An In-Depth Study of Continuous Subgraph Matching

Xibo Sun
Hong Kong University of Science and Technology
xsunax@cse.ust.hk

Shixuan Sun*
National University of Singapore
sunsx@comp.nus.edu.sg

Qiong Luo
Hong Kong University of Science and Technology
luo@cse.ust.hk

Bingsheng He
National University of Singapore
hebs@comp.nus.edu.sg

ABSTRACT

Continuous subgraph matching (CSM) algorithms find the occurrences of a given pattern on a stream of data graphs online. A number of incremental CSM algorithms have been proposed. However, a systematical study on these algorithms is missing to identify their advantages and disadvantages on a wide range of workloads. Therefore, we first propose to model CSM as incremental view maintenance (IVM) to capture the design space of existing algorithms. Then, we implement six representative CSM algorithms, including IncIsoMatch, SJ-Tree, Graphflow, IEDyn, TurboFlux, and SymBi, in a common framework based on IVM. We further conduct extensive experiments to evaluate the overall performance of competing algorithms as well as study the effectiveness of individual techniques to pinpoint the key factors leading to the performance differences. We obtain the following new insights into the performance: (1) existing algorithms start the search from an edge in the query graph that maps to an updated data edge, potentially leading to many invalid partial results; (2) all matching orders are based on simple heuristics, which appear ineffective at times; (3) index updates dominate the query time on some queries; and (4) the algorithm with constant delay enumeration bears significant index update cost. Consequently, no algorithm dominate the others in all cases. Therefore, we give a few recommendations based on our experiment results. In particular, the SymBi index is useful for sparse queries or long running queries. The matching orders of IEDyn and TurboFlux work well on tree queries, those of Graphflow on dense queries or when both query and data graphs are sparse, and otherwise, we recommend SymBi's matching orders.

*Shixuan Sun is the corresponding author.

1 INTRODUCTION

Subgraph matching (SM) is a fundamental operation in graph analysis, finding all matches of a query graph Q in a data graph G. Extensive studies have been conducted on static graphs, utilizing pruning strategies, determining query plans, and proposing auxiliary data structures to improve the performance [50]. In contrast, as many real-world graphs change over time, *continuous subgraph matching* (CSM) reports matches in a graph stream. Specifically, for each graph update ΔG in the stream, CSM finds positive or negative matches (called *incremental matches*) on insertion or deletion of edges. In this paper, we study the problem of exact CSM. For example, $\{(u_0, v_0), (u_1, v_4), (u_2, v_5), (u_3, v_8)\}$ is a match given Q and G in Figure 1. When the edge $e(v_6, v_{10})$ is inserted to G in Figure 1c, a positive match $\{(u_0, v_2), (u_1, v_6), (u_2, v_5), (u_3, v_{10})\}$ occurs in G'. After that, $e(v_0, v_4)$ is deleted from G' in Figure 1d. A negative match $\{(u_0, v_0), (u_1, v_4), (u_2, v_5), (u_3, v_8)\}$ disappears in G''.

(a) Query graph Q.　　**(b) Data graph G.**

(c) G': Insert $e(v_6, v_{10})$ into G.　　**(d) G'': Delete $e(v_0, v_4)$ from G'.**
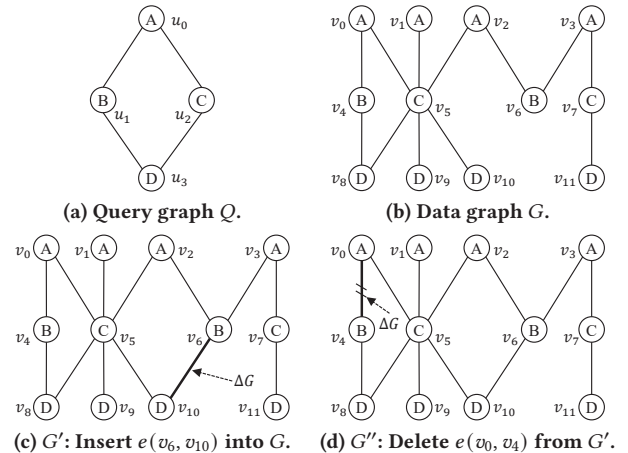
Figure 1: Example graphs.

The topic of CSM is essential in various scenarios. For instance, social network providers detect the spread of rumors among users by matching rumor patterns in the streams of message transmission graphs [56]. Similarly, financial administrators monitor cyclic patterns in transaction graphs for suspected money laundering offenses [45]. Another example is that IT departments continuously analyze system logs that record communications between computers to detect system anomalies [36].

A number of exact CSM algorithms have been proposed, including IncIsoMatch [19], SJ-Tree [16], Graphflow [30], IEDyn [26, 27], TurboFlux [33], and SymBi [39]. To find incremental matches efficiently, these algorithms all start the execution from the update

edge in the data graph and recursively enumerate results by mapping a *query vertex* (i.e., a vertex in Q) to a *data vertex* (i.e., a vertex in G) at a step. They further proposed a variety of techniques to accelerate the enumeration and compared with previous algorithms in experiments. Figure 2 summarizes the timeline of these algorithms in previous experiments in the literature [16, 19, 26, 27, 30, 33, 39].
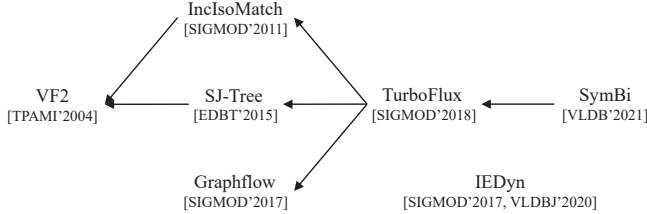

Figure 2: Comparison of algorithms in previous experiments. An edge from A to B represents that the experiments of A involved B. VF2 denotes a re-computation method using VF2 [17], a classical SM algorithm.

We make the following observations on existing work. First, no prior work has proposed a common framework to abstract the CSM problem and identify key performance factors. Second, existing work focused on evaluating the overall performance, but not individual strategies. Third, even though previous experiments have good coverage of algorithms (i.e., there is a path from the latest algorithm to other algorithms in Figure 2), there is no comprehensive comparison between all these algorithms.

Our Work. We propose to conduct an in-depth study on sequential algorithms on exact CSM. We first model CSM as the *incremental view maintenance* (IVM) problem to capture the design space of existing algorithms. Figure 3 illustrates the general idea. Given a query graph Q, the data graph G is the base data. $\mathcal{M}$ is the set of matches in G, which is materialized. IVM aims to keep $\mathcal{M}$ consistent with the base data by computing and applying the incremental change $\Delta\mathcal{M}$ incurred by the base data update. A subtle difference from IVM is that CSM computes $\Delta\mathcal{M}$, but does not store $\mathcal{M}$.

Based on this IVM model, we have three categories of existing CSM methods. The first kind is the *recomputation-based* method, illustrated by 1.1-1.4 in Figure 3. Given ΔG on G, this algorithm first computes the set $\mathcal{M}$ of matches in G, and then obtains G' by applying ΔG to G. It next finds the set $\mathcal{M}'$ of matches in G', and finally gets $\Delta\mathcal{M}$ by computing the difference between $\mathcal{M}$ and $\mathcal{M}'$. It finds $\mathcal{M}$ and $\mathcal{M}'$ with SM algorithms, which generally enumerate matches with the assistance of a lightweight index.

The other two kinds of methods are incremental. Specifically, the *direct-incremental* method, illustrated by 2.1 in Figure 3, computes $\Delta\mathcal{M}$ by directly searching matches containing edges in ΔG from the data graph. In contrast, the *index-based incremental* method, shown by 3.1-3.2 in Figure 3, maintains a lightweight index $\mathcal{A}$ on which we can enumerate all matches of Q in G. Given ΔG, it will first update $\mathcal{A}$ by computing and applying incremental changes $\Delta\mathcal{A}$ to $\mathcal{A}$, and then enumerate $\Delta\mathcal{M}$ based on the index. Thus, the lightweight index $\mathcal{A}$ is also a materialized view.

Based on our IVM model, we design a common framework for CSM and study six representative sequential algorithms on exact CSM, including IncIsoMatch, SJ-Tree, Graphflow, IEDyn, TurboFlux, and SymBi, within the framework. These algorithms mainly differ in

the method of building the index (if any) and the method optimizing the matching order, so we introduce these algorithms regarding indexing and matching orders, and give an in-depth comparison and analysis. Moreover, we compare the indexing methods with those in SM algorithms and discuss their commonalities and differences.
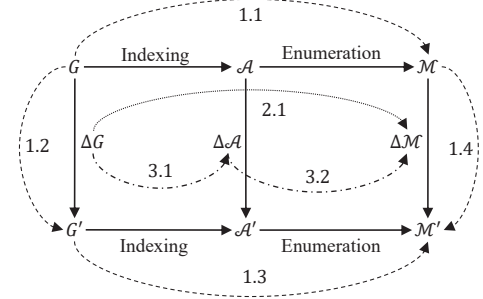

Figure 3: Modeling continuous subgraph matching as incremental view maintenance. 1.1-1.4 show the execution flow of a recomputation-based method (IncIsoMatch); 2.1 shows that of a direct-incremental algorithm (Graphflow); and 3.1-3.2 shows that of index-based incremental algorithms (SJ-Tree, IEDyn, TurboFlux and SymBi).

For a fair comparison, we re-implement the six algorithms in C++ within our framework and optimize them with our best efforts. This choice is because programming languages (Java or C++) as well as graph data structures (CSR, adjacency arrays, or others) affect the empirical performance. Moreover, only SJ-tree's source code is publicly available. With our implementation, We first evaluate the overall performance of competing algorithms in *query time*, i.e., the elapsed time of answering a query given a graph stream, and the number of *unsolved queries*, i.e., queries that cannot be completed within a time limit (one hour in our experiments). Moreover, we count the number of candidates in the index. Through experiments, we have the following results consistent with previous work.

(1) The recomputation-based method is much slower than incremental methods [33].
(2) SJ-Tree runs out of memory in most cases due to the caching of partial results [33].
(3) SymBi is more stable than the other algorithms (i.e., it has fewer unsolved queries) [39].
(4) The index reduces the number of data vertices involved in computation [33, 39], and the pruning power of the index in SymBi is stronger than that of TurboFlux [39].

However, we also find some new results: 1) the latest algorithms do not consistently outperform the old ones, 2) the algorithm with constant delay enumeration bears significant index update cost, and 3) the direct-incremental method runs faster than the index-based method in some cases. Specifically,

(1) On tree queries, IEDyn and TurboFlux slightly outperform SymBi, and all of them run much faster than Graphflow.
(2) On sparse cyclic queries, both Graphflow and SymBi run faster than TurboFlux, and Graphflow performs better than SymBi on data graphs without dense substructures.
(3) On dense cyclic queries, Graphflow generally outperforms both TurboFlux and SymBi.

(4) If edges are updated at the sparse regions in the data graph, Graphflow runs faster than IEDyn, TurboFlux, and SymBi.

Motivated by these new findings, we conduct more detailed experiments to pinpoint the factors leading to the performance differences among competing algorithms. As the algorithms mainly differ in the indexes and matching orders, we evaluate their effectiveness individually.

Particularly, given a matching order, we compare the query performance using different indexes to evaluate *the effectiveness of the index*. Then, given the same index, we compare the query performance using different matching orders to study *the effectiveness of the matching order*. Based on the results, we conclude *both are the key factors leading to the performance differences*. Furthermore, in the experiments, we observe that some queries have a long running time; even worse, some queries cannot be completed within the time limit by any algorithms. Therefore, we collect detailed metrics (e.g., the number of results and invalid partial results) and perform case studies to answer the question: *Where did time go in these queries?* Based on our extensive experiments, we give a recommendation of algorithms on different workloads and discuss issues in existing algorithms.

2 BACKGROUND

2.1 Preliminaries

This paper focuses on the undirected edge- and vertex-labeled graph $g = (V, E)$ where V is a set of vertices and E is a set of edges. We also use $V(g)$ and $E(g)$ to denote the vertex set and edge set of graph g. Given $u \in V$, $N(u)$ denotes the neighbors of u, i.e., the vertices adjacent to u. $d(u)$ is the degree of u, i.e., $d(u) = |N(u)|$. L_V (resp. L_E) is the function mapping from a vertex $u \in V$ (resp. an edge $e \in E$) to a label l in a label set Σ_V (resp. Σ_E). We use L to denote both mappings for brevity. Q and G denote the query graph and the data graph, respectively. We use $e(u, u')$ to denote the edge between the vertices u and u'. We call vertices and edges of Q (resp. G) query vertices and query edges (resp. data vertices and data edges).

Definition 1 defines *subgraph isomorphism* and *subgraph homomorphism*. The difference between them is that subgraph homomorphism allows duplicate vertices in a result, while subgraph isomorphism does not. We call a subgraph isomorphism (or homomorphism) a *match* for short. *Subgraph matching* (SM) finds all matches $\mathcal{M}$ of Q in G.

Definition 1. Given graphs g and g', a subgraph isomorphism (resp. homomorphism) is an **injective function** (resp. **mapping**) $M: V(g) \rightarrow V(g')$ such that 1) $\forall u \in V(g), L(u) = L(M(u))$; 2) $\forall e(u, u') \in E(g), L(e(u, u')) = L(e(M(u), M(u')))$; and 3) $\forall e(u, u') \in E(g), e(M(u), M(u')) \in E(g')$.

The data graph G is dynamic in this paper. $\Delta\mathcal{G}$ is a sequence of graph update operations $(\Delta G_1, \Delta G_2, ...)$ on G where $\Delta G = \{\Delta e\}$ is a set of edge insertions/deletions and $\Delta e = (+/-, e)$ is the insertion/deletion of an edge e. Let G' be the graph after applying the update ΔG on G. The *incremental matches* $\Delta\mathcal{M}$ are the difference between $\mathcal{M}$ and $\mathcal{M}'$ where $\mathcal{M}$ and $\mathcal{M}'$ represent the matches of Q in G and G', respectively. We call the results newly appearing the

positive matches, and that disappearing the *negative matches*. We define the *continuous subgraph matching* problem as follows.

Problem Statement. Given Q, G and $\Delta\mathcal{G}$, continuous subgraph matching (CSM) finds incremental matches $\Delta\mathcal{M}$ for each $\Delta G \in \Delta\mathcal{G}$.

In the context of both subgraph isomorphism and homomorphism, SM and CSM are NP-hard [19]. A special case is that SM and CSM are tractable for *acyclic queries* (i.e., query graphs with no cycle) under the setting of subgraph homomorphism [57]. Since subgraph isomorphism can be obtained by checking the injective constraint on subgraph homomorphism results, we use subgraph homomorphism as the default setting to illustrate the algorithms, following previous work. Nevertheless, we conduct experiments in both subgraph homomorphism and subgraph isomorphism settings. We assume that all $\Delta e \in \Delta G$ have the same operation (either insertion or deletion) for ease of presentation. ΔG represents the set of edges in the graph update, and $\Delta G.op$ denotes the operation. Given ΔG has mixed operations, we can handle ΔG as follows: 1) separate ΔG into ΔG_+ and ΔG_- based on the operation; 2) delete edges e from ΔG_+ and ΔG_- if it exists in both of them; and 3) first evaluate the query given ΔG_- and then evaluate it given ΔG_+. Thus, the assumption does not break the generality of our work.

Graph Data Model. We generally model the graph data frequently updated as *dynamic graphs* or *streaming graphs* [42]. Dynamic graphs model the graph data as a sequence of update operations on an initial data graph. In contrast, streaming graphs view the data as a sequence of update operations and consider the graph constructed by the updates within a sliding window. The window can be defined by a time-constraint or count-constraint. The old edges will be deleted with the sliding of the window. Furthermore, the models can be further divided into *single-update* ($|\Delta G| = 1$) based and *batch-update* ($|\Delta G| > 1$) based according to the number of edges in ΔG [18]. Existing CSM algorithms generally use the dynamic graph model with single-update [16, 19, 26, 27, 30, 33, 39].

Multi-way Join. Given Q and G, we can model the SM problem as a multi-way join $Q = \bowtie_{e(u_x, u_y) \in E(Q)} R(u_x, u_y)$ where each vertex in $V(Q)$ corresponds to an attribute, each edge $e(u_x, u_y) \in E(Q)$ corresponds to a relation $R(u_x, u_y)$ and $R(u_x, u_y) = \{e(v_x, v_y) \in E(G) | L(u_x) = L(v_x) \wedge L(u_y) = L(v_y) \wedge L(e(u_x, u_y)) = L(e(v_x, v_y))\}$ [1, 38, 51]. Worst-case optimal join (WCOJ) [41] is a class of join algorithms whose running time matches the largest query result size. The latest algorithms [1, 38, 51] evaluate Q with WCOJ, which extends partial results by vertex-at-a-time, instead of the traditional binary join, which extends partial results by edge-at-a-time.

As shown in Algorithm 1, to enumerate all matches, WCOJ extends partial results by mapping a query vertex to a data vertex along an order of query vertices. Algorithm 1 presents the sketch

---

Algorithm 1: Evaluating Q by vertex-at-a-time

1 **Procedure** Enumerate(φ, M, i)
2   **if** $i = |\varphi| + 1$ **then** Output M, **return**;
3   **else if** $i = 1$ **then** $u \leftarrow \varphi[i], C_M(u) \leftarrow \bigcap_{u' \in N_-^\varphi(u)} \pi_{\{u\}} R(u, u')$;
4   **else** $u \leftarrow \varphi[i], C_M(u) \leftarrow \bigcap_{u' \in N_+^\varphi(u)} R(u' : M(u'), u)$;
5   **foreach** $v \in C_M(u)$ **do**
6   **if** v *is not visited* **then**
7   Add (u, v) to M;
8   Enumerate($\varphi, M, i + 1$);
9   Remove (u, v) from M;

of the procedure in the context of SM [51]. φ denotes the *matching order*, which is a permutation of query vertices. $N_+^\varphi(u)$ (resp. N_-^φ) denotes the neighbors of u positioned before (resp. after) u in φ. M is a mapping from query vertices to data vertices and i records the recursion depth. Initially, $M = \{\}$ and $i = 1$. If all query vertices are mapped to data vertices, then output a result and return. Given M, we first compute the candidates $C_M(u)$ of the next query vertex u in φ (Line 2). If u is the first vertex, then $C_M(u)$ contains the common values of u in relations that contain u (Line 3). Otherwise, $C_M(u)$ contains the common neighbors of data vertices mapped to query vertices $u' \in N_+^\varphi(u)$ where $M(u')$ denotes the data vertex mapped to u' in M and $R(u' : M(u'), u)$ represents the neighbors of $M(u')$ in the relation $R(u', u)$ (Line 4). After that we loop over $C_M(u)$ to map a query vertex to a data vertex and continue the search (Lines 5-9). If we want to find subgraph homomorphisms, we can remove the check at Line 6. To avoid Cartesian product in the enumeration, the matching order φ is generally *connected*, i.e., $N_+^\varphi(u) \neq \emptyset$ given $u \in V(Q)$ except the first vertex in φ.

Incremental View Maintenance (IVM). Materialized views are queries whose results are stored [46]. We can use them to speed up query evaluation instead of computing from scratch. *Incremental view maintenance* keeps materialized views consistent with the base data by computing and applying the incremental changes incurred by the base data update [21]. The topic is well studied in relational databases for queries with different relational operators such as select-project-join [15]. Given a multi-way join $Q = \bowtie_{i \in [1,n]} R_i$, ΔR_i denotes the changes on the relation R_i, R_i' is the relation before applying the changes, and R_i is the relation after applying the changes. The incremental results of Q (i.e., the difference between $\bowtie_{i \in [1,n]} R_i'$ and $\bowtie_{i \in [1,n]} R_i$) can be computed by Equation 1 [10, 21, 22]. Join-based CSM methods [2, 30, 37] obtain Rs and ΔRs directly from the data graph G. In comparison, index-based algorithms [26, 27, 33, 39] apply pruning rules to filter the relations, store the results into indexes, and then obtain Rs and ΔRs from the indexes rather than from G.

$$
\begin{aligned}
\Delta Q_1 &= \Delta R_1 \bowtie R_2' \bowtie \dots \bowtie R_n', \\
\Delta Q_i &= R_1 \bowtie \dots \bowtie R_{i-1} \bowtie \Delta R_i \bowtie R_{i+1}' \bowtie \dots \bowtie R_n', \\
\Delta Q_n &= R_1 \bowtie \dots \bowtie R_{n-1} \bowtie \Delta R_n, \\
\Delta Q &= \bigcup_i \Delta Q_i.
\end{aligned}
\tag{1}
$$

Constant Delay Enumeration. Given Q, G, and $\Delta \mathcal{G}$ an algorithm finds $\mathcal{M}$ (or $\Delta \mathcal{M}$) in two phases: the preprocessing phase and the enumeration phase. In the latter, the algorithm reports each match in $\mathcal{M}$ (or $\Delta \mathcal{M}$) without repetition, followed by the end-of-enumeration message EOE. If the maximum time between the start of the enumeration phase and the output of the first match, between the output of two consecutive matches, and between the output of the last match and EOE only depends on the size of Q, then the enumeration is a *constant delay* enumeration. Researchers proved that there exist an algorithm for *acyclic queries* (i.e., query graphs with no cycle) under the setting of graph homomorphism which enumerates $\mathcal{M}$ with constant delay after $O(|V(Q)||E(G)|)$ time preprocessing [5, 6].

2.2 Related Work

Subgraph Matching. Subgraph matching, which finds all matches of Q of G, has been widely studied. Ullmann [55] first proposed a graph exploration-based backtracking approach. Then, various approaches are presented [12, 17, 47, 48, 59, 61] to reduce the overall matching cost. The most recent algorithms [7, 8, 23, 24, 32] summarized the data graph into an auxiliary data structure, which helps generate the query plan and enumerate subgraph matches. In contrast, another approach is to regard the graph as a database where attributes and relations correspond to vertices and edges and to conduct multi-way joins to find all matches [1, 3, 38, 50, 51, 54]. Nevertheless, both exploration-based and join-based algorithms follow the procedure in Algorithm 1 to find all matches [51].

Continuous Subgraph Matching. In addition to CSM algorithms targeting arbitrary queries, a collection of studies conduct continuous subgraph matching on specific query types. CASQD [40] was proposed to find cliques and stars in graph streams, and Qiu et al. [45] presented GraphS that detects cycles. Subgraph isomorphism may be too restrictive for some applications, and therefore, Wang et al. [14] proposed to answer approximate subgraph containment queries that allow false-positive results. Additionally, Fan et al. [19] and Song et al. [49] worked on graph simulation [25], which relaxes the injective constraint. Other researchers found matches on the graph summarization instead of the data graph [53, 60]. Additionally, Ammer et al. [2] and Gao et al. [20] detect patterns in the distributed environment.

Multi-query Optimization for Subgraph Matching. Several recent papers study the problem of matching a group of query graphs at one time. Specifically, Pugliese et al. [44] utilized a merged view of multiple query graphs to update results incrementally. Zervakis et al. [58] designed a query graph clustering algorithm to handle a large number of continuous queries. Mhedhbi et al. [37] proposed a general greedy optimizer to share computation among multiple instances of continuous queries. Currently, we focus on continuous subgraph matching for one query graph at a time.

Subgraph Counting in Graph Streams. In some applications, it is not necessary to explicitly enumerate all the subgraphs. Therefore, some work only counts the number of occurrences of the subgraph. Pavan et al. [43], Jha et al. [28], and Kara et al. [31] proposed to count triangles in graph streams, while Manjunath et al. [35] focused on counting cycles of a given length. For counting arbitrary subgraphs, Kane et al. [29] proposed a novel method based on random vector, and Assadi et al. [4] designed an efficient algorithm adopting edge sampling.

3 A GENERIC MODEL

To have a systematical study on existing continuous subgraph matching (CSM) algorithms and identify their advantages and disadvantages on a wide range of workloads, we attempt to develop a methodology to have common abstractions among different CSM algorithms. We find that existing CSM algorithms can be modelled as well studied incremental view maintenance (IVM) problems in relational databases [15]. Specifically, given Q and G, we can find all matches $\mathcal{M}$ with the multi-way join Q, and Q can be regarded as a materialized view over the base data G. Thus, given Q and G, and $\Delta G \in \Delta \mathcal{G}$, the CSM problem is equivalent to maintain the

materialized view incrementally and the incremental matches $\Delta\mathcal{M}$ can be computed by Equation 1 (i.e., $\Delta\mathcal{M} = \Delta Q$). A subtle difference is that IVM maintains all results of Q incrementally by updating $\mathcal{M}$ with $\Delta\mathcal{M}$, while CSM targets at $\Delta\mathcal{M}$ only. In the following, we present a common framework based on IVM for different CSM algorithms. In Section 4, we use the common framework to compare the similarities and differences among the six algorithms under study.

As ΔG is relatively small compared with G, using $E(G)$ in Q can result in many *dangling tuples* (i.e., the tuples do not appear in any join results). Therefore, we introduce an indexing phase before the enumeration. Particularly, the indexing phase is to prune relations without breaking their completeness (Definition 2). We use $\mathcal{A}$ to denote the set of relations after pruning. However, the indexing phase also comes with overhead in its maintenance. The relations in $\mathcal{A}$ are also materialized views of G. We must update it to keep its completeness with respect to each data graph snapshot. To the end, the query evaluation is divided into two phases: 1) update $\mathcal{A}$ given the graph update; and 2) enumerate incremental results using $\mathcal{A}$.

Definition 2. Given Q, G, and a query edge $e(u, u') \in E(Q)$, the relation corresponding to $e(u, u')$, $R(u, u')$, is a *complete relation* if it contains all data edges $e(v, v') \in E(G)$ such that there exists a match of Q in G which maps $e(u, u')$ to $e(v, v')$.

Algorithm 2 presents a common framework of CSM based on IVM. Given Q and G, we first generate an initial matching order φ_0 used in the indexing phase (Line 1). Then, we build an index $\mathcal{A}$ in the offline processing stage (Line 2). In online processing, we process the insertion by applying ΔG to G, updating the index, and finding incremental matches (Lines 4-7). In contrast, we handle the deletion in the reverse order (Lines 8-11). Because positive matches appear in the updated graph and index, while negative matches only exist in the data graph and index before the update. Intuitively, we find incremental results by enumerating the matches containing edges in ΔG (Lines 12-19). We first relabel the query edges from 1 to n, where n is the number of query edges. For each relation R_i, we compute a subset ΔR_i containing only those edges updated (Line 13-14). Then, we find results based on Equation 1 (Lines 15-18). Given $i \in [k + 1, n]$, ΔQ_k uses $R_i - \Delta R_i$ to avoid reporting duplicate matches (Line 17). We evaluate ΔQ_k using Algorithm 1, or traditional binary joins. Finally, we output $\Delta\mathcal{M}$ as positive matches if $\Delta G.op$ is an insertion operation or negative matches otherwise (Line 19). For direct-incremental methods, we ignore the execution of Lines 2, 6, and 10. For recomputation-based methods, we use Q instead of ΔQ to find matches in the data graph both before and after the update and get the incremental matches by computing the difference. Proposition 1 shows the correctness of our framework. We put the proof of Proposition 1 and an example of the common framework in our full paper [52].

PROPOSITION 1. *The incremental matches output by our framework are correct. Namely, on each update, the $\Delta\mathcal{M}$ reported by Algorithm 2 is exactly the difference of $\mathcal{M}$ before and after the update.*

From the common framework, we can capture the key performance factors for CSM: 1) the efficiency and effectiveness of index, which determines the maintenance cost of $\mathcal{A}$ and the relation cardinality in ΔQ_k; and 2) the effectiveness of the join plan evaluating ΔQ_k. Existing algorithms mainly differ in the optimization of the

---

Algorithm 2: A Common Framework of CSM using IVM

Input: a query graph Q, a data graph G, an update stream $\Delta\mathcal{G}$
Output: incremental matches $\Delta\mathcal{M}$ for each $\Delta G \in \Delta\mathcal{G}$
/* Offline preprocessing   */
1 Generate an initial matching order φ_0;
2 $\mathcal{A} \leftarrow \text{BuildInitialIndex}(Q, G, \varphi_0)$;
   /* Online processing   */
3 **foreach** $\Delta G \in \Delta\mathcal{G}$ **do**
4   **if** $\Delta G.op$ *is* + **then**
5   Apply ΔG to G;
6        $\text{UpdateIndex}(Q, G, \mathcal{A}, \varphi_0, \Delta G)$;
7        $\text{FindIncrementalMatch}(Q, G, \mathcal{A}, \Delta G)$;
8   **else if** $\Delta G.op$ *is* − **then**
9        $\text{FindIncrementalMatch}(Q, G, \mathcal{A}, \Delta G)$;
10       $\text{UpdateIndex}(Q, G, \mathcal{A}, \varphi_0, \Delta G)$;
11   Apply ΔG to G;

12 **Procedure** $\text{FindIncrementalMatch}(Q, G, \mathcal{A}, \Delta G)$
13   **for** $k \leftarrow 1$ *to* n *where* $n = |E(Q)|$ **do**
14       $\Delta R_k \leftarrow R_k \cap \Delta G$;
15    $\Delta\mathcal{M} \leftarrow \{\}$;
16   **for** $k \leftarrow 1$ *to* n *where* $n = |E(Q)|$ **do**
17       $\Delta Q_k \leftarrow (\bowtie_{i\in[1,k-1]} R_i) \bowtie \Delta R_k \bowtie (\bowtie_{i\in[k+1,n]} (R_i - \Delta R_i))$;
18       $\Delta\mathcal{M} \leftarrow \Delta\mathcal{M} \cup \Delta Q_k$;
19   Output $\Delta\mathcal{M}$ as positive/negative matches if $\Delta G.op$ is +/−;

---

two problems within the framework. We will present the details in the next section.

4 ALGORITHMS UNDER STUDY

4.1 Overview

This paper studies six sequential algorithms on exact CSM, including IncIsoMatch, SJ-Tree, Graphflow, IEDyn, TurboFlux, and SymBi. IncIsoMatch is recomputation-based, while the other four are incremental methods. Our IVM-based framework can capture all CSM algorithms under study. First, it supports CSM algorithms with different enumeration methods, including vertex-at-a-time based and edge-at-a-time based. Second, it supports algorithms with and without index. Particularly, if the CSM has no index, it directly obtains relations from G. In contrast, it enumerates incremental matches based on the index for indexing-based methods. Finally, our framework supports both algorithms that are of constant delay enumeration and those not.

Rather than storing relations in the index, native methods [26, 27, 33, 39] generate a candidate vertex set $C(u)$ for each query vertex u. These algorithms can be seamlessly integrated into our framework. Particularly, given $e(u, u') \in E(Q)$, if they maintain edges between candidates of $C(u)$ and $C(u')$, then the relation $R(u, u')$ is constructed by adding these edges. Therefore, for ease of understanding, we introduce the indexing method of native algorithms in terms of the generation of candidate vertex sets.

In the following, we first introduce the ordering, indexing, and enumeration techniques of each algorithm based on our framework and then show their space and time complexity. The algorithms `BuildInitialIndex` and `UpdateIndex` for index-based methods, the implementation details within our framework, and the formal complexity analysis are presented in our full paper [52]. Note that, within our framework, all algorithms under study can keep the same time and space complexity as the original papers. Finally, we summarize the differences between these algorithms.

In the original papers, the competing algorithms work on dynamic graphs with single-update. Without loss of generality, we assume that $\Delta G = \{e(v_x, v_y)\}$, and the query edge $e(u_x, u_y)$ has the same label with $e(v_x, v_y)$, i.e., $L(u_x) = L(v_x), L(u_y) = L(v_y)$ and $L(e(u_x, u_y)) = L(e(v_x, v_y))$. We focus on the data structures and filtering rules in the indexing phase, and the join evaluation methods (especially the matching order) in the enumeration phase (Line 17 in Algorithm 2) given ΔG.

4.2 IncIsoMatch [19]

To the best of our knowledge, IncIsoMatch is the first CSM algorithm. It finds incremental matches by recomputation and does not generate the initial matching order φ_0 or index $\mathcal{A}$. On each update, IncIsoMatch extracts a subgraph G' of G based on $e(v_x, v_y)$ and enumerates results on G' instead of G to reduce the search space.

Enumeration. Let *dia* denote the *diameter* of Q, i.e., the length of the longest shortest path in Q. Given $\Delta G = \{e(v_x, v_y)\}$, V_{dia} contains the vertices in G within *dia* hops from both v_x and v_y, and G_{dia} is the vertex-induced subgraph of G on V_{dia}. IncIsoMatch computes $\Delta \mathcal{M}$ (Lines 15-18 in Algorithm 2) as follows: 1) enumerate all matches $\mathcal{M}$ of Q in G_{dia} without $e(v_x, v_y)$; 2) enumerate all matches $\mathcal{M}'$ of Q in G_{dia} with $e(v_x, v_y)$; and 3) computes the difference between $\mathcal{M}$ and $\mathcal{M}'$.

Complexity. The time and space cost of extracting G_{dia} by breath-first search is $O(|E(G)| + |V(G)|)$. IncIsoMatch is not worst-case optimal since the number of matches it finds may be greater than the number of incremental matches.

4.3 Graphflow [30]

Graphflow is a system that supports both SM and CSM queries with multi-way joins. It enumerates $\Delta \mathcal{M}$ without φ_0 or $\mathcal{A}$.

Enumeration. Given $\Delta G = \{e(v_x, v_y)\}$, Graphflow generates φ as follows: 1) add u_x, u_y to φ; and 2) repeatedly add a query vertex u^*, which is not in φ and with the maximum number of neighbors in φ, to φ until all query vertices are selected. If there are ties among vertices, Graphflow picks the vertex with a larger degree. As the matching order for each query edge is fixed, Graphflow generates φs offline and directly retrieves it in online processing. Graphflow evaluates the join query by extending partial results along φ.

Complexity. Graphflow finds ΔM with worst-case optimality.

4.4 SJ-Tree [16]

SJ-Tree evaluates the join query with binary joins using the index.

Ordering. The initial matching order φ_0 contains a sequence of query edges. Given Q and G, SJ-Tree defines the selectivity of $e(u, u') \in E(Q)$ as the size of the relation $R(u, u')$ obtained from G. SJ-Tree generates φ_0 based on the selectivity: 1) add the edge with the minimum selectivity to φ_0; and 2) repeatedly add the edge, which has the minimum selectivity among the edges with at least one endpoint in φ_0, to φ_0 until all query edges are selected.

Indexing. The index $\mathcal{A}$ is a left-deep tree, where the ith leaf node on the left maintains the relation associated with the ith query edge in φ_0, and an internal node records partial results of the query. As SJ-Tree uses the hash join, the relation of a node Q' is stored as a hash table. The key is the common vertices between Q' and Q'' where Q'' is the sibling node of Q' in the left-deep tree.

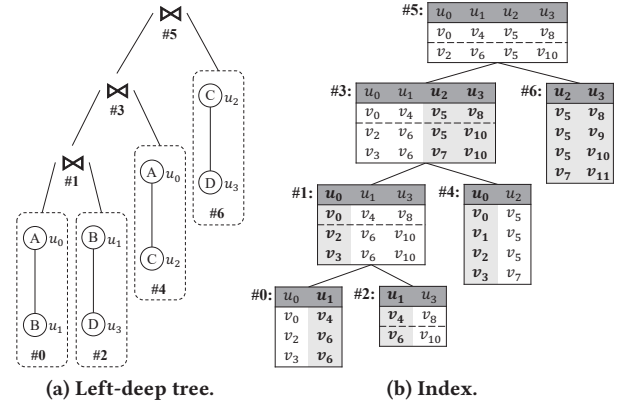**(a) Left-deep tree.**   **(b) Index.**

Figure 4: SJ-Tree for the update in Figure 1c.

Enumeration. In online processing, SJ-Tree only supports the insertion operation. Given $\Delta G = \{e(v_x, v_y)\}$, SJ-Tree inserts $e(v_x, v_y)$ into $R(u_x, u_y)$, and triggers the incremental computation along the bottom-up order of the left-deep tree. The partial results of each join operation are inserted into the relations of internal nodes. To fit the algorithm into our framework, SJ-Tree does not call the function UpdateIndex (Line 6) but directly enumerates incremental matches using the binary join (Line 17) and caches all partial results to the index. Similarly, offline processing is achieved by running SM on G and caching the partial results.

Complexity. The space cost of the index, the time complexity of building and updating the index, and the time complexity of the incremental enumeration are $O(|E(G)|^{|E(Q)|})$ due to the edge-at-a-time join approach. Therefore, SJ-Tree is not worst-case optimal.

Example 1. Figure 4b shows the index where each table corresponds to a node in the left-deep tree in Figure 4a and the join keys are grey shaded. The tuples below the dashed line in each table are newly added on the update operation. As $e(v_6, v_{10})$ can be mapped to $e(u_1, u_3)$ based on their labels, it is first inserted into Table #2. Then, the insertion triggers the join in the bottom-up order of the left-deep tree. The partial results generated by the join operation are stored in each table.

4.5 TurboFlux [33]

The index of SJ-Tree takes exponential space. To solve the problem, TurboFlux proposes to store matches of paths in Q without materialization. Furthermore, TurboFlux evaluates the query with the vertex-at-a-time method (Algorithm 1).

Ordering. Given Q and G, TurboFlux first generates a spanning tree Q_T of Q rooted at u_r. u_r is selected as follows: 1) pick $e(u, u')$ appearing the least frequently in G; and 2) pick the endpoint of $e(u, u')$ appearing least frequently in G. Next, TurboFlux enlarges Q_T by repeatedly adding the edge, which has the fewest matches in G and has one endpoint in Q_T, until all query vertices are selected. φ_0 is a depth-first-order of all $v \in V(Q_T)$ starting from u_r. At Lines 2, 6, and 10 in Algorithm 2, TurboFlux pass Q_T to the functions rather than Q.

Indexing. The index of TurboFlux, called DCG, has the same structure as Q_T, maintaining candidate vertex sets and data edges between candidates. Given $u \in V(Q_T)$, we denote P_u as the path
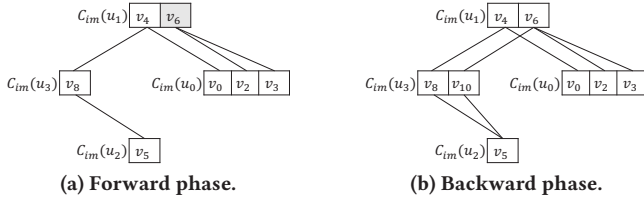
$C_{im}(u_1)$ | v_4 | v_6

$C_{im}(u_3)$ | v_8 $C_{im}(u_0)$ | v_0 | v_2 | v_3

$C_{im}(u_2)$ | v_5

(a) Forward phase.

$C_{im}(u_1)$ | v_4 | v_6

$C_{im}(u_3)$ | v_8 | v_{10} $C_{im}(u_0)$ | v_0 | v_2 | v_3

$C_{im}(u_2)$ | v_5

(b) Backward phase.

Figure 5: DCG for the update in Figure 1c.

from u_r to u in Q_T and T_u as the subtree of Q_T rooted at u. DCG puts a data vertex into a candidate set based on Proposition 2.

PROPOSITION 2. *Given $v \in C(u)$, if the mapping (u, v) appears in a match of Q, then v satisfies that 1) it appears in a match of P_u; and 2) it appears in a match of T_u.*

TurboFlux also maintains $C_{im}(u)$, i.e., the *implicit candidate* set of u for each $u \in V(Q)$. $v \in C_{im}(u)$ if v only satisfies the first condition. In offline processing, DCG sets candidate sets in two phases. In the forward phase, DCG processes each u along φ_0 by the following operations: given $v \in V(G)$, if $L(v) = L(u)$ and v has a neighbor in $C_{im}(u')$ for each $u' \in N_+^{\varphi_0}(u)$, then insert v to $C_{im}(u)$. Next, in the backward phase, DCG processes each u along reversed order of φ_0 by the following operations: given $v \in V(G)$, if $L(v) = L(u)$, v has a neighbor in $C(u')$ for each $u' \in N_-^{\varphi_0}(u)$ and $v \in C_{im}(u)$, then insert v to $C(u)$.

In the online phase, given $\Delta G = \{e(v_x, v_y)\}$, if $e(u_x, u_y)$ belongs to $E(Q_T)$, then we assume that u_x precedes u_y in φ_0 without loss of generality. DCG updates candidate sets using the same operations as those in offline processing starting from u_y. If $e(u_x, u_y)$ is a non-tree edge, TurboFlux does not update DCG. The index update for deletion is similar to that for insertion, and thus we omit the details.

Example 2. Given Q in Figure 1a, suppose that Q_T is rooted at u_1 with $e(u_0, u_2)$ as the non-tree edge. Figure 5a shows the DCG in which each node records an implicit candidate set. A vertex v is grey shaded if $v \in C_{im}(u)$ but $v \notin C(u)$. When $e(v_6, v_{10})$ is inserted into G, DCG first starts the forward phase. v_{10} is inserted into $C_{im}(u_3)$ and the edge $e(v_{10}, v_5)$ is recorded. In the backward phase, v_{10} is inserted to $C(u_3)$ since v_{10} has a subtree matching T_{u_3}. After that, v_6 is inserted to $C(u_1)$ because v_6 has a subtree matching T_{u_1}.

Enumeration. The enumeration only considers the candidates in $C(u)$ for each $u \in V(Q)$ according to Proposition 2. Given Q and G, TurboFlux first generates φ starting from u_r as follows: 1) repeatedly delete the leaf node $u \in V(Q_T)$ with the minimum number of matches of P_u in DCG; and 2) φ is the reverse sequence of deletion. After that, given $\Delta G = \{e(v_x, v_y)\}$, TurboFlux generates a matching order φ' starting from u_x and v_y as follows: 1) add u_x, u_y to φ'; 2) add the vertices in P_{u_x} along the order from u_x to u_r; and 3) add other vertices to φ' along the order of the φ that starts from u_r. Because all matching orders are fixed in online processing, we can precompute them for each query edge.

Complexity. The space and time complexity of the DCG are both $O(|E(G)||V(Q)|)$ and the enumeration is wort-case optimal.

4.6 SymBi [39]

The index of TurboFlux does not use the non-tree edges to prune candidate vertex sets. To solve the problem, SymBi proposes to prune candidate vertex sets using all query edges.

Ordering. Given Q, SymBi first builds a DAG (directed acyclic graph) Q_D of Q by executing a breadth-first-search from a root vertex u_r. Each edge is directed from the earlier visited vertex to the later visited one. The height of Q_D is the length of the longest path from u_r to sink vertices u_s. SymBi selects a query vertex as u_r such that Q_D has the highest height. φ_0 is the breadth-first order of Q starting from u_r.

Indexing. SymBi builds an index called DCS, maintaining candidate vertex sets and data edges between candidates similar to TurboFlux. We denote $\mathcal{P}_u$ as the set of paths from u_r to u in Q_D and $\mathcal{T}_u$ as the set of paths from u to each u_s in Q_D. SymBi puts a data vertex into a candidate set based on Proposition 3.

PROPOSITION 3. *Given $v \in C(u)$, if the mapping (u, v) appears in a match of Q, then v satisfies that 1) for each $P \in \mathcal{P}_u$, (u, v) exists in a match of P; and 2) for each $P \in \mathcal{T}_u$, (u, v) exists in a match of P.*

The offline and online processing of SymBi is the same as that of TurboFlux, except that the first input parameter of function-calls at Lines 2, 6, and 10 in Algorithm 2 are Q rather than Q_T.

Enumeration. The enumeration only considers the candidates $C(u)$ for each $u \in V(Q)$ according to Proposition 3. We illustrate the matching order determination of SymBi based on Algorithm 1. Given an intermediate result M, $N_M(u)$ is the set of neighbors of u in M. X_M is the set of query vertices not in M but $N_M(u) \neq \emptyset$. At Line 4, SymBi selects $u \in X_M$ with the minimum value of $|C_M(u)|$ as the next query vertex to extend M. $C_M(u)$ is computed based on $N_M(u)$ instead of $N_+^\varphi(u)$.

Complexity. The space and time complexity of the DCS are both $O(|E(G)||E(Q)|)$ and the enumeration is worst-case optimal.

4.7 IEDyn [26, 27]

IEDyn is a CSM algorithm for acyclic queries, achieving constant delay enumeration under the setting of graph homomorphism.

Ordering. Given Q and G, IEDyn first selects an arbitrary $u_r \in V(Q)$ as the root of Q. Then, IEDyn generates φ_0 by executing a depth-first-search on Q starting from u_r.

Indexing. The index of IEDyn maintains candidate vertex sets and data edges between candidates, similar to TurboFlux. IEDyn puts a data vertex into a candidate set based on Proposition 4.

PROPOSITION 4. *If the mapping (u, v) appears in a match of Q in G, then v appears in a match of T_u.*

IEDyn only performs the backward phase, and the operations are the same as those of TurboFlux. We call the index built in the offline phase the *global index*. On each update, IEDyn can update the global index directly and find all matches $\mathcal{M}$ based on the global index. However, since CSM focuses on ΔM, IEDyn maintains a *local index*, storing an additional candidate set $C_\Delta(u)$ for each $u \in V(Q)$. In online processing, given $\Delta G = \{e(v_x, v_y)\}$, we assume that u_x precedes u_y in φ_0. For each $u \notin P_{u_x}$, IEDyn assigns $C(u)$ to $C_\Delta(u)$ directly. After that, IEDyn performs the same operations as those in offline processing starting from u_x to set $C_\Delta(u)$ where $u \in P(u_x)$.

Enumeration. The matching order for enumeration φ is the same as φ_0. The enumeration is performed on the local index. After the enumeration, the local index is merged into the global index.

Example 3. Suppose we get Q by removing the edge $e(u_2, u_3)$ in Figure 1a and G from Figure 1b, Figure 6a shows the global index

Algorithm	Computation Method	Index		Enumeration		Functionality						
		Space Complexity	Time Complexity	Worst-case Optimality	Constant Delay	Edge +	Edge −	Vertex +	Vertex −	Label Update	Batch Update	Early Termination
IncIsoMatch	Recomputation	N/A	N/A	✗	✗	✓	✓	✓	✓	✓	✗	✗
Graphflow	Direct Inc	N/A	N/A	✓	✗	✓	✓	✓	✓	✓	✓	✓
SJ-Tree	Index-based Inc	$O\left(\|E(G)\|^{\|E(Q)\|}\right)$	$O\left(\|E(G)\|^{\|E(Q)\|}\right)$	✗	✗	✓	✗	✓	✗	✗	✗	✗
IEDyn	Index-based Inc	$O(\|E(G)\|\|V(Q)\|)$	$O(\|E(G)\|\|V(Q)\|)$	✓	✓	✓	✓	✓	✓	✓	✓	✓
TurboFlux	Index-based Inc	$O(\|E(G)\|\|V(Q)\|)$	$O(\|E(G)\|\|V(Q)\|)$	✓	✗	✓	✓	✓	✓	✓	✗	✓
SymBi	Index-based Inc	$O(\|E(G)\|\|E(Q)\|)$	$O(\|E(G)\|\|E(Q)\|)$	✓	✗	✓	✓	✓	✓	✓	✗	✓

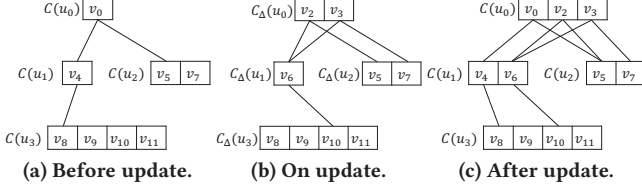**(a) Before update.**   **(b) On update.**   **(c) After update.**

Figure 6: Global and local index for update in Figure 1c.

before the update. When $e(v_6, v_{10})$ is inserted into G, it matches $e(u_1, u_3)$. IEDyn first assigns $C(u_2)$ to $C_\Delta(u_2)$ and $C(u_3)$ to $C_\Delta(u_3)$. Then, since v_6 has a subtree matching T_{u_1}, v_6 is inserted in $C_\Delta(u_1)$. After that, v_2 and v_3 are inserted in $C_\Delta(u_0)$ since both of them have neighbors in $C_\Delta(u_1)$ and $C_\Delta(u_2)$. The local index on update is shown in Figure 6b. After the enumeration, the local index is merged into the global index, shown in Figure 6c.

Complexity. The time and space complexity of the index of IEDyn are both $O(|E(G)||V(Q)|)$. The enumeration is worst-case optimal. Additionally, the algorithm can enumerate incremental results of acyclic queries under graph homomorphism with constant delay (Proposition 5). We put the proof in our full paper [52].

PROPOSITION 5. *Given Q, G, and $\Delta G \in \Delta\mathcal{G}$, where Q is a tree, IEDyn enumerates all incremental matches with constant delay after $O(|E(G)||V(Q)|)$ time of preprocessing.*

4.8 Comparison

The space complexity of the index and the time complexity of updating the index on each update of competing algorithms are summarized in Table 1. In the following, we compare the algorithms under study in their enumeration, and functionalities.

Enumeration. Table 1 summarizes the worst-case optimality of each algorithm. IncIsoMatch is not worst-case optimal since the number of matches it finds may be greater than the number of incremental matches. SJ-Tree extends a partial match by an edge at a time, and thus not worst-case optimal either. Among all methods, only IEDyn supports constant delay enumeration under the setting of tree homomorphism, as stated in Proposition 5. The enumeration of IncIsoMatch is not of constant delay because the algorithm needs to find all matches in G_{dia} to get the first incremental result. SJ-Tree does not support constant delay enumeration either. When an edge is inserted into a leaf node of the left-deep tree, the edge is joined with all partial results on its sibling node with the same key. However, there may be up to $V(G)$ partial results that do not lead to a match. TurboFlux and SymBi also fail to enumerate incremental matches with constant delay, because their indexes store both $C(u)$ and $C_{im}(u)$ for each $u \in V(Q)$. When the algorithms

get the neighbors of a candidate (Line 3 or Line 4 in Algorithm 1), they may visit up to $V(G)$ vertices v where $v \in C_{im}(u')$ but $v \notin C(u')$. Since our framework supports different indexing and ordering methods, it can accommodate both algorithms that are of constant delay enumeration and those not.

Functionality. Table 1 also illustrates the functionalities of the algorithms under study. All algorithms support edge insertion. All algorithms expect SJ-Tree support edge deletion. No algorithm provides specialized methods for vertex update. But a vertex insertion (resp. deletion) can be achieved by inserting (resp. deleting) the vertex and all the edges adjacent to the vertex. Therefore, all algorithms support vertex insertion, while SJ-Tree does not support vertex deletion because it cannot perform edge deletion. Similarly, no algorithm provides specialized methods for label updates of a vertex or an edge. But the label update of a vertex (resp. an edge) can be achieved by deleting the vertex (resp. the edge) and then inserting a vertex (resp. an edge) with the new label. Therefore, IncIsoMatch, Graphflow, IEDyn, TurboFlux, and SymBi support label updates because they can perform both the insertion and the deletion of a vertex or an edge. Furthermore, only Graphflow and IEDyn were proposed with batch update support initially. The index update operations of SJ-Tree, TurboFlux, and SymBi work for single edge insertion or deletion only. As IncIsoMatch extracts a subgraph of G based on the distance to the updated edge, it processes a single update at a time. We also show the early termination support of each algorithm, namely the ability to return after finding a specific number of incremental results on each update. IEDyn, Graphflow, TurboFlux, and SymBi support early termination. IncIsoMatch cannot return before finding all incremental matches since it computes them based on the difference between results in G_{dia} before and after the update. SJ-Tree cannot terminate early either, because its index records all the partial results of the join operation and must be kept consistent with each graph data snapshot.

5 EXPERIMENT SETTING

We study six algorithms in our experiments, including IncIsoMatch (IM), SJ-Tree (SJ), Graphflow (GF), IEDyn (DYN), TurboFlux (TF), and SymBi (SYM). As most of those algorithms except SJ-Tree are not open-sourced, we have to use our homegrown algorithm based on the common IVM model. We implement all algorithms in C++ and optimize them with our best efforts. Our implementation of SJ-Tree is faster than the original version. To state the performance of those CSM algorithms, we also include the latest and open-sourced SM algorithm RapidMatch (RM) [51] in our experiments. The graph is stored as adjacent arrays. Vertices in each neighbor array are

Table 2: Datasets.

Datasets	$|V|$	$|E|$	$|\Sigma_V|$	$|\Sigma_E|$	d_{avg}	d_{max}	c_{max}
Amazon (az)	0.4M	2.4M	6	1	12.2	0.2M	10
LiveJournal (lj)	4.9M	42.9M	30	1	18.1	4.3M	350
Netflow (nf)	3.1M	2.9M	1	7	2.0	0.2M	8
LSBench (ls)	5.2M	20.3M	1	44	8.2	2.3M	27

sorted by their IDs. Therefore, an edge is inserted or deleted with the binary search of the arrays. The code was compiled by g++ 8.3.1. We experiment on a Linux machine with two Intel Xeon Gold 5218 CPUs and 512GB RAM.

Datasets. Following previous studies [16, 33, 39], we use Netflow and LSBench, two dynamic graphs, in our experiments. Netflow is a real-world graph containing anonymized passive traffic traces [13]. LSBench is a synthetic social network produced by the Linked Stream Benchmark data generator [34]. The insertion (resp. deletion) rate is defined as the ratio of the number of edge insertions (resp. deletions) to the total number of edges in the original dataset. We use data graphs with the insertion rate of 10% by default. Specifically, we set the first 90% edges of each dataset as the initial graph and the remaining 10% as the insertion stream. Edges of LSBench and Netflow contain 44 and 7 labels, respectively. The label distribution is skewed, e.g., 13.7% edges in LSBench have the same label and as high as 70.9% edges in Netflow have the same label.

In addition to the two commonly used datasets, we further generate dynamic graphs from static graphs. We evaluate static graphs including Amazon and LiveJournal by randomly sampling 10% edges as the insertion stream as in related work [30, 37]. The original datasets are unlabeled, and we randomly assign one of 6 and 30 labels to each vertex in Amazon and LiveJournal, respectively.

Table 2 summarizes the graph size, number of labels, the average degree d_{avg}, the maximum degree d_{max}, and the maximum core number c_{max} of each dataset. Netflow and Amazon are sparse and have no dense sub-structures, while Livejournal and LSBench are relatively dense. In summary, our datasets cover a wide range of settings, e.g., the distribution of labels and the density of graphs. In addition, we perform sensitivity studies on those data sets, such as altering the insertion and deletion rate in our experiments.

Query graphs. Following previous work [33, 39], we generate query graphs by randomly extracting subgraphs from the data graph. We divide query graphs into three types based on the density: tree, sparse ($d_{avg} \leqslant 3$) and dense ($d_{avg} > 3$). Both sparse and dense are cyclic queries. For each type, we generate query graphs with $V(Q)$ varied from 4 to 12 in an increment of two. We generate 100 query graphs of each size and each query type as a query set. Due to the space limit, we report the results on query sets containing query graphs with 6 vertices by default.

Metrics. We measure the *query time*, which is the elapsed time of the online processing given a graph update stream. We exclude the time for data graph update because it is the same for all algorithms in our framework. For the index-based method, the query time consists of the *indexing time*, which is the time on updating the index, and the *enumeration time*, which is the time on enumerating results. To complete the experiments in a reasonable time, we set the time limit for processing a query to one hour. We say that a query is *unsolved* if the execution exceeds the time limit. Given an unsolved query, if the algorithm finds fewer than 10^9 incremental

matches within the time limit, then we call it a *hard unsolved* query because the algorithm achieves considerable performance if it can find many results despite that it runs out of time. Additionally, we count the number of candidates for each query vertex to compare the pruning power of indexes.

By default, we report the average value of a query set. To compare the performance of two methods A and B on an individual query, we examine the *individual speedup* of A over B, which is $\frac{1}{|Q|} \sum_{Q \in Q} \frac{t_B(Q)}{t_A(Q)}$ where Q is a query set and $t(Q)$ is the query time on Q. To compare the relative performance of multiple methods on an individual query in terms of a metric X, we compute the *relative performance* of each method A, which is $\frac{1}{|Q|} \sum_{Q \in Q} \frac{X_A(Q)}{X^*(Q)}$ where $X_A(Q)$ is the value of A and $X^*(Q)$ is the maximum value among competing algorithms.

6 EXPERIMENTAL RESULTS

We first evaluate the overall performance of competing algorithms, and then examine the effectiveness of indexes and matching orders.

6.1 Overall Comparison

Figure 7 presents the query time breakdown of competing algorithms under subgraph isomorphism. For an index-based method, the suffixes "-IDX" and "-ENUM" represent the indexing and enumeration time in the incremental matching phase, respectively. We only report the results of IM and SJ in *az* because, in other datasets, IM frequently runs out of time, while SJ runs out of memory. The figure shows that IM is much slower than incremental algorithms due to the recomputation. In the following, we focus on GF, DYN, TF, SYM, and RM, which have valid experiment results in all datasets. We also show the query time under subgraph homomorphism in our full paper [52]. There is little difference in performance between the two settings. Consequently, in the remaining experiments, we only report the results under subgraph isomorphism, a common practice in real-world applications [9, 36, 45, 56]. Nevertheless, our findings applies to both subgraph isomorphism and homomorphism.

On tree queries, TF and DYN are slightly better than SYM, all of which outperform GF. In contrast, GF runs faster than others on dense queries because the indexing time can dominate the cost. SYM and GF generally run much faster than TF on sparse queries. As the time complexity of indexing of TF is lower than that of SYM, TF takes less indexing time than SYM. DYN spends more time on indexing than TF and SYM since it builds the local index and merges it back to the global index on each update. Nevertheless, no CSM algorithm can dominate all others in each case. Even though DYN is of constant delay enumeration, it does not outperform all other competitors.

RM runs faster than CSM algorithms in most cases because (1) All CSM algorithms force the execution starting from the edge updated, whereas RM chooses a matching order optimized based on the entire data graph; and (2) CSM algorithms may join the same relation on multiple updates, while RM performs the join operation only once. However, RM may underperform CSM algorithms in short-running queries because the time RM spends on finding matches in the initial graph may offset the query time on subsequent updates.

Table 3 lists the number of unsolved queries. GF and DYN have more unsolved queries than others on tree queries, while TF has
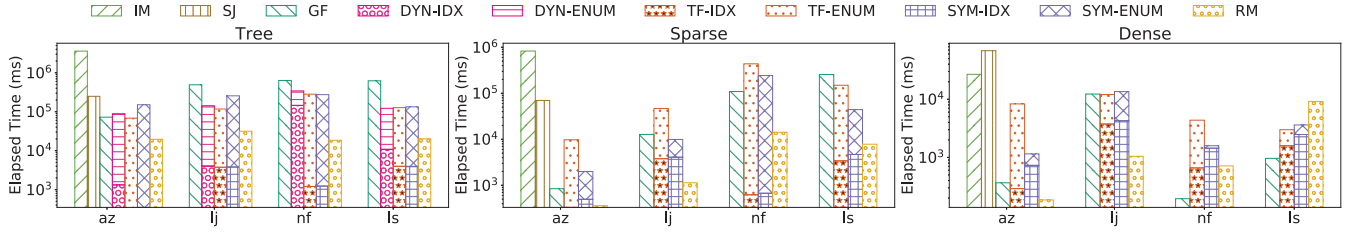
Figure 7: Comparison of competing algorithms on query time under subgraph isomorphism.

Table 3: The number of unsolved queries (U.) and hard unsolved queries (HU.). The unsolved queries include hard unsolved ones. Dataset *az* and Algorithm RM are omitted as there is no unsolved query.

Query Structure	Method	lj		nf		ls		Total	
		U.	HU.	U.	HU.	U.	HU.	U.	HU.
Tree	DYN	4	0	57	18	11	0	72	18
	TF	3	0	38	3	11	0	52	3
	GF	6	0	55	19	11	4	72	23
	O-DYN	3	0	45	4	11	0	59	4
	O-TF	3	0	38	2	11	0	52	2
	O-GF	3	0	45	7	12	0	60	7
	SYM	4	0	41	1	12	0	57	1
Sparse	TF	0	0	20	15	35	32	55	47
	GF	0	0	2	2	33	30	35	32
	O-TF	0	0	20	14	29	26	49	40
	O-GF	0	0	2	1	17	14	19	15
	SYM	0	0	11	6	4	1	15	7
Dense	TF	0	0	1	1	2	2	3	3
	GF	0	0	0	0	0	0	0	0
	O-TF	0	0	0	0	0	0	0	0
	O-GF	0	0	0	0	0	0	0	0
	SYM	0	0	0	0	0	0	0	0
Total	TF	3	0	59	19	48	34	110	53
	GF	6	0	57	21	44	34	107	55
	O-TF	3	0	58	16	40	26	101	42
	O-GF	3	0	47	8	29	14	79	22
	SYM	4	0	52	7	16	1	72	8

more than GF and SYM on cyclic queries. SYM is more robust than others, with fewer unsolved queries, especially the hard unsolved ones. To further investigate the index cost, we ran the queries without enumerating any match. We found that 18 queries in DYN did not finish in time whereas the other schemes finished all queries.

Summary. According to the experiment results, we have the following findings on the overall performance.

(1) The recomputation-based method is much slower than incremental methods. SJ-Tree runs out of memory in most cases.
(2) Although SYM is more stable than other algorithms, no algorithm can dominate others in each case.
(3) On tree queries, TF and DYN are slightly better than SYM, all of which run much faster than GF.
(4) On sparse queries, TF performs worse than SYM and GF, and GF runs faster than SYM in sparse data graphs; otherwise, SYM has the best performance.
(5) On dense queries, GF runs faster than SYM and TF.
(6) Index update in DYN bears more significant overhead than the other schemes, probably due to the maintenance for constant delay enumeration.
(7) RM outperforms CSM methods in most cases.

6.2 Effectiveness of Individual Techniques

6.2.1 Effectiveness of Indexes. We examine the pruning power and the impact of the index on the overall performance.

Pruning Power. Figure 8 shows the number of candidates generated by different indexing methods. *Baseline* denotes the number of candidates obtained based on the vertex and edge labels. We can see that DYN, TF, and SYM can significantly reduce the number of candidates. On tree queries, the pruning power of TF is competitive with that of SYM, and they have a smaller number of candidates than DYN and Baseline. In contrast, SYM outperforms TF on cyclic queries because SYM utilizes the non-tree edges to filter invalid candidates. For the same reason, the gap between SYM and TF enlarges with the increase of the query graph density.

Impact on Overall Performance. As the index of SYM has the best pruning power, we optimize GF, DYN, and TF by integrating their matching orders with the index of SYM. We call the optimized methods O-GF, O-DYN, and O-TF for short. Then, we compare the overall performance of O-GF, O-DYN, and O-TF, with that of GF, DYN, and TF, to examine the impact of the index. Figure 9 shows the individual speedup. With the index, O-GF achieves significant speedup over GF on tree and sparse queries. However, the index sometimes degrades the performance, e.g., the queries in *az* and on dense queries, because the indexing time is non-trivial compared with the query time (see Figure 7). Additionally, the speedup on long-running workloads (e.g., queries in *nf* and *ls*) is higher than that on short-running (e.g., queries in *az* and *lj*). As DYN and TF initially have indexes, the speedup of O-DYN over DYN and O-TF over TF is lower than that of O-GF over GF.

Moreover, we report the number of unsolved queries on O-GF, O-DYN, and O-TF in Table 3. We can see that the number of unsolved queries is reduced. In particular, the index significantly reduces the number of hard unsolved queries of GF on tree queries, because the index can rule out most invalid candidates on acyclic queries, as discussed in Section 4.8. Despite that the index of SYM does not support constant delay enumeration, it still reduces the number of unsolved queries of DYN since no local index is needed.
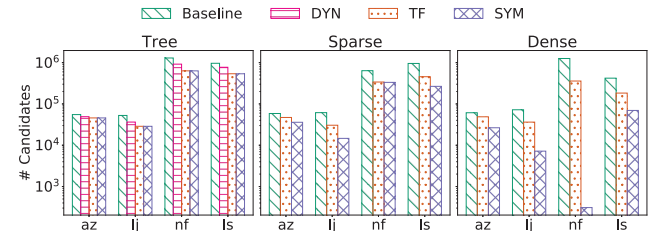

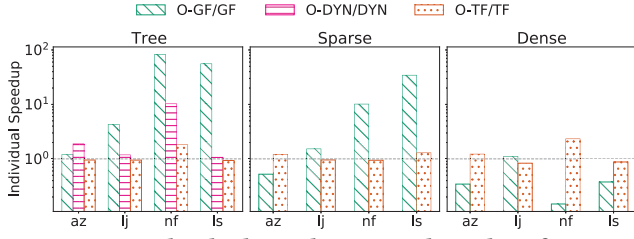Figure 8: Number of candidates generated by different indexing methods.

Figure 9: Individual speedup using the index of SYM.

Summary. We have the following answer for the question: *What is the effectiveness of the index?*

(1) The index of SYM achieves the best pruning power.
(2) The index can reduce the number of candidates and improve the performance, especially on tree and sparse queries.
(3) The index provides more benefits on long-running queries, while the update of the index can dominate the query time on short-running queries.
(4) Although the index of SYM does not support constant delay enumeration, it speeds up the incremental matching of DYN.

6.2.2 Effectiveness of Matching Orders. For a fair comparison, we evaluate the performance of O-GF, O-DYN, O-TF, and SYM, which have the same index.

Impact on #partial results. Figure 10 presents the relative performance of the number of partial results. A smaller value indicates that the method generates fewer partial results. SYM has the fewest partial results among all methods due to the dynamic matching order selection. O-TF performs much worse than O-GF and SYM on sparse and dense queries because its ordering method ignores the effect of non-tree edges, as discussed in Section 4.8. As there are few incremental matches on dense queries against *nf* and *ls*, the relative performance of the three methods is one in these cases.

Impact on Overall Performance. Figure 11 presents the relative performance of the query time. A smaller value indicates that the method has a shorter query time. We can see that O-GF, O-DYN, and O-TF outperform SYM on tree queries despite that SYM results in the fewest partial results. Because the overhead of optimizing matching orders at runtime offsets the benefit. In contrast, SYM and O-GF are competitive on cyclic queries. O-TF performs worse than other algorithms on sparse and dense queries due to many partial results. As shown in Table 3, SYM has fewer hard unsolved queries than O-GF, O-DYN, and O-TF, which demonstrates that the ordering method of SYM is more robust. On sparse queries in the sparse dataset (e.g., *nf*), O-GF has much fewer hard unsolved queries than other methods. In contrast, all the four methods have only a few hard unsolved queries on tree queries because the index can reduce
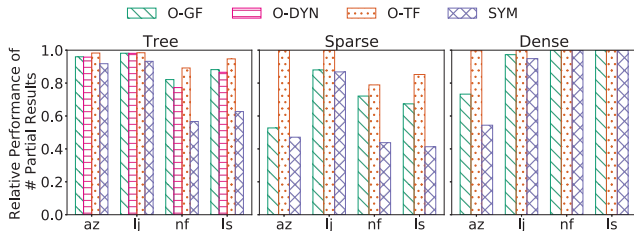

Figure 10: Relative performance of #partial results using different matching orders given the index of SYM.
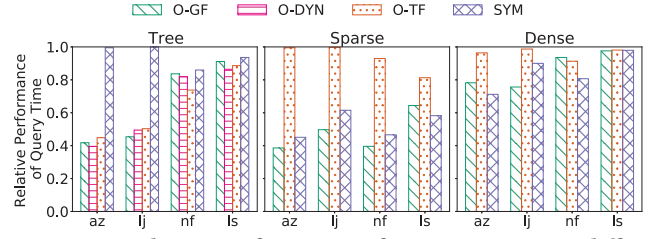

Figure 11: Relative performance of query time using different matching orders given the index of SYM.

many invalid candidates on acyclic queries, as discussed in Section 4.8. Nevertheless, all methods have hard unsolved queries because they all optimize matching orders based on simple heuristics, e.g., the number of candidates and the degree of query vertices.

Detailed Metrics. In order to get more insights, we collect the detailed metrics of GF, TF, and SYM in *ls*. The results in other datasets are similar. Due to the space limit, we focus on sparse queries on which competing algorithms have several unsolved queries. We relabel query *ID*s in the ascending order of the query time. For each query, we report the number of results (denoted by *#RES*), and the number of *invalid partial results* (denoted by *#INV*), i.e., the partial results from which we cannot generate final results. According to Algorithm 1, a partial result is invalid because 1) the local candidate vertex set is empty; or 2) the data vertex has been mapped. We count the number of invalid partial results caused by the two conditions, which are denoted by *EMP* and *VIS*.

As shown in Figure 12, the query time is closely related to the number of results and invalid partial results. Benefiting from the index, SYM generates fewer invalid partial results than GF and TF, and it runs out of time because of the large result size. In comparison, unsolved queries of GF are generally due to the huge number of invalid partial results, as there is no index in GF. Despite that TF utilizes the index, the non-tree edges of the query graph are not considered by its filtering rules. Therefore, many unsolved queries are caused by numerous invalid partial results. Furthermore, we can see that most invalid partial results of SYM are caused by *VIS*. This is because (1) in *ls*, all data vertices and query vertices have the same label, and a data vertex can appear in the candidate vertex sets of multiple query vertices; and (2) the index of SYM is constructed based on graph homomorphism, without concerning *VIS* at all. GF and TF also have invalid partial results caused by *VIS*, but *#EMP* dominates *#INV* due to the low pruning power of the algorithm.

Case Study. As discussed in the above experiments, algorithms run out of time on some queries because of the huge number of results. Additionally, we observe that some queries cannot be completed by any CSM algorithm within the time limit, and only a few
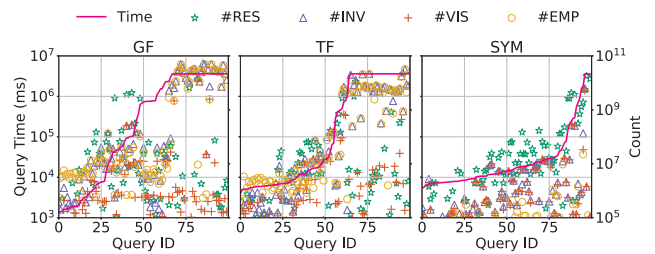

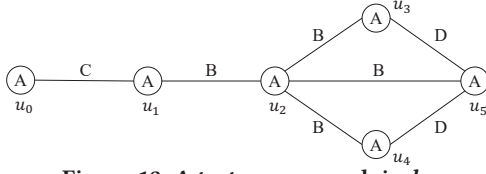**Figure 12: Detailed metrics of sparse queries in *ls*.**

Figure 13: A test query graph in *ls*.

results are found. In contrast, RM can enumerate all matches of Q on the snapshot of G after applying the entire stream $\Delta\mathcal{G}$ quickly. This motivates us to conduct a study on these cases. Figure 13 shows such a query in *ls*. O-GF, O-TF, and SYM cannot find even one result within one hour. Our investigation finds that the poor performance is caused by the ineffective matching order. Specifically, given the insertion of a data edge with label C, all competing algorithms will start the search by mapping it to $e(u_0, u_1)$. For example, O-GF evaluates the query with the matching order $\varphi = (u_1, u_0, u_2, u_5, u_3, u_4)$. The vertex-induced subgraph of Q on the first four vertices is a tree, with many matches. When trying to further extend them, the local candidate vertex set of u_3 (or u_4) is usually empty. Consequently, there is a large number of invalid partial results due to *EMP*. SM algorithms generally start the enumeration from the dense part of Q, which can terminate such invalid search paths at an early stage [8, 51]. Unfortunately, the existing CSM framework forces the search to start from the query edge mapped to the updated data edge e to ensure the matches reported contain e.

Summary. Based on the results, we have the following answer to the question: *What is the effectiveness of the matching orders?*

(1) On tree queries, the ordering methods of DYN and TF are generally better than those of other algorithms.
(2) On sparse queries, the ordering method of GF performs well on sparse data graphs; otherwise, the ordering method of SYM achieves the best performance.
(3) On dense queries, the ordering method of GF generally outperforms those of TF and SYM.
(4) Although the ordering method of SYM is more robust than those of other algorithms, all methods generate ineffective matching orders at times because they are based on simple heuristics.
(5) Forcing the search from the query edge mapped to the data edge potentially leads to many invalid partial results.

Moreover, according to the findings on indexes and matching orders, we can obtain the answer to the question: *What is the key factors leading to the performance difference?*

(1) DYN, TF, and SYM run faster than GF on tree queries because the index significantly reduces the invalid partial result size.
(2) TF has a poor performance on cyclic queries because its matching order does not consider the impact of non-tree edges.
(3) SYM has fewer hard unsolved queries than others because its matching order is robust.
(4) GF has a better performance than SYM on dense queries because the indexing cost can dominate the query time.

Additionally, we have the following answer to the question: *Where did time go in these queries?*

(1) The query time is closely related to the number of results and invalid partial results.

(2) The competing algorithms can find a few results using a long time on some queries because of the large number of invalid partial results, which are incurred by *VIS* and *EMP*.
(3) The query vertices with the same label can result in many *VIS*s, while the ineffective index can lead to many *EMP*s.
(4) The tree queries have a long running time and many unsolved queries because they have many results in the data graph.
(5) The sparse queries have more hard unsolved queries than other types because they have a mixed sub-structure (e.g., a path with a diamond in Figure 13), and the simple heuristics in existing ordering methods easily fail to process them.
(6) The dense queries generally have a shorter running time than other types because they have fewer results, and the invalid search paths can be terminated at an early stage due to constraints of query edges.

Omitted Experiment Results. We evaluate the standard deviation, response time, memory usage, offline indexing time, detailed metrics, and scalability of competing algorithms. We also report the performance of the algorithms with various query graph properties [11], data graph properties, and updated edge properties. The trends of the experimental results are similar to those under our default setting. As such, we put the results and findings in the full version of this paper [52] due to the space limit.

7 CONCLUSION

In this paper, we conduct an in-depth study on the continuous subgraph matching (CSM) problem. We first propose to model the problem as incremental view maintenance to capture the design space of existing algorithms. Then, we design a common framework based on the model for CSM to depict, analyze and implement six CSM algorithms. Finally, we conduct extensive experiments to evaluate competing algorithms and give an in-depth analysis.

Recommendation. According to our experiments, we make the following recommendation of choices of existing indexing and ordering methods. For the index, use the index of SymBi if the query graph is sparse; or the query takes a long running time. Otherwise, use the direct-incremental methodology. For the matching order, if the query graph is a tree, then use the matching order of IEDyn or TurboFlux; if the query graph is dense or both query and data graphs are sparse, then use the matching order of Graphflow; otherwise, use the matching order of SymBi.

Issues. Our experiments on studying the effectiveness of individual techniques find severe issues in existing CSM algorithms. First, to keep the reported matches containing the updated data edge e, the existing framework starts the search from the query edge mapped to e, leading to many invalid partial results. Second, all matching orders are based on simple heuristics, which can generate ineffective matching orders. Third, although the index generally improves the performance of the query, the overhead of updating the index can offset its benefit on some short-running queries. Due to these issues, existing algorithms encounter performance issues even on small queries.

ACKNOWLEDGMENTS

REFERENCES

[1] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*. 431–446.

[2] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows. *Proc. VLDB Endow.* 11, 6 (2018), 691–704.

[3] 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. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1371–1382.

[4] Sepehr Assadi, Michael Kapralov, and Sanjeev Khanna. 2019. A Simple Sublinear-Time Algorithm for Counting Arbitrary Subgraphs via Edge Sampling. In *10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA (LIPIcs)*, Avrim Blum (Ed.), Vol. 124. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:20.

[5] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *Computer Science Logic, 21st International Workshop, CSL 2007*, Vol. 4646. 208–222.

[6] Christoph Berkholz, Fabian Gerhardt, and Nicole Schweikardt. 2020. Constant delay enumeration for conjunctive queries: a tutorial. *ACM SIGLOG News* 7, 1 (2020), 4–33.

[7] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*. 1447–1462.

[8] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*. 1199–1214.

[9] Laurent Bindschaedler, Jasmina Malicevic, Baptiste Lepers, Ashvin Goel, and Willy Zwaenepoel. 2021. Tesseract: distributed, general graph pattern mining on evolving graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 458–473.

[10] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. 1986. Efficiently Updating Materialized Views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*. 61–71.

[11] Angela Bonifati, Wim Martens, and Thomas Timm. 2017. An Analytical Study of Large SPARQL Query Logs. *Proc. VLDB Endow.* 11, 2 (2017), 149–161.

[12] 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.

[13] CAIDA. 2013. The CAIDA UCSD Anonymized Internet Traces 2013. https://catalog.caida.org/details/dataset/passive_2013_pcap. (2013), (Accessed Date: Mar 07, 2022).

[14] Lei Chen and Changliang Wang. 2010. Continuous Subgraph Pattern Search over Certain and Uncertain Graph Streams. *IEEE Trans. Knowl. Data Eng.* 22, 8 (2010), 1093–1109.

[15] Rada Chirkova, Jun Yang, et al. 2011. Materialized views. *Foundations and Trends in Databases* 4, 4 (2011), 295–405.

[16] 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 *Proceedings of the 18th EDBT International Conference on Extending Database Technology*. 157–168.

[17] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 26, 10 (2004), 1367–1372.

[18] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*. 918–934.

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

[20] Jun Gao, Chang Zhou, and Jeffrey Xu Yu. 2016. Toward continuous pattern detection over evolving large graph with snapshot isolation. *VLDB J.* 25, 2 (2016), 269–290.

[21] Timothy Griffin and Leonid Libkin. 1995. Incremental maintenance of views with duplicates. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. 328–339.

[22] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. 157–166.

[23] 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 *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*. 1429–1446.

[24] 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 ACM SIGMOD International Conference on Management of Data*. 337–348.

[25] Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. 1995. Computing Simulations on Finite and Infinite Graphs. In *36th Annual Symposium on Foundations of Computer Science*. 453–462.

[26] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*. 1259–1274.

[27] 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.

[28] Madhav Jha, C. Seshadhri, and Ali Pinar. 2013. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 589–597.

[29] Daniel M. Kane, Kurt Mehlhorn, Thomas Sauerwald, and He Sun. 2012. Counting Arbitrary Subgraphs in Data Streams. In *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012 (Lecture Notes in Computer Science)*, Vol. 7392. 598–609.

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

[31] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2019. Counting Triangles under Updates in Worst-Case Optimal Time. In *22nd International Conference on Database Theory, ICDT 2019 (LIPIcs)*, Vol. 127. 4:1–4:18.

[32] 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. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*. 925–937.

[33] 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 ACM SIGMOD International Conference on Management of Data*. 411–426.

[34] Danh Le-Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter Boncz, Thomas Eiter, and Michael Fink. 2012. Linked stream data processing engines: Facts and figures. In *International Semantic Web Conference*. 300–312.

[35] Madhusudan Manjunath, Kurt Mehlhorn, Konstantinos Panagiotou, and He Sun. 2011. Approximate Counting of Cycles in Streams. In *ESA 2011, 19th Annual European Symposium, Saarbrücken*, Vol. 6942. 677–688.

[36] Emaad Manzoor, Sadegh M Milajerdi, and Leman Akoglu. 2016. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1035–1044.

[37] Amine Mhedhbi, Chathura Kankanamge, and Semih Salihoglu. 2021. Optimizing One-time and Continuous Subgraph Queries using Worst-case Optimal Joins. *ACM Trans. Database Syst.* 46, 2 (2021), 6:1–6:45.

[38] 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.

[39] 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.

[40] Jayanta Mondal and Amol Deshpande. 2016. Casqd: continuous detection of activity-based subgraph pattern queries on dynamic graphs. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. 226–237.

[41] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case Optimal Join Algorithms. *J. ACM* 65, 3 (2018), 16:1–16:40.

[42] Anil Pacaci, Angela Bonifati, and M Tamer Özsu. 2020. Regular path query evaluation on streaming graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1415–1430.

[43] A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. 2013. Counting and Sampling Triangles from a Graph Stream. *Proc. VLDB Endow.* 6, 14 (2013), 1870–1881.

[44] Andrea Pugliese, Matthias Bröcheler, V. S. Subrahmanian, and Michael Ovelgönne. 2014. Efficient Multiview Maintenance under Insertion in Huge Social Networks. *ACM Trans. Web* 8, 2 (2014), 10:1–10:32.

[45] 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, 12 (2018), 1876–1888.

[46] Raghu Ramakrishnan. 1998. *Database Management Systems*. WCB/McGraw-Hill.

[47] Carlos Rivero and Hasan Jamil. 2017. Efficient and scalable labeled subgraph matching using SGMatch. *Knowledge & Information Systems* 51, 1 (2017).

[48] 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.

[49] Chunyao Song, Tingjian Ge, Cindy X. Chen, and Jie Wang. 2014. Event Pattern Matching over Graph Streams. *Proc. VLDB Endow.* 8, 4 (2014), 413–424.

[50] 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.

[51] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. Rapid-Match: A Holistic Approach to Subgraph Query Processing. *Proc. VLDB Endow.* 14, 2 (2020), 176–188.

[52] Xibo Sun, Shixuan Sun, Qiong Luo, and Bingsheng He. 2022. An In-Depth Study of Continuous Subgraph Matching (Complete Version). *CoRR* abs/2203.06913 (2022).

[53] Nan Tang, Qing Chen, and Prasenjit Mitra. 2016. Graph Stream Summarization: From Big Bang to Big Crunch. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1481–1496.

[54] Ha Nguyen Tran, Jung-jae Kim, and Bingsheng He. 2015. Fast Subgraph Matching on Large Graphs using Graphics Processors. In *Proceedings of the 20th DASFAA International Conference on Database Systems for Advanced Applications*, Vol. 9049. 299–315.

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

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

[57] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference*. 82–94.

[58] Lefteris Zervakis, Vinay Setty, Christos Tryfonopoulos, and Katja Hose. 2020. Efficient Continuous Multi-Query Processing over Graph Streams. In *Proceedings of the 23nd EDBT International Conference on Extending Database Technology*. 13–24.

[59] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: distance index based subgraph matching in biological networks. In *Proceedings of the 12th EDBT International Conference on Extending Database Technology*. 192–203.

[60] Peixiang Zhao, Charu C. Aggarwal, and Min Wang. 2011. gSketch: On Query Estimation in Graph Streams. *Proc. VLDB Endow.* 5, 3 (2011), 193–204.

[61] Peixiang Zhao and Jiawei Han. 2010. On Graph Query Optimization in Large Networks. *Proc. VLDB Endow.* 3, 1 (2010), 340–351.