Clearly, CSS maintains the Cartesian product for all matching results. That is, each group in CSS contributes exactly one edge for an embedding of the query graph. To further facilitate the matching processing, we introduce the state of vertices in CSS to indicate if its neighbors are all well "matched" in G. In the matching phase, we only traverse on such vertices in CSS to reduce redundant partial matchings.

Note that an edge in G must have at least one matching edge in Q if it contributes in a matching result. Thus, we simply ignore the irrelevant edges, i.e., these having no matches in Q. Our subsequent discussions always assume that G only contains query related edges.



(a) Coarse CSS of G (b) Refined CSS of G with node state (c) CSS of $G + \Delta e_1$

Figure 5: An example candidate storage structure CSS of the data graph in Figure 4.

4.2 Index Structure

Our index structure CSS is essentially a k-partite graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where k is the number of edges in the query graph, i.e., $|E(Q)|$. For ease of presentation, we first introduce a coarse version of CSS, and then develop a variety of pruning techniques to refine it.

Initializing CSS. We begin with the concept of CSS. For presentation simplicity, we assume that edges in Q have different labels, i.e., $L(\epsilon) \neq L(\epsilon')$ for any two edges $\epsilon, \epsilon' \in E(Q)$, and show how to extend our techniques for the case where multiple edges have the same label in Section 4.5. In the following, we refer the vertices and edges in CSS as nodes and links, respectively, to distinguish from those in query or data graphs.

Definition 5 (Candidate Storage Structure (CSS)). *Given a query graph $Q = (V(Q), E(Q), \mathcal{O})$ and a data graph $G = (V(G), E(G))$, the candidate storage structure $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a k-partite graph induced by Q and G with the following properties.*

- *Nodes in $\mathcal{V}$ are divided into $|E(Q)|$ disjoint groups $V_1, \cdots, V_{|E(Q)|}$, i.e., $\mathcal{V} = \bigcup_{i=1}^{|E(Q)|} V_i$ and $V_i \cap V_j = \emptyset$ for $i \neq j$, where each group V_i is corresponding to an edge $\epsilon_i \in E(Q)$, containing all edges in G matching to ϵ_i, i.e., $\forall e \in V_i : L(e) = L(\epsilon_i)$.*
- *$\forall e, e' \in \mathcal{V}$ belonging to different node groups, $(e, e') \in \mathcal{E}$ if e and e' are adjacent in G and their corresponding mapping edges in Q are adjacent.*

Example 4. *Figure 5(a) shows a coarse version of CSS based on the query graph in Figure 3 and the data graph in Figure 4 before insertion of edge Δe_1. Take ϵ_1 for example. Because there are two edges (v_2, v_3) and (v_6, v_3) matching ϵ_1, two nodes are created in CSS, which are (v_2, v_3, l_4, t_2) and (v_6, v_3, l_4, t_6). We create other nodes in a similar way. Note that although (v_1, v_4) has the same edge label l_2 with ϵ_4, it is not a matching edge of ϵ_4 because the label of their endpoint vertices are different. After creating the nodes in CSS, we add links for these with a common endpoint vertex. For example, there is a link between (v_2, v_5, l_1, t_3) and (v_3, v_5, l_5, t_5) since they are incident on v_5. Note that although (v_2, v_3, l_4, t_2) and (v_6, v_3, l_4, t_6) are incident on v_3, we do not add a link between them. This is because they belong to the same node group.*

Given an edge $\epsilon \in E(Q)$, we also use $\mathcal{V}(\epsilon)$ to denote the group of nodes in $\mathcal{V}$ matching to ϵ and call ϵ as the *mapping edge* of nodes in $\mathcal{V}(\epsilon)$. We may simply use the edge e in the data graph to denote the corresponding node in CSS. Clearly, the index structure CSS introduced in Definition 5 is sufficient to generate the matching results since it basically materializes the Cartesian product for the query graph by only considering the structural information. To facilitate the matching processing, we aim to make it as succinct as possible by taking the temporal information into consideration.

Objective 1. *To improve the computation efficiency, we have to refine the nodes and links of CSS as much as possible.*

Refining CSS. Next, we introduce effective techniques to realize Objective 1. We begin with an important concept.

Definition 6 (Preceding Set). *Given an edge ϵ in a query graph $Q = (V(Q), E(Q), \mathcal{O})$, the preceding set of ϵ, denoted as $\mathcal{P}(\epsilon)$, is a set of immediate preceding edges of ϵ in $\mathcal{O}$.*

Note that the preceding set is defined only based on the timing order $\mathcal{O}$. For example, the preceding set of ϵ_5 is $\mathcal{P}(\epsilon_5) = \{\epsilon_2, \epsilon_3\}$, while $\mathcal{P}(\epsilon_3) = \{\epsilon_1\}$ in the query graph in Figure 3.

Lemma 1. *Given an edge e in the data graph G and its matching edge ϵ in the query graph, we create a node in $\mathcal{V}(\epsilon)$ for e, if for each edge $\epsilon' \in \mathcal{P}(\epsilon)$, there exists at least one node $e' \in \mathcal{V}(\epsilon')$ with $e' \prec e$.*

Proof. Suppose there exists one edge $\epsilon' \in \mathcal{P}(\epsilon)$, such that we cannot find a node $e' \in \mathcal{V}(\epsilon')$ with $e' \prec e$. This implies that e cannot be part of matching result since its preceding edge e' does not exist. Besides, this condition will never be satisfied as the arrival time of an edge grows consistently. □

Lemma 1 states that a node e can be removed from CSS if we cannot find a corresponding node in CSS for all preceding edges of e. Take (v_8, v_7, l_3, t_1) for example. We have that its matching edge in query graph is ϵ_5 and $\mathcal{P}(\epsilon_5) = \{\epsilon_2, \epsilon_3\}$. Since there does not exist matching node in CSS for both ϵ_2 and ϵ_3 at time t_1, we can remove (v_8, v_7, l_3, t_1) from CSS safely.

According to Definition 5, a link is added into CSS between two nodes if they are adjacent in the data graph and their corresponding mapping edges in query graph are adjacent as well. However, we observe that this rule is still loose, which may lead to many redundant links. Next, we introduce a much strict link creating rule without missing any results.

Lemma 2. *Given two nodes e and e' in CSS, and their mapping edges ϵ and ϵ', respectively, we create a link between them if and only if the following four conditions hold:*
(1) e and e' belong to different node groups;
(2) e and e' are adjacent;
(3) ϵ and ϵ' are adjacent; and
(4) the arrival time of e and e' obey the time partial order defined by ϵ and ϵ'.

Proof. We consider condition (4) since the first three are given in Definition 5. According to the definition of time-constrained match, e and e' must satisfy the time partial order defined by ϵ and ϵ' if they co-exist in a matching result. □

Example 5. *Consider nodes (v_3, v_5, l_5, t_5) and (v_6, v_3, l_4, t_6) in Figure 5(a), with mapping edges ϵ_3 and ϵ_1, respectively. Because $(v_3, v_5, l_5, t_5) \prec (v_6, v_3, l_4, t_6)$ while $\epsilon_3 \succ \epsilon_1$, we therefore do not need to create a link between (v_3, v_5, l_5, t_5) and (v_6, v_3, l_4, t_6) although they are incident on v_3. The refined CSS is shown in Figure 5(b).*

Setting node state. To further facilitate the matching processing, we introduce the state of nodes in CSS based on the connection situation with other nodes.

Definition 7 (**Node State**). *Given a CSS graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, for a node $e \in \mathcal{V}(\mathcal{G})$, the state of e, denoted as $s(e)$, is a binary value 0 or 1, which is determined as follows. $s(e) = 1$ if there exists a link between e and at least one node in $\mathcal{V}(\epsilon')$ for each $\epsilon' \in N(\epsilon, Q)$ where ϵ is mapping edge of e and Q is the query graph, $s(e) = 0$ otherwise.*

Example 6. *Consider nodes (v_2, v_3, l_4, t_2) and (v_6, v_3, l_4, t_6) in the CSS in Figure 5(b), where the mapping edge is ϵ_1 and $N(\epsilon_1, Q) = \{\epsilon_2, \epsilon_3\}$. Since for both ϵ_2 and ϵ_3, (v_2, v_3, l_4, t_2) indeed has a link to the corresponding nodes, i.e., (v_2, v_5, l_1, t_3) and (v_3, v_5, l_5, t_5) respectively, we have that the state of (v_2, v_3, l_4, t_2) is 1. Contrarily, because there is no link between (v_6, v_3, l_4, t_6) and nodes for ϵ_3, the state of (v_6, v_3, l_4, t_6) is 0.*

Intuitively, for a node e in CSS, $s(e) = 1$ indicates that all neighbors of e are well "matched". This implies that we can ignore nodes with state being 0 during the matching processing since they must not be part of a matching result due to unfinished neighbors. We will discuss the matching processing based on CSS in Section 5.

4.3 Index Construction

Algorithm details. Algorithm 2 summarizes the details of constructing CSS. We initialize the node set $\mathcal{V}$ and link set $\mathcal{E}$ of CSS as $\emptyset$ (Line 1). Then, we sequentially process the edges of G in ascending order of the arrival time to construct $\mathcal{V}$ and $\mathcal{E}$ (Lines 2-9). In particular, given the matching edge ϵ of e, we check if e is order-free such as (v_5, v_8, l_2, t_9) in Figure 4, or there exists a corresponding node in $\mathcal{V}(\epsilon')$ for each edge $\epsilon' \in \mathcal{P}(\epsilon)$ based on Lemma 1 (Line 4). Note that, the partial order must hold as long as $\mathcal{V}(\epsilon')$ is not empty since we process edges in order. If yes, we create a node in $\mathcal{V}(\epsilon)$ for e (Line 5). Next, we create links between e and other nodes by applying Lemma 2 (Lines 6-10). More specifically, we iteratively consider each edge $e' \in N(e, G)$ and retrieve its matching edge ϵ' (Line 7). If edge relationship and the timing order between ϵ and ϵ', as stated in Lemma 2, are satisfied, we find the corresponding node in $\mathcal{V}(\epsilon')$ for edge e' and create a link between e and e' (Lines 8-10). After constructing the graph structure of CSS, we set the state of nodes in $\mathcal{V}$ according to Definition 7 (Lines 11-15).

Analysis. Below we analyze the space and time complexities of CSS. We use $d_{\max}(g)$ to denote the maximum vertex degree in graph g. To simplify the analysis, we assume there is no time constraint defined, i.e., $\mathcal{O} = \emptyset$, which does not affect the correctness of our analysis, since the time constraint always reduces the size of CSS.

Algorithm 2: BuildCSS(Q, G)

Input : $Q = (V(Q), E(Q), \mathcal{O})$: a query graph with timing order
$G = (V(G), E(G))$: a data graph
Output : $\mathcal{G} = (\mathcal{V}, \mathcal{E})$: the CSS index structure

1 $\mathcal{V} \leftarrow \emptyset$; $\mathcal{E} \leftarrow \emptyset$;
2 **for each** $e \in E(G)$ in ascending order of the arrival time **do**
3 $\epsilon \leftarrow$ the matching edge of e in Q;
4 **if** $\epsilon \notin \mathcal{O}$ **or** $\forall \epsilon' \in \mathcal{P}(\epsilon) : \mathcal{V}(\epsilon') \neq \emptyset$ **then**
5 $\mathcal{V}(\epsilon) \leftarrow \mathcal{V}(\epsilon) \cup \{e\}$;
6 **for each** $e' \in N(e, G)$ **do**
7 $\epsilon' \leftarrow$ the matching edge of e' in Q;
8 **if** $\epsilon' \neq \epsilon$ **and** $\epsilon' \cap \epsilon \neq \emptyset$ **and** $(\epsilon' \prec \epsilon$ **or** $\epsilon' \notin \mathcal{O}$ **or** $\epsilon \notin \mathcal{O})$ **then**
9 Find node e' in $\mathcal{V}(\epsilon')$;
10 $\mathcal{E} \leftarrow \mathcal{E} \cup \{(e, e')\}$;
11 **for each** $\epsilon \in E(Q)$ **do**
12 **for each** $e \in \mathcal{V}(\epsilon)$ **do**
13 **if** $\exists e' \in \mathcal{V}(\epsilon')$ s.t. $(e, e') \in \mathcal{E}$ for each $\epsilon' \in N(\epsilon, Q)$ **then**
14 $s(e) \leftarrow 1$;
15 **else**
16 $s(e) \leftarrow 0$;
17 **return** $\mathcal{G} = (\mathcal{V}, \mathcal{E})$;

Theorem 1. *The space complexity of CSS index structure is bounded by $O(d_{\max}(G) \times |E(G)|)$.*

Proof. We need $O(|E(G)|)$ space to store nodes in CSS since each data edge is dispatched to one node group. Given two data edges e and e', we add a link between e and e' only if $e \cap e' \neq \emptyset$ according to Lemma 2. Thus, to estimate the number of links in CSS, we need to compute the number of adjacent edge pairs in G. Now, consider an arbitrary vertex v in G. It is evident the number of adjacent edge pairs incident on v is $\binom{d(v)}{2} < d(v)^2/2$ in the worst case when the $d(v)$ edges have different labels. Therefore, the total number of adjacent edge pairs in G is at most $\sum_{v \in V(G)} d(v)^2/2 \leq d_{\max}(G) \times \sum_{v \in V(G)} d(v)/2 = d_{\max}(G) \times |E(G)|$. Since the query edges have different labels, an adjacent edge pair contributes at most one link in CSS based on Lemma 2. Thus, the overall space cost of CSS is bounded by $O(d_{\max}(G) \times |E(G)|)$. □

Theorem 2. *The time complexity of BuildCSS is $O((|E(Q)| + d_{\max}(G)) \times |E(G)|)$.*

Proof. The time cost of BuildCSS consists of two parts, namely building graph structure and setting node state. First, for each edge $e \in E(G)$, we need to check if there exists a corresponding node in CSS for each edge in the preceding sets (Line 4). The time cost is bounded by $O(|E(Q)|)$ since the partial order in $\mathcal{O}$ is defined on edges. To create links, we need to visit all neighbor edges of e. For each neighbor edge $e' \in N(e, G)$, we can find its matching edge ϵ' (Line 7) and check the time partial order (Line 8) in $O(1)$ time by using a hashmap to store the query edges. The corresponding node of e' in $\mathcal{V}(\epsilon')$ can also be found in $O(1)$ time by using a hashmap to record the position of nodes in $\mathcal{V}(\epsilon')$. Therefore, the time for graph construction is $O(|E(G)| \times (|E(Q)| + d_{\max}(G)))$. It is easy to

Algorithm 3: UpdateCSS-Ins($\mathcal{G}$, e)

```
Input  : G = (V, E): the CSS index structure
         e: the inserted edge
1  ϵ ← the matching edge of e in Q;
2  if ϵ ∉ O or ∀ϵ' ∈ P(ϵ) : V(ϵ') ≠ ∅ then
3      V(ϵ) ← V(ϵ) ∪ {e};
4      for each e' ∈ N(e, G) do
5          ϵ' ← the matching edge of e' in Q;
6          if ϵ' ≠ ϵ and ϵ' ∩ ϵ ≠ ∅ and (ϵ' ≺ ϵ or ϵ' ∉ O or ϵ ∉ O)
           then
7              Find node e' in V(ϵ');
8              E ← E ∪ {(e, e')};
9              if s(e') = 0 then
10                 if ∃e'' ∈ V(ϵ'') s.t. (e', e'') ∈ E for each
                   ϵ'' ∈ N(ϵ', Q) then
11                     s(e') ← 1;
12     if ∃e' ∈ V(ϵ') s.t. (e, e') ∈ E for each ϵ' ∈ N(ϵ, Q) then
13         s(e) ← 1;
14     else
15         s(e) ← 0;
```

see that the node state can be set by visiting the links in CSS only once. With Theorem 1, we have that the overall time complexity of BuildCSS is $O(|E(G)| \times (|E(Q)| + d_{\max}(G)))$. □

Remark. Theorem 1 and Theorem 2 show that both the space cost and construction time cost of CSS are polynomial to the data graph size in the worst case.

4.4 Index Maintenance

The processing of index update is similar to that of index construction since we build CSS according to the arrival time of edges. A nice property of index update is that, when an edge is inserted, we only need to update nodes in neighbor groups of CSS. With this property, our index CSS can be updated efficiently.

Algorithm details. Algorithm 3 illustrates the details of index update when an edge e is inserted. First, we create a node for e based on Lemma 1 (Lines 1-3), and add links between e and other nodes by applying Lemma 2 (Lines 4-8). After that, we update the the node state for the neighbor node e' if its state is previously 0 (Lines 9-11). Last, we set the state of e according to Definition 7 (Lines 12-15).

Example 7. *Consider the edge insertion event $\Delta e_1 = (v_8, v_9, l_3, t_{10})$ in Figure 4. Since the mapping edge of Δe_1 is ϵ_5 in Figure 3(a), we then check the edges in its preceding set $\mathcal{P}(\epsilon_5) = \{\epsilon_2, \epsilon_3\}$, where the corresponding node groups are both non-empty, as depicted in Figure 5(b). Thus, we create node (v_8, v_9, l_3, t_{10}) in $\mathcal{V}(\epsilon_5)$, and a link between it and the neighbor node (v_5, v_8, l_2, t_9). We also update the state of (v_5, v_8, l_2, t_9) to 1, because it has at least one link to the neighbor node groups, i.e., $\mathcal{V}(\epsilon_2)$, $\mathcal{V}(\epsilon_3)$, and $\mathcal{V}(\epsilon_5)$. Last, we set the state of (v_8, v_9, l_3, t_{10}) to 1. The updated CSS is shown in Figure 5(c).*

Lemma 3. *The time complexity of UpdateCSS-Ins is $O(d(e) \times d_{\max}(Q))$ where e is the newly inserted edge.*

Proof. The time to update graph structure is $O(d(e))$ according to Theorem 2. Since we need to update the state for neighbors of e, and each is bounded by $O(d_{\max}(Q))$ time, it is immediate that the overall update time is $O(d(e) \times d_{\max}(Q))$. □

Discussion. In Section 2.2, we mention that Timing is inefficient because it needs to visit all global matches in the expansion list to check if the incoming edge can extend them as new matches. However, in CSS, we only need to visit the neighborhood of the incoming edge, which is crucial to the performance improvement.

4.5 Extending to Multiple Mapping Edges

Recall that we previously assume that edges in the query graph Q have different labels. That is, an edge in the data graph G has only one matching edge in Q. Next, we discuss how to handle the case where multiple edges in Q have the same label.

Implementation details. We still maintain a node group for each edge in Q. When a new edge e is arriving, we first collect its matching edges in Q, denoted by $\Psi(e)$. Then, for each matching edge $\epsilon \in \Psi(e)$, we create a node for e in the node group $\mathcal{V}(\epsilon)$ based on Lemma 1, and add links between the created node and other nodes based on Lemma 2. Note that, since a data graph edge might exist in different groups, we do not add a link between nodes created from the same data graph edge. However, we do not need to consider this constraint for subgraph homomorphism. The overall processing is exactly the same as described in Algorithm 2 or Algorithm 3 except we now need to process all edges in $\Psi(e)$ rather than a single edge.

Cost analysis. We next analyze the space cost of CSS with the above extension.

Theorem 3. *The space complexity of CSS is bounded by $O(d_{\max}(Q) \times d_{\max}(G) \times |E(Q)| \times |E(G)|)$ in general case.*

Proof. In the worst case, each data edge is added to each node group of CSS if all edges in Q and G have the same label, where we need $O(|E(Q)| \times |E(G)|)$ space to store nodes. To estimate the number of links, we need to compute the number of adjacent edge pairs in both Q and G, which are actually bounded by $d_{max}(Q) \times |E(Q)|$ and $d_{max}(G) \times |E(G)|$, respectively, according to the proof of Theorem 1. Now, an adjacent edge pair in G may contribute a link for each pair in Q in the worst case. Thus, the total number of links in CSS is $O(d_{\max}(Q) \times d_{\max}(G) \times |E(Q)| \times |E(G)|)$ in the worst case. □

5 CSS-BASED SUBGRAPH MATCHING

In this section, we present the matching algorithm to enumerate the matching results based on CSS.

5.1 Matching Enumeration Algorithm

General idea. As aforementioned, the CSS basically maintains the Cartesian product for all matching results, where each node group in CSS contributes exactly one edge for an embedding of the query graph. To obtain the matches, we adopt the widely used *depth-first paradigm* [13, 42, 50]. In specific, we maintain a partial embedding and recursively add the nodes in CSS into the partial embedding to generate full embeddings. To enhance the performance, we only expand the nodes with state 1 and enforce the connectivity of the matching order.

Algorithm 4: FindIncrementalMatch($\mathcal{G}$, Q, e)

Input : $\mathcal{G} = (\mathcal{V}, \mathcal{E})$: the CSS index structure
$Q = (V(Q), E(Q), \mathcal{O})$: a query graph with timing order
e: the inserted edge
Output : $\mathcal{M}$: the incremental matching results

1 $\epsilon \leftarrow$ the matching edge of e in Q;
2 **if** ϵ is an order-free or closing edge **then**
3 $\quad \varphi \leftarrow$ generate a matching order for edges in $E(Q) - \{\epsilon\}$;
4 $\quad$ Enumerate(φ, 1, $\langle e, \epsilon \rangle$);
5 **return** $\mathcal{M}$;

6 **procedure** Enumerate(φ, i, M)
7 **if** $|M| = |E(Q)|$ **then**
8 $\quad \mathcal{M} \leftarrow \mathcal{M} \cup \{M\}$;
9 $\quad$ **return**;
10 $\epsilon' \leftarrow \varphi[i]$;
11 $C \leftarrow$ ComputeCandidate(ϵ', M);
12 **for each** $e' \in C$ **do**
13 $\quad M \leftarrow M \oplus \langle e', \epsilon' \rangle$;
14 $\quad$ Enumerate(φ, $i + 1$, M);
15 $\quad M \leftarrow M \ominus \langle e', \epsilon' \rangle$;

16 **procedure** ComputeCandidate(ϵ', M)
17 $C \leftarrow \emptyset$;
18 **for each** $e' \in \mathcal{V}(\epsilon') : s(e') = 1$ **and** $e' \notin M$ **do**
19 $\quad \lambda \leftarrow 0$;
20 $\quad$ **for each** $\langle e'', \epsilon'' \rangle \in M$ **do**
21 $\quad\quad$ **if** $\epsilon' \cap \epsilon'' \neq \emptyset$ **and** $(e', e'') \notin \mathcal{E}$ **then**
22 $\quad\quad\quad \lambda \leftarrow 1$;
23 $\quad\quad\quad$ **break**;
24 $\quad$ **if** $\lambda = 0$ **then**
25 $\quad\quad C \leftarrow C \cup \{e'\}$;
26 **return** C;

Matching activation. When a new edge arrives, we may not need to execute the matching process for it, because it cannot generate any new matching results for sure. Before presenting the details, we introduce the concept of closing edge. For an edge $\epsilon \in E(Q)$, we say ϵ is a *closing* edge if there does not exist another $\epsilon' \in E(Q)$ such that $\epsilon \prec \epsilon'$. Otherwise, ϵ is *non-closing*.

Lemma 4. *Given the newly arriving edge e and its matching edge ϵ in Q, if ϵ is neither an order-free nor a closing edge, e cannot generate any matching result.*

Proof. Since ϵ is neither order-free nor closing, there must be another $\epsilon' \in E(Q)$ such that $\epsilon \prec \epsilon'$. If e is contained in a matching result, there must be another edge e' in the data graph to match ϵ' with $e \prec e'$, which contradicts to the fact that e is the latest arriving edge. Thus, e cannot contribute in an embedding. □

The algorithm FindIncrementalMatch. We start by checking if the matching edge of e is order-free or closing (Lines 1-2). If yes, we generate a matching order for edges in $E(Q)$ except ϵ (Line 3). Based on that, we enumerate all matching results using the procedure Enumerate (Line 4).

In the procedure Enumerate, we maintain a matching offset value i and a partial matching M, which are initialized as 1 and $\langle e, \epsilon \rangle$, respectively. During the processing of Enumerate, we first check if M is a full match, i.e., $|M| = |E(Q)|$. If yes, we collect a matching result and finish the current branch (Lines 7-9). Otherwise, we get the next matching edge ϵ' and calculate its candidates using ComputeCandidate (Lines 10-11). After that, we search the subspaces (Lines 12-15). More specifically, we iteratively select edges in C to form a match pair with ϵ'. At each step, we update M by adding a match pair $\langle e', \epsilon' \rangle$ and enter the new search space to consider the next matching edge (Lines 13-14). After finishing the new search space, we should remove $\langle e', \epsilon' \rangle$ from M before going to next iteration (Line 15).

In the procedure ComputeCandidate, we compute the candidate matching edges for ϵ', given the partial match M. Based on the structure of CSS, we have that the candidates must be a subset of nodes in $\mathcal{V}(\epsilon')$ with node state being 1 and not yet used in M (Line 18). For each node e', we check its connectivity with the already matched edges in M (Lines 20-23). In specific, if there exists a match pair $\langle e'', \epsilon'' \rangle \in M$, such that ϵ' and ϵ'' are adjacent while there is no link between e' and e'', it is immediate that e' cannot be a candidate. By removing the condition $e' \notin M$ in Line 18, Algorithm 4 can enumerate all subgraph homomorphisms of Q.

Example 8. *Continuing Example 7, when (v_8, v_9, l_3, t_{10}) is inserted, we illustrate the overall running process of FindIncrementalMatch. Since the matching edge ϵ_5 is a closing edge, we need to execute the matching process. Suppose we use the depth-first traverse to generate the matching order for edges in the query graph. We have that $\varphi = \{\epsilon_4, \epsilon_2, \epsilon_1, \epsilon_3\}$. Next, we first calculate the candidate for ϵ_4, which is (v_5, v_8, l_2, t_9) in $\mathcal{V}(\epsilon_4)$. Then, we find 2 candidates for ϵ_2,including $\langle (v_2, v_5, l_1, t_3), \epsilon_2 \rangle$ and $\langle (v_6, v_5, l_1, t_4), \epsilon_2 \rangle$. By continuing this processing, we can get a full match $\{\langle (v_8, v_9, l_3, t_{10}), \epsilon_5 \rangle, \langle (v_5, v_8, l_2, t_9), \epsilon_4 \rangle, \langle (v_2, v_5, l_1, t_3), \epsilon_2 \rangle, \langle (v_2, v_3, l_4, t_2), \epsilon_1 \rangle, \langle (v_3, v_5, l_5, t_5), \epsilon_3 \rangle\}$.*

Cost analysis. It is clear that FindIncrementalMatch enumerates the matches based on the widely used *depth-first paradigm* [13, 42, 50], except that it expands the partial matching by an edge at a time rather than a vertex. Next, we introduce the concept of line graph to facilitate the cost analysis [10, 19]. Given a graph G, its line graph G' is constructed by converting the edges in G to vertices in G', and connecting two vertices in G' if and only if the corresponding edges in G share a common endpoint. The line graph of the query graph in Figure 3(a) is shown in Figure 6. In the rest of cost analysis, whenever discussing a query graph Q, we refer to its line graph.

To enhance the performance, we enforce connectivity of the matching order. That is, given a spanning tree Q_T of Q, the matching order, $\varphi = (\epsilon_1, \cdots \epsilon_n)$, of Q_T is generated as follows. For each node ϵ_i except ϵ_1, its parent $\epsilon_i.p$ in Q_T is always before ϵ_i in the matching order. Note that, ϵ_1 is always the matching edge of the inserted edge e since we enforce the matching starting from e. For example, suppose the spanning tree Q_T of the query graph in Figure 6, consists of edges (ϵ_5, ϵ_4), (ϵ_4, ϵ_2),(ϵ_2, ϵ_1), and (ϵ_4, ϵ_3) as depicted by thick lines, then a possible matching order is $(\epsilon_5, \epsilon_4, \epsilon_2, \epsilon_1, \epsilon_3)$ where $\epsilon_3.p = \epsilon_4$. Here, (ϵ_1, ϵ_3) and (ϵ_2, ϵ_3) are two non-tree edges since they are not included in Q_T.

Given a specific matching order, we follow the cost model proposed in [42] to analyse the matching cost of FindIncrementalMatch, which is described as follows.

$$T_{iso} = B_1 + \Sigma_{i=2}^{n} \Sigma_{j=1}^{B_{i-1}} d_i^j (r_i + 1) \quad (1)$$

Here, B_i is the *search breadth* at depth i, denoting the number of possible embeddings in $\mathcal{G}$ (i.e., the CSS index structure) for the subgraph of Q induced by $\{\epsilon_1, \ldots, \epsilon_i\}$, d_i^j is the number of neighbor nodes of $M_{i-1}^j(\epsilon_i.p)$ in $\mathcal{V}(\epsilon_i)$ with state 1 where M_{i-1}^j is the j-th embedding in $\mathcal{G}$ for the subgraph induced by $\{\epsilon_1, \ldots, \epsilon_{i-1}\}$ and $M_{i-1}^j(\epsilon_i.p)$ is the node to which the parent $\epsilon_i.p$ of ϵ_i in Q_T maps, and r_i is the number of non-tree edges between ϵ_i and edges before ϵ_i in the matching order. Intuitively, a partial matching M_{i-1}^j of $\{\epsilon_1, \ldots, \epsilon_{i-1}\}$ in $\mathcal{G}$ is extended by mapping ϵ_i to each node e' in $\mathcal{V}(\epsilon_i)$ that is adjacent to $M_{i-1}^j(\epsilon_i.p)$, and e' is a successful mapping of ϵ_i if it satisfies all connection requirements specified by the r_i non-tree edges of ϵ_i.

Example 9. *Given the matching order $(\epsilon_5, \epsilon_4, \epsilon_2, \epsilon_1, \epsilon_3)$ of Q in Figure 6, $r_1 = r_2 = r_3 = r_4 = 0$ while $r_5 = 2$ since there are two non-trees between ϵ_3 and edges before it in the matching order, i.e., (ϵ_1, ϵ_3) and (ϵ_2, ϵ_3). $M_2^1 = \{\langle(v_8, v_9, l_3, t_{10}), \epsilon_5\rangle, \langle(v_5, v_8, l_2, t_9), \epsilon_4\rangle\}$, then the neighbors of $M_2^1(\epsilon_2.p)$ (i.e., (v_5, v_8, l_2, t_9)) are (v_2, v_5, l_1, t_3) and (v_6, v_5, l_1, t_4) both with state 1, and thus $d_3^1 = 2$.*

Remark. According to above cost analysis, the overall matching cost is influenced by the matching order. In the literature, different matching orders are proposed [7, 8, 17, 42, 44] to improve the matching performance. However, as suggested in [43], there is no clear winner. In this paper, we use the least-frequent first matching order [20, 36]. That is, among all node groups that are connected to the current partial embedding, we dynamically pick the one with the least number of nodes with state 1. Note that the matching order optimization is out of the research scope of this work.

5.2 Node Merging Optimizations

Main idea. According to Equation 1, the matching cost of FindIncrementalMatch is also determined by the search breadth B_i, which is further determined by the candidate size at each matching extension (Line 11 of Algorithm 4). This is because we need to branch on each candidate node when extending the embedding. Observe that nodes in CSS might have the same neighborhood structure, which implies that we can process them in the same way. Motivated by this observation, we develop node merging techniques to reduce the candidate size. The general idea is to merge nodes having the same neighborhood structure together as a single node. After the matching process, we can restore the matching results by simply replacing this node with all merged nodes.

More specifically, we introduce two node merging strategies, namely *index-time merging* and *search-time merging*. The index-time merging aims to merge nodes during the index building or updating stage, where the merged nodes in CSS must have exactly the same neighborhood structure. While the search-time merging is more flexible, which dynamically merges nodes as long as their neighborhood structures are the same under a search branch.

Technical details. We begin with the concept of *similar node*.

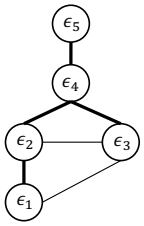


Figure 6: The line graph of the graph in Figure3(a).

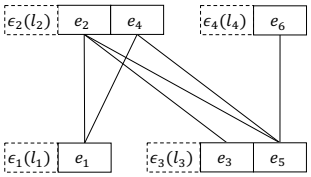


Figure 7: An example index structure CSS.

Definition 8. *Given an edge-centric index structure CSS, a set $\mathcal{N}$ of nodes in CSS are similar if*
(1) nodes in $\mathcal{N}$ are all with state 1, i.e., $\forall e \in \mathcal{N}$, $s(e) = 1$; and
(2) nodes in $\mathcal{N}$ have the same neighborhood structure, i.e., sharing the same adjacent nodes.

Apparently, if two nodes are similar to each other, they must belong to the same node group in CSS. Note that the first condition is necessary since two isolated nodes in different group satisfy the second condition as well. According to Definition 8, we have the following lemma.

Lemma 5. *Given two similar nodes $e = (u, v, l, t)$ and $e'(u', v', l', t')$, and their mapping edge ϵ in the query graph,*
(1) if ϵ is an inner edge, i.e., $d(\epsilon) > 1$, then e and e' must be the same edge appearing at different time stamps in the streaming graph, i.e., $u = u'$, $v = v'$, $l = l'$, and $t \neq t'$; and
(2) if ϵ is a leaf edge, i.e., $d(\epsilon) = 1$, then e and e' must be either the same edge appearing at different time stamps, or two adjacent edges, i.e., $u = u'$, $v \neq v'$, $l = l'$, and $t \neq t'$.

Proof. We only prove case (1), and (2) can be proved similarly. We prove by contradiction. Suppose e and e' are two different edges, e.g., $u \neq u'$ and $v = v'$. Let $\hat{e}$ be the adjacent edge of ϵ incident on the mapping vertex of u and u'. Because $s(e) = s(e') = 1$, there must exist at least one node in $\mathcal{V}(\hat{e})$ connecting e and e'. According to Lemma 2, e and e' must be adjacent to different nodes in $\mathcal{V}(\hat{e})$ since $u \neq u'$. This contradicts to the fact that similar nodes share the same adjacent nodes. □

Consider the query graph in Figure 3, where all edges except ϵ_5 are inner edges. It is easy to verify that (v_8, v_9, l_3, t_{10}) and (v_8, v_7, l_3, t_{11}) in Figure 4 are similar nodes if edge (v_8, v_7) appears again in the following time stamp t_{11}. Besides, it is also evident that Lemma 5 only gives the necessary conditions for similar nodes, rather than the sufficient conditions. That is, although an edge appears several times in the streaming graph introducing multiple instances, they might not be similar in CSS because we also consider the time constraint when adding links between nodes. Next, we show that similar nodes are replaceable in any matching result.

Lemma 6. *Given a query graph Q, let $\langle e, \epsilon\rangle$ be a match pair in a matching result M of Q. Then, M is still a correct matching result by replacing e with e' if e' is a similar node of e.*

Based on Lemma 6, for a set of similar nodes, we only need to consider one of them during the match processing, and obtain other matching results with simple node replacement. Another

nice property is that the similar relationship between nodes are permanent. That is, once two nodes are identified as similar nodes, they will always be similar as the streaming graph evolves. With this property, we come up with the following node merging strategy.

Definition 9 (**Index-time Merging**). *The index-time merging aims to merge similar nodes in CSS during the index building or updating stage.*

The index-time merging is very efficient because there is no need to maintain the merged nodes as the streaming graph evolves. However, the merging condition is very strict since it requires the similar nodes having exactly the same adjacent nodes, which might make it less effective in practice. Fortunately, we observe that although two nodes do not have the same neighborhood structure, we may still be able to merge them together during the matching search stage.

Definition 10 (**Search-time Merging**). *The search-time merging aims to merge nodes in CSS during the matching search stage, which only requires nodes having the same neighborhood structure under a specific search branch.*

Example 10. *Consider the CSS in Figure 7. Suppose all nodes are with state* 1 *and the matching order is* $\epsilon_4, \epsilon_3, \epsilon_2, \epsilon_1$. *It is clear that* e_2 *and* e_4 *cannot be merged by index-time merging. However, after obtaining the partial match* $(\langle e_6, \epsilon_4\rangle, \langle e_5, \epsilon_3\rangle)$, *we find that* e_2 *and* e_4 *have the same neighbor* e_1 *in the rest of CSS. Therefore, we can merge them together to reduce the candidate size for* ϵ_2.

Remark. It is worth mentioning that the vertex merging technique is also used in the neighbor equivalence class (NEC) based matching algorithm [18]. However, the NEC is defined on the query graph, which is orthogonal to our node-merging techniques. We remark that it is non-trivial to apply NEC-based matching algorithm to our problem as this technique is particularly designed to facilitate batch subgraph matching.

6 EXPERIMENTAL STUDY

In this section, we empirically evaluate the performance of the our proposals. All experiments are conducted on PCs with Intel Xeon 2 × 3.0GHz CPU containing 36 cores and 512GB RAM running Ubuntu 20.04.5 LTS. We run an algorithm against a single core.

6.1 Experiment Setup

Algorithms. We evaluate the following algorithms.

- **TC-Match**. Our CSS based algorithm.
- **TC-Match(PostVerify)**. TC-Match without incorporating the temporal information in CSS (i.e., Lemma 1 and Lemma 2), but using a post-processing step for verifying the timing order constraints.
- **Timing**. The SOTA for TCSM [30, 31].
- **RapidFlow**. Adapted algorithm from the SOTA for CSM [46].
- **CaLiG**. Adapted algorithm from the SOTA for CSM [53].

It is easy to verify that a match of TCSM must be a match of CSM. With this property, we can immediately come up with a baseline method as follows. First, applying the off-the-shelf CSM algorithms to list all embeddings of the query graph, and then verifying them posteriorly by considering the timing order constraints. We notice that RapidFlow [46] and CaLiG [53] are two state-of-the-art algorithms for CSM. For presentation convenience, we use the name of RapidFlow and CaLiG, respectively, to denote the adapted algorithms for our problem. We obtain the source codes of Timing, RapidFlow, and CaLiG from the authors of [31], [46], and [53], respectively. All algorithms are implemented in standard C++ with STL library support and compiled with GNU GCC.

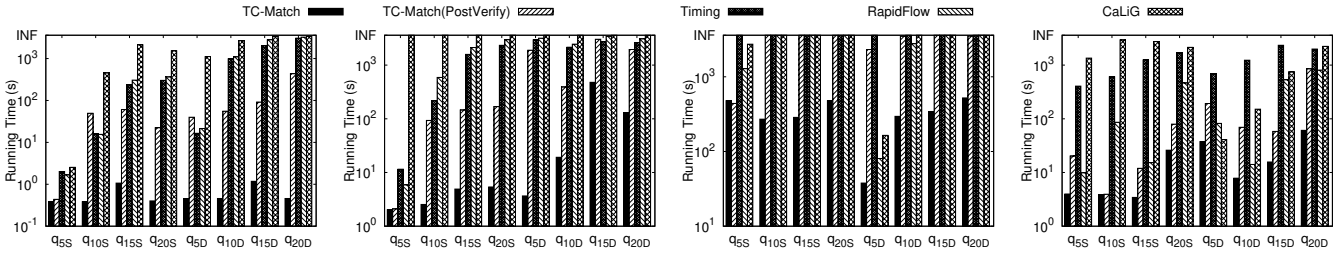
Table 2: The detailed information of datasets. $|\sum_V|$ is the number of distinct vertex labels. $|\sum_E|$ is the number of distinct edge labels. d_{avg} is the average vertex degree.

Datasets	$\|V\|$	$\|E\|$	$\|\sum_V\|$	$\|\sum_E\|$	d_{avg}
WikiTalk	1.1 M	7.8 M	1.1 M	1	13.7
CAIDA	0.9 M	29.2 M	0.07 M	2	61.8
LiveJournal	4.9 M	42.9 M	30	1	18.1
LSBench	5.2 M	54 M	101	1	20.6

Datasets. We use 4 datasets that are commonly used to evaluate existing techniques for continuous subgraph matching [36, 46, 47, 53], which covers all 3 datasets used in [30, 31] to evaluate Timing. *WikiTalk* is obtained from the Stanford SNAP library [1], where a directed edge (u, v, t) denotes that a user u edits user v's talk page at time t. We consider the user name as vertex label and network behavior "edit" as the only edge label. *CAIDA* is obtained from Anonymized Internet Traces [2], where each network communication record is a six-tuple, including the source IP/port, the destination IP/port, the protocol in use, and the communication timestamp. The vertex label and edge label are considered as the port and protocol, respectively. *LiveJournal* [46] is randomly sampled from a static graph to form the update stream, where the vertex labels are also randomly assigned from a label set. *LSBench* [27] is a synthetic dynamic social network, where each record is a five-tuple, consisting of subject type/id, predicate, object type/id. We use the subject/object's type and the predicate as vertex label and edge label, respectively. Table 2 summarizes the detailed information of the 4 datasets. Unless otherwise specified, we report the results on edge insertion for brevity. We use the first 60% edges of a dataset as initial graph and the rest 40% as graph update stream.

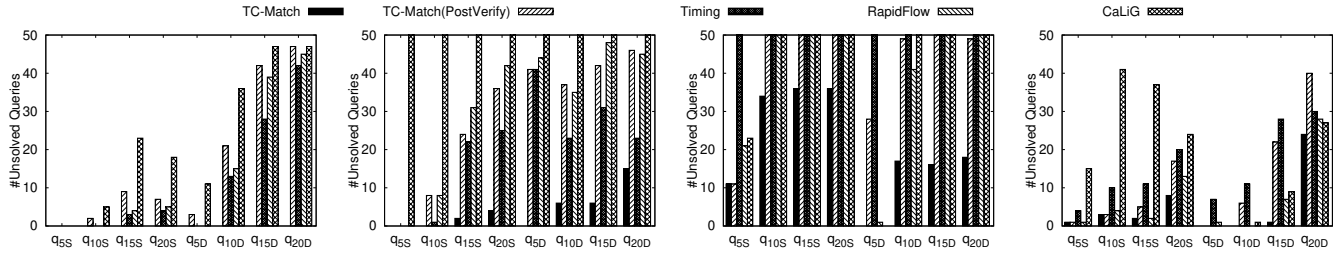
Query graphs. Following existing works [36, 46, 47], we generate query graphs as connected subgraphs of the data graph via random walks, and classify them into 2 categories based on the density, namely *sparse* ($d_{avg} \leq 3$) and *dense* ($d_{avg} > 3$). For each type, we generate graphs with 4 different sizes (i.e., number of vertices), including 5, 10, 15, and 20. We denote the 8 query sets as q_{5S}, q_{10S}, q_{15S}, q_{20S}, q_{5D}, q_{10D}, q_{15D}, and q_{20D}, where q_{iS} and q_{iD} denote query sets with i vertices and types of **S**parse and **D**ense, respectively. For each query graph, we randomly generate a collection of timing order relations without contradictions. To reduce the variance, we generate 50 queries for each query graph.

Metrics. For each testing, we run an algorithm for a query set, containing 50 query graphs. Given a query, we report the elapsed query time, consisting of time for index update and incremental match search. We terminate an algorithm if it cannot finish in 1 hour (denoted as INF) for a query graph, which is marked as an unsolved query. In the experiment, we report the average running time over 50 query graph. We also report the memory usage.



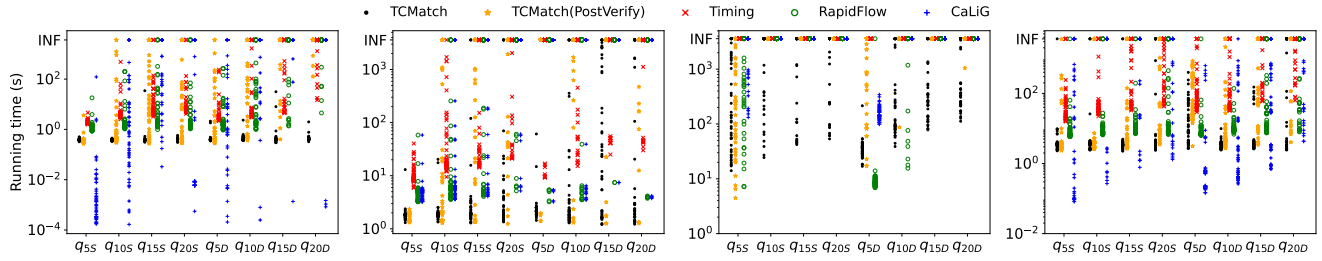
(a) *WikiTalk* (b) *CAIDA* (c) *LiveJournal* (d) *LSBench*

Figure 8: Comparison of competing algorithms on average query time.



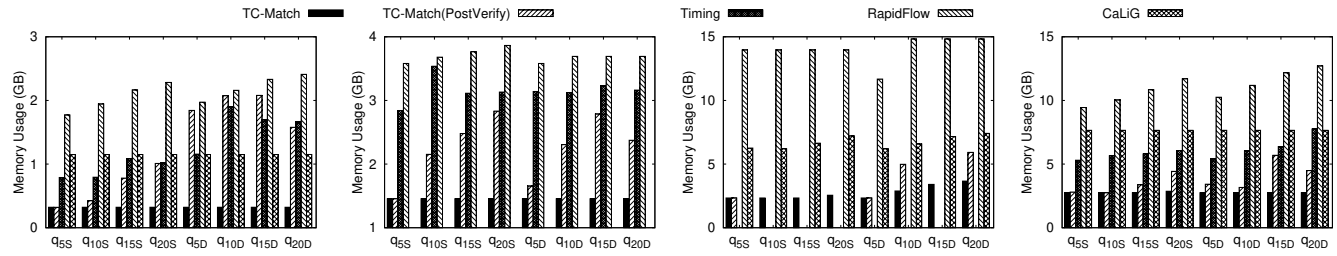
(a) *WikiTalk* (b) *CAIDA* (c) *LiveJournal* (d) *LSBench*

Figure 9: Evaluating the number of unsolved queries.



(a) *WikiTalk* (b) *CAIDA* (c) *LiveJournal* (d) *LSBench*

Figure 10: Comparison of competing algorithms on individual query time. Each dot denotes the query time of a query.



(a) *WikiTalk* (b) *CAIDA* (c) *LiveJournal* (d) *LSBench*

Figure 11: Evaluating the memory usage.

6.2 Overall Performance Comparison

Average query time. Figure 8 reports the average query time on each query graph set. Note that the query time is considered as 1 hour if an algorithm cannot finish within 1 hour. It is observed that TC-Match significantly outperforms the competitors on the majority of query settings and can achieve up to 3 orders of magnitude

speedup, e.g., on q_{10D}, q_{15D}, and q_{20D} on *WikiTalk*. For example, on this dataset, TC-Match spends less than 1 second for all query sets, while it takes the existing methods more than 200 seconds on 5 query sets, i.e., q_{15S}, q_{20S}, q_{10D}, q_{15D}, and q_{20D}. Among the three existing methods, there is no clear winner. For example, Timing outperforms RapidFlow and CaLiG on most query settings on *WikiTalk* and *CAIDA*, while it is beaten by RapidFlow with a large performance gap on all query settings on *LSBench*. The reason is that the number of incremental matches on *WikiTalk* and *CAIDA* is large, which is unfriendly to RapidFlow or CaLiG since the post-precessing step for verifying the timing order constraints is time cost-expensive. On *LSBench*, however, the number of matching results is small, which favors RapidFlow. We also notice that while TC-Match(PostVerify) is faster than the existing methods on most query settings, it is significantly outperformed by TC-Match, which demonstrates the effectiveness of the temporal information in CSS.

Unsolved queries. We count the number of unsolved queries for each algorithm in Figure 9. In general, CaLiG has more unsolved queries than others. It is interesting that CaLiG failed to process any query on *CAIDA*. This is because that *CAIDA* is vertex multiple-labelled graph, while CaLiG can only handle a single label for a vertex/edge. To work for *CAIDA*, it needs to iteratively process for each label of a vertex, which is time costly. Compared to the baseline methods, TC-Match significantly reduces the number of unsolved queries. For example, TC-Match has no unsolved queries on *WikiTalk*, and a few (less than 10) ones on *CAIDA* and *LSBench* under the majority of query settings. It is observed that *LiveJournal* is relatively hard to process because compared to other datasets, *LiveJournal* contains more query relevant edges. This also explains why Timing failed on all queries on this dataset, since it needs to materialize the intermediate results (see Section 2.2 for details).

Individual query time. Because the query time on different queries varies greatly and the average value may hide the performance of competing algorithms on each individual query, we therefore also report the individual query time in Figure 10, where a dot denotes the query time of a query. In general, TC-Match performs much more steadily than the competitors. For example, the running time of CaLiG spans evenly from 0.1 millisecond to a few minutes (or even INF) on q_{10S} and q_{5D} on *CAIDA*. This implies that CaLiG is very sensitive to the specific query because it needs to posteriorly verify the timing order constraints. Timing and RapidFlow show similar performance results.

Memory usage. Figure 11 reports the memory usage of the four algorithms. Note that we do not show the memory usage of an algorithm if its running time is INF for all queries under an experiment setting. It is observed that TC-Match consumes the least amount of memory under all experiment settings. More specifically, the memory consumption of TC-Match is reduced by 48.7%-83.1%, 60%-86.7%, and 52.4%-72.2% compared to Timing, RapidFlow, and CaLiG, respectively. This is because TC-Match uses a space cost-effective index structure CSS. Besides, TC-Match also consumes much less memory than TC-Match(PostVerify), which implies that the temporal information can substantially refine CSS by pruning the links not satisfying the timing order constraints.

Evaluating edge deletion. In this experiment, we evaluate the performance of the competing algorithms for edge deletion. The experiment results shown in Figure 12 report that TC-Match significantly outperforms the competitors under most query settings. The overall performance of the competing algorithms is rather similar to that for edge insertion as shown in Figure 8.

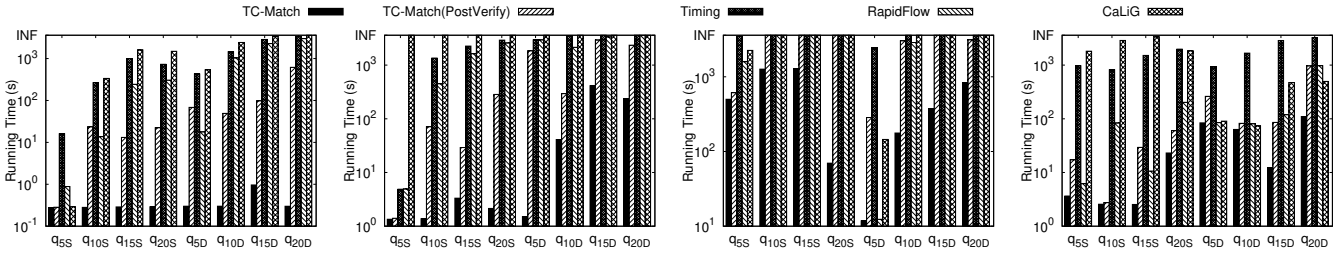
6.3 Evaluating Individual Techniques

Query time breakdown. Given a graph update, TC-Match first updates the index CSS and then enumerates the incremental matches. Figure 13 shows the query time breakdown to update time and search time on individual queries. We present the results on queries of q_{10D} on *CAIDA* and *LSBench* as representatives. Generally, the update time occupies the majority of time cost on most queries, especially on *LSBench*, which implies that our index structure CSS is very effective on filtering the unpromising edges or substructures. Nevertheless, there are also a few exceptions on *CAIDA*, due to a large number of matching results on these queries, needing more time to enumerate. It is also observed that the search time fluctuates greatly, especially on *CAIDA*. This is because the incremental matches vary on different queries.

Effectiveness of node merging. Last, we evaluate the effectiveness of node merging techniques proposed in Section 5.2. Figure 14 reports the experiment results on two representative datasets *CAIDA* and *LSBench*, where TC-Match(NSM) and TC-Match(NM) stand for TC-Match without search-time merging and without both, respectively. We observe that, by using node mergings, TC-Match can achieve more than 1 order of performance improvement, and even up to 3 orders on q_{5D} on *CAIDA*. Interestingly, the performance gap between TC-Match and TC-Match(NSM) is generally less significant, especially on q_{5S}, q_{10S}, and q_{5D} on *CAIDA*, which means that the benefit brought by search-time merging is limited under these settings. This is mainly because most similar nodes are already merged by the index-time merging, and therefore there is not enough nodes having the same neighborhood structure during the search stage. However, the search-time merging can still accelerate the query by more than 5x speedup on some queries, e.g., q_{10D} and q_{15D} on *CAIDA* and *LSBench*, respectively.

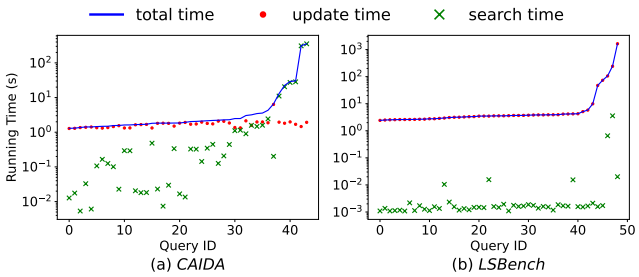
7 RELATED WORK

Batch subgraph matching. Following the previous studies [43, 46, 47], we classify the representative subgraph matching algorithms into *exploration-based* and *join-based* methods. Since first proposed by Ullmann [50], the graph exploration-based method has been extensively studied in the database community. Algorithms under this category can be further categorized by whether to use index or auxiliary structures. Ullmann [50], VF2 [13], QuickSI [42], and RI [9] directly enumerate all matches. GADDI [55], SPath [56], and SGMatch [38] facilitate the enumeration by constructing and using indices on sub-structures (e.g., paths). Recent researches, including GraphQL [20], TurboIso [18], CFL [8], CECI [7], DP-iso [17], VEQ [25], and Gup [5], boost the performance by building an auxiliary data structure for the query graph in a preprocessing step. The SOTA Circinus [23] proposes a compression-based backtracking method to share computation for repeated path in the DFS backtracking tree. In contrast to these exploration-based methods, the join-based methods conduct multi-way joins to find the matches for the query [3, 6, 33, 45, 49].



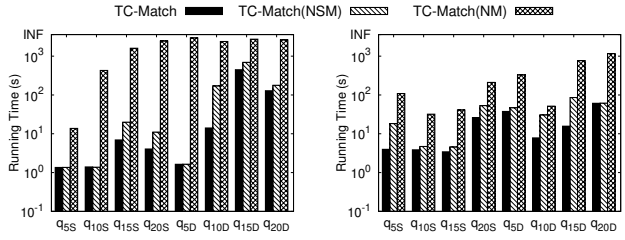
(a) *WikiTalk* (b) *CAIDA* (c) *LiveJournal* (d) *LSBench*

Figure 12: Evaluating edge deletion.



(a) *CAIDA* (b) *LSBench*

Figure 13: Query time breakdown of each individual query.



(a) *CAIDA* (b) *LSBench*

Figure 14: Evaluating node merging optimization.

Continuous subgraph matching. The first CSM algorithm IncIsoMatch [14, 15] finds the incremental matches by computing the difference of matches between the original graph and updated graph. To tackle the inefficiency of IncIsoMatch due to large touched subgraph, latest algorithms all adopt the incremental methodology. SJ-Tree [12] models a CSM query as a multi-way join and find matches with a left-deep tree. To facilitate the join process, SJ-Tree builds index to store all partial results of the join, leading to large memory usage due to the exponential number of partial results. Graphflow [24] improves efficiency by starting the join from the updated edge. However, many invalid candidates can involve in the computation. To alleviate this issue, TurboFlux [26] constructs a tree-structured index based on the spanning tree of Q, where each node contains the candidates of a query vertex. The index is maintained dynamically to keep consistent with each updated graph, and the incremental matches are enumerated starting from the updated edge in the index. symbi [36] improves the pruning power by constructing a graph-based index structure based on a directed acyclic instead of a spanning tree. The SOTA CSM algorithm RapidFlow [46] proposes dual matching to eliminate redundant computation caused by automorphisms in Q, and a matching order not necessarily starting from the updated edge. Another SOTA CSM algorithm CaLiG [53] proposes a cost-effective index structure, and a kernel-and-shell based incremental matching method. In addition, there are solutions on optimizing the processing of multiple queries [32, 52, 54]. In the literature, Hasse [48] is particularly designed for TCSM. However, this method is less generic because it is developed on a strict assumption of the query graph. This assumption is later removed by Timing [30, 31].

Temporal subgraph matching. Recently, several studies have solved the problem of temporal subgraph matching [4, 16, 28, 34, 41]. In essence, they aim to find all the time-constrained embeddings over a static temporal graph, which is orthogonal to our problem TCSM, because we need to report all incremental matches for each graph update. In the literature, the problem of durable subgraph matching on temporal graphs has also been investigated [29, 39, 40].

8 CONCLUSION

In this paper, we study the problem of time-constrained continuous subgraph matching over streaming graphs. To deal with this problem, we propose TC-Match, a novel and effective approach. First, we design a space and time cost-effective index structure CSS, which can encapsulate the partial embedding and timing order information between edges in the streaming graph. We theoretically show that CSS has polynomial space and construction time complexities. Second, we develop an efficient CSS-based matching algorithm with node merging optimizations. Extensive experiments show that TC-Match significantly outperforms the competitors by up to 3 orders of magnitude.

REFERENCES

[1] [n.d.]. http://snap.stanford.edu/data/wiki-talk-temporal.html.
[2] [n.d.]. https://www.caida.org.
[3] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. *Proceedings of the 2016 International Conference on Management of Data* (2016).
[4] Amir Pouya Aghasadeghi, Jan Van den Bussche, and Julia Stoyanovich. 2023. Temporal graph patterns by timed automata. *The VLDB Journal* (05 2023), 1–23.
[5] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. GuP: Fast Subgraph Matching by Guard-based Pruning. *Proceedings of the ACM on Management of Data* 1 (2023), 1 – 26.

[6] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015).

[7] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. *Proceedings of the 2019 International Conference on Management of Data* (2019).

[8] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and W. Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. *Proceedings of the 2016 International Conference on Management of Data* (2016).

[9] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics* 14 (2013), S13 – S13.

[10] Lei Cai, Jundong Li, Jie Wang, and Shuiwang Ji. 2020. Line Graph Neural Networks for Link Prediction. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44 (2020), 5103–5113.

[11] Zijun Cheng, Rujie Dai, Leiqi Wang, Ziyang Yu, Qiujian Lv, Yan Wang, and Degang Sun. 2023. GHunter: A Fast Subgraph Matching Method for Threat Hunting. *2023 26th International Conference on Computer Supported Cooperative Work in Design (CSCWD)* (2023), 1014–1019.

[12] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. 2015. A selectivity based approach to continuous pattern detection in streaming graphs. (2015), 157–168.

[13] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26 (2004), 1367–1372.

[14] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, and Xin Wang andYinghui Wu. 2011. Incremental graph pattern matching. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. 925–936.

[15] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental graph pattern matching. *ACM Transactions on Database Systems (TODS)* 38, 3 (2013), 1–47.

[16] Eric L. Goodman and Dirk Grunwald. 2019. Streaming Temporal Graphs: Subgraph Matching. *2019 IEEE International Conference on Big Data (Big Data)* (2019), 4977–4986.

[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. *Proceedings of the 2019 International Conference on Management of Data* (2019).

[18] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 International Conference on Management of Data*.

[19] Frank Harary and R. Z. Norman. 1960. Some properties of line digraphs. *Rendiconti del Circolo Matematico di Palermo* 9 (1960), 161–168.

[20] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 International Conference on Management of Data*.

[21] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. *Proceedings of the 2017 ACM International Conference on Management of Data* (2017).

[22] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2019. General dynamic Yannakakis: conjunctive queries with theta joins under updates. *The VLDB Journal* 29 (2019), 619–653.

[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. *Proceedings of the ACM on Management of Data* 1 (2023), 1 – 26.

[24] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1695–1698.

[25] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile Equivalences: Speeding up Subgraph Query Processing and Subgraph Matching. *Proceedings of the 2021 International Conference on Management of Data* (2021).

[26] 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 *Proceedings of the 2018 International Conference on Management of Data*. 411–426.

[27] Danh Le-Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter A. Boncz, Thomas Eiter, and Michael Fink. 2012. Linked Stream Data Processing Engines: Facts and Figures. In *International Workshop on the Semantic Web*.

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

[29] Faming Li, Zhaonian Zou, and Jianzhong Li. 2023. Durable Subgraph Matching on Temporal Graphs. *IEEE Transactions on Knowledge and Data Engineering* 35 (2023), 4713–4726.

[30] Youhuan Li, Lei Zou, M. Tamer Özsu, and Dongyan Zhao. 2019. Time Constrained Continuous Subgraph Search Over Streaming Graphs. In *Proceedings of the2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1082–1093.

[31] Youhuan Li, Lei Zou, M. Tamer Ozsu, and Dongyan Zhao. 2022. Space-Efficient Subgraph Search Over Streaming Graph With Timing Order Constraint. *IEEE Transactions on Knowledge and Data Engineering* 34 (2022), 4453–4467.

[32] Amine Mhedhbi, Chathura Kankanamge, and Semih Salihoglu. 2021. Optimizing one-time and continuous subgraph queries using worst-case optimal joins. *ACM Transactions on Database Systems (TODS)* 46, 2 (2021), 1–45.

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

[34] Giovanni Micale, Giorgio Locicero, Alfredo Pulvirenti, and Alfredo Ferro. 2021. TemporalRI: subgraph isomorphism in temporal networks with multiple contacts. *Applied Network Science* 6 (2021), 1–22.

[35] Sadegh M. Milajerdi, Birhanu Eshete, Rigel Gjomemo, and Venkat Venkatakrishnan. 2019. POIROT: Aligning Attack Behavior with Kernel Audit Records for Cyber Threat Hunting. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019).

[36] 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. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1298–1310.

[37] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time Constrained Cycle Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 11 (2018), 1876–1888.

[38] Carlos R. Rivero and Hasan M. Jamil. 2017. Efficient and scalable labeled subgraph matching using SGMatch. *Knowledge and Information Systems* 51 (2017), 61–87.

[39] Konstantinos Semertzidis and Evaggelia Pitoura. 2016. Durable graph pattern queries on historical graphs. *2016 IEEE 32nd International Conference on Data Engineering (ICDE)* (2016), 541–552.

[40] Konstantinos Semertzidis and Evaggelia Pitoura. 2019. Top-k Durable Graph Pattern Queries on Temporal Graphs. *IEEE Transactions on Knowledge and Data Engineering* 31 (2019), 181–194.

[41] Konstantinos Semertzidis and Evaggelia Pitoura. 2020. A Hybrid Approach to Temporal Pattern Matching. *2020 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)* (2020), 384–388.

[42] 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 (2008), 364–375.

[43] Shixuan Sun and Qiong Luo. 2020. In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1083–1098.

[44] Shixuan Sun and Qiong Luo. 2022. Subgraph Matching With Effective Matching Order and Indexing. *IEEE Transactions on Knowledge and Data Engineering* 34 (2022), 491–505.

[45] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: A Holistic Approach to Subgraph Query Processing. *Proc. VLDB Endow.* 14 (2020), 176–188.

[46] Shixuan Sun, Xibo Sun, Bingsheng He, and Qiong Luo. 2022. RapidFlow: An Efficient Approach to Continuous Subgraph Matching. *Proceedings of the VLDB Endowment* (2022).

[47] Xibo Sun, Shixuan Sun, Qiong Luo, and Bingsheng He. 2022. An in-depth study of continuous subgraph matching. *Proceedings of the VLDB Endowment* 15, 7 (2022), 1403–1416.

[48] Xiaoli Sun, Yusong Tan, Q. Wu, and Jing Wang. 2017. Hasse diagram based algorithm for continuous temporal subgraph query in graph stream. *2017 6th International Conference on Computer Science and Network Technology (ICCSNT)* (2017), 241–246.

[49] Ha Nguyen Tran, Jung jae Kim, and Bingsheng He. 2015. Fast Subgraph Matching on Large Graphs using Graphics Processors. In *International Conference on Database Systems for Advanced Applications*.

[50] Julian R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *Journal of the ACM (JACM)* 23 (1976), 31 – 42.

[51] Shihan Wang and Takao Terano. 2015. Detecting rumor patterns in streaming social media. *2015 IEEE International Conference on Big Data (Big Data)* (2015), 2709–2715.

[52] Xi Wang, Qianzhen Zhang, Deke Guo, and Xiang Zhao. 2022. Continuous multi-query optimization for subgraph matching over dynamic graphs. *Semantic Web* 13 (2022), 601–622.

[53] Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. 2023. Fast Continuous Subgraph Matching over Streaming Graphs via Backtracking Reduction. *Proceedings of the ACM on Management of Data* 1 (2023), 1 – 26.

[54] Lefteris Zervakis, Vinay Setty, Christos Tryfonopoulos, and Katja Hose. 2020. Efficient continuous multi-query processing over graph streams. (2020), 13–24.

[55] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: distance index based subgraph matching in biological networks. In *International Conference on Extending Database Technology*.

[56] Peixiang Zhao and Jiawei Han. 2010. On graph query optimization in large networks. *Proceedings of the VLDB Endowment* 3 (2010), 340 – 351.