Fast Continuous Subgraph Matching over Streaming Graphs via Backtracking Reduction

RONGJIAN YANG, School of Data Science, Fudan University, China
ZHIJIE ZHANG, School of Data Science, Fudan University, China
WEIGUO ZHENG, School of Data Science, Fudan University, China
JEFFREY XU YU, The Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong, China

Streaming graphs are drawing increasing attention in both academic and industrial communities as many graphs in real applications evolve over time. Continuous subgraph matching (shorted as CSM) aims to report the incremental matches of a query graph in such streaming graphs. It involves two major steps, i.e., candidate maintenance and incremental match generation, to answer CSM. Throughout the course of continuous subgraph matching, incremental match generation backtracking over the search space dominates the total cost. However, most previous approaches focus on developing techniques for efficient candidate maintenance, while incremental match generation receives less attention despite its importance in CSM. Aiming to minimize the overall cost, we propose two techniques to reduce backtrackings in this paper. We present a cost-effective index CaLiG that yields tighter candidate maintenance, shrinking the search space of backtracking. In addition, we develop a novel incremental matching paradigm KSS that decomposes the query vertices into conditional kernel vertices and shell vertices. With the matches of kernel vertices, the incremental matches can be produced immediately by joining the candidates of shell vertices without any backtrackings. Benefiting from reduced backtrackings, the elapsed time of CSM decreases significantly. Extensive experiments over real graphs show that our method runs faster than the state-of-the-art algorithm orders of magnitude.

CCS Concepts: • **Information systems → Stream management**.

Additional Key Words and Phrases: Subgraph Matching, Streaming Graph, Backtracking Reduction

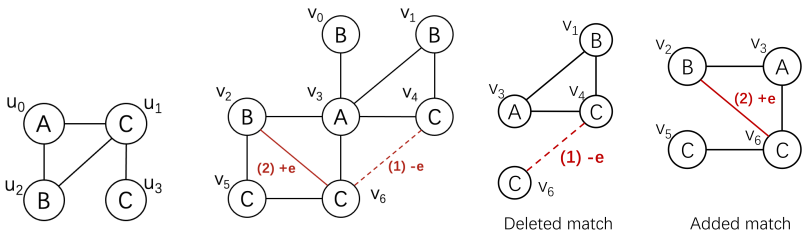
ACM Reference Format:
Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. 2023. Fast Continuous Subgraph Matching over Streaming Graphs via Backtracking Reduction. *Proc. ACM Manag. Data* 1, 1, Article 15 (May 2023), 26 pages. https://doi.org/10.1145/3588695

1 INTRODUCTION

Graphs have been widely used to represent complex relations among a variety of objects, spanning from social networks, knowledge graphs, and road networks to electrical networks. In most real-world applications, the graph evolves over time by adding vertices/edges or deleting vertices/edges, called a streaming graph. For example, in a social network, new user registration and closing an account can be regarded as a vertex addition and deletion in the graph, respectively; creating and cancelling the interaction (e.g., following) between users correspond to edge addition and deletion,

Authors' addresses: Rongjian Yang, School of Data Science, Fudan University, China, rjyang20@fudan.edu.cn; Zhijie Zhang, School of Data Science, Fudan University, China, zhijiezhang18@fudan.edu.cn; Weiguo Zheng, School of Data Science, Fudan University, China, zhengweiguo@fudan.edu.cn; Jeffrey Xu Yu, The Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong, China, yu@se.cuhk.edu.hk.

2836-6573/2023/5-ART15 $15.00
https://doi.org/10.1145/3588695



(a) Query Graph (b) Streaming Graph (c) Incremental Matches

Fig. 1. A running example of CSM.

respectively. Such a social network is a typical streaming graph. How to efficiently manage and query streaming graphs is drawing increasing attention [2, 12, 17, 25, 26, 31, 33, 38].

As an important task, monitoring a pattern Q of interest over a streaming graph G is to perform continuous subgraph matching (shorted as CSM) to report the incremental matches, i.e., the increased or decreased subgraph matches of Q owing to the graph updates. As shown in the running example in Figure 1, one match is deleted after deleting edge (v_4, v_6) and another match is added after adding edge (v_2, v_6).

CSM is useful in a wide range of applications, such as recommendation systems [6, 14, 39], fraud detection [34, 37], and cyber security [9, 10], etc.

For example, a cycle pattern can be served as a strong indication of a fake transaction in e-commerce platforms [34], where the accounts of users (buyers or sellers) are represented as vertices and online transactions, e.g., payment activities, are denoted as dynamic edges. With CSM, suspicious transactions would be detected to generate real-time alerts and trigger prompt actions.

1.1 The Existing Methods

With well-developed subgraph matching algotithms [3, 4, 16], one naive approach for CSM is to enumerate the matches before and after graph updating respectively. However, applying subgraph matching directly suffers from intractable cost [19, 36], as the characteristics of dynamic graph updates are not fully exploited. Considering the fact that the update edge is contained in the desired matches if any, the index-free algorithms, e.g., IncIsoMat [11] and Graphflow [23], find the changed matches by expanding the added or deleted edge without generating unnecessary matches. Such algorithms may waste too much time in exploring the data graph even if there are not any matches since they do not employ any information before updating.

To catch the incremental matches in real-time, more efforts have been devoted to developing index techniques to maintain intermediate results recently [10, 24, 32]. SJ-Tree [10] defines a left-deep decomposition tree, in which each node maintains a set of partial matches. Since the number of matches could be exponential, SJ-tree suffers from an intractable storage overhead. Instead of maintaining partial matches, TurboFlux [24] and SymBi [32] elaborate on the candidate generation and maintenance by exploiting auxiliary data structures and carefully designed filter rules. TurboFlux [24] generates a spanning tree T_Q of Q and introduces an auxiliary data structure called data-centric graph (DCG), to maintain whether an edge in G matches an edge in T_Q or not. It uses the bottom-up method to conduct filtration with only the edges in T_Q taken into account. For graph updates, TurboFlux renews the states of edges in DCG to determine whether every two edges could be matched and whether each edge in G could be contained in a match of Q. SymBi [32] adopts a directed acyclic graph (DAG) built from Q and checks forward or backward *neighbors*,

Table 1. Proportion of candidate maintenance (%)

Method	Deezer	Email	Github	Lastfm	Skitter	Twitch
TurboFlux	0.13	0.093	0.072	0.19	0.091	0.091
SymBi	0.306	0.380	0.072	0.971	0.476	0.003
CaLiG	1.402	2.652	0.913	1.623	1.112	0.235

covering the non-tree edges overlooked by TurboFlux. SymBi employs top-down and bottom-up dynamic programming on the DAG. Only those vertices (that) pass the top-down filer need to perform the bottom-up filter and only those (that) have passed both filters are the valid candidates. Both TurboFlux and SymBi emphasize the succinctness and update efficiency of the index, even if the time cost of maintaining candidates usually accounts for a very small part of the overall cost (as discussed in Section 1.2). In the search stage, TurboFlux adopts a backtracking algorithm with a fixed matching order that depends on the estimated size of query paths. SymBi selects a dynamic match order, which could be adaptively changed during the backtracking, according to the estimated size of extendable candidates of query vertices. They just focus on the matching order rather than improving the backtracking framework.

1.2 Bottleneck of CSM

Among all the aforementioned approaches, the index-based methods often perform better than the others. To further accelerate continuous subgraph matching, a crucial question needs to be answered: what is the bottleneck of improving the efficiency performance? As discussed above, the existing methods mainly focus on how to maintain candidate vertices fast by using succinct indexes. *Surprisingly, incremental match generation, a more important and crucial issue, has not received much attention up to now, despite its dominating role in answering CSM.* Specifically, finding incremental matches involves two major steps, namely candidate maintenance (including candidate generation and index update) and incremental match generation. Once obtaining new candidate vertices caused by graph updates, incremental match generation is conducted to find the increased or decreased matches of Q. Due to the NP-hardness of subgraph isomorphism, incremental matching generation dominates the overall cost. Table 1 reports the proportion of candidate maintenance over the total time cost of different algorithms on 6 real graphs from SNAP [28], where CaLiG is our proposed method. The results are averaged on 50 randomly generated queries for each data graph. It is observed that the time cost of candidate maintenance in TurboFlux and SymBi is less than 0.5% on most graphs.

Actually, the target of an index for CSM is to locate the candidate vertices, further reducing the cost of incremental match generation. A better index should not just focus on fast candidate generation at the cost of delivering more false-positive candidates, degrading the incremental match generation. Hence, there is a trade-off between update efficiency and candidate accuracy for index designing. To measure the accuracy of an index, we use the ratio of false positives, i.e., the ratio of vertices in the candidate set but not in any delivered matches. The fewer false positives, the tighter the candidates generated. Following the principle of cost-effectiveness, we find the indexes of TurboFlux and SymBi are not tight enough. It is worth spending a little more time to obtain tighter candidates, achieving considerable speedups in the subsequent incremental match generation.

A common approach to incremental match generation is backtracking over search space, that is, it extends the partial match by one vertex each time and backtracks if failing or finally succeeding.

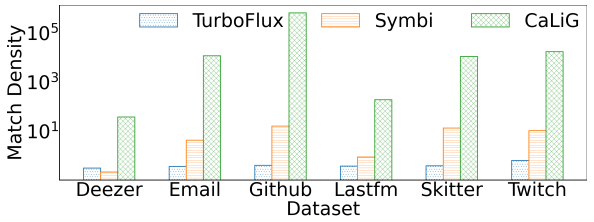


Fig. 2. Averaged match density of different algorithms on 50 random queries for each graph.

To reduce backtrackings, it is expected that more matches can be returned by fewer backtrackings, i.e., one backtracking is supposed to generate more matches. Thus, we define a metric *match density* as the average number of matches generated by one backtracking. Larger match density indicates the algorithm finds all the matches with fewer backtrackings, usually less elapsed time as well. Figure 2 presents the averaged match density of TurboFlux, SymBi, and CaLiG. It is clear that both TurboFlux and SymBi have lower match density and one backtracking could only find about 0.2∼10 matches on average, meaning most backtrackings are invalid and leaves much room to improve.

1.3 Our Approach and Contributions

As discussed above, most previous methods are dedicated to candidate maintenance, rather than directly reducing the cost of match generation. Different from them, we propose two techniques, namely a novel index structure CaLiG (to obtain tighter candidates) and a powerful incremental matching paradigm (to avoid unnecessary backtrackings), aiming to minimize the overall cost of CSM.

First, we propose a cost-effective structure, called *candidate lighting graph (shorted as CaLiG)*, a directed graph where each node represents a matching pair of vertices (u, v). CaLiG provides a tighter candidate space than previous methods, reducing the search space and total backtrackings. Cost-effective, means (that) it is worthwhile to build up an index for tighter candidate maintenance because of the significant speed-up in the subsequent incremental match generation, even if a little bit more time may be required.

In addition, we develop an efficient incremental matching paradigm, *kernel-and-shell search (shorted as KSS)*. The underlying principle is that some vertices are not necessary to be extended one by one following repeated backtrackings. KSS decomposes the query into conditional kernel vertices and shell vertices. The partial matches for the kernel vertices are computed by backtracking first, then the complete incremental matches can be produced immediately by joining the candidates of shell vertices without any backtrackings. We prove that finding the minimum conditional kernel set is an NP-hard problem, and thus propose an efficient greedy method. To evaluate the efficiency of the proposed method, we conducted extensive experimental studies over real graphs. The results confirm the significant superiority of our method.

In summary, we make the following contributions.

- We propose a cost-effective index CaLiG that yields tighter candidate maintenance than the existing methods, which in turn reduces the total backtrackings.
- We design a novel subgraph matching paradigm, called kernel-and-shell search, which can produce incremental matches by simply joining candidates of the shell vertices, reducing the unpromising backtrackings.
- We formalize the problem of minimum conditional kernel set and prove its NP-hardness. To enhance continuous subgraph matching, KSS dynamically selects the best conditional kernel and shell sets for different update edges.

Table 2. Notations

Notations	Descriptions
Q and G	query graph and data graph
$u \in V_Q$ and $v \in V_G$	query vertex and data vertex, respectively
$e(v_i, v_j) \in E_G$	edge between v_i and v_j
$N_Q(u)$	neighbors of u in graph Q
$N_G(v)$	neighbors of v in graph G
d_Q and d_G	maximum degree of Q and G, respectively
(u, v)-MP	a matching pair consists of u and v
(u, v)-MP.state	the lighting state of (u, v)-MP
$In_{CaLiG}(u, v)$	in-neighbors of (u, v)-MP in CaLiG
$Out_{CaLiG}(u, v)$	out-neighbors of (u, v)-MP in CaLiG
$BI(u, v)$	the bigraph for (u, v)-MP

- Extensive experiments over real graphs have demonstrated that our proposed method outperforms the state-of-the-art algorithm orders of magnitude.

2 PROBLEM DEFINITION AND OVERVIEW

In this paper, we focus on undirected vertex-labeled graphs, though our algorithm can be extended to directed and edge-labeled graphs, where each vertex/edge has one label. For the edge-labeled graph, we can build a vertex-labeled graph by taking each edge as a vertex that connects two endpoints, where the label of the newly added vertex is that of the original edge.

A graph is denoted as a tuple $G = \{V_G, E_G, L\}$, where V_G and E_G are the set of vertices and edges, respectively. L is the function mapping a label, i.e., $L(v)$, to a vertex $v \in V_G$. Table 2 lists the frequently used notations in the paper.

2.1 Problem Definition

Definition 2.1 (Subgraph Isomorphism). *Given a query graph $Q = \{V_Q, E_Q, L\}$ and a data graph $G = \{V_G, E_G, L\}$, Q is subgraph isomorphic to G if there exists an injective function $f: V_Q \rightarrow V_G$, such that*

(1) $\forall\, u \in V_Q$, we have $L(u) = L(f(u))$ where $f(u) \in V_G$, and
(2) $\forall\, e(u_1, u_2) \in E_Q$, we have $e(f(u_1), f(u_2)) \in E_G$.

Subgraph matching returns all subgraphs of G that are isomorphic to Q. Each match can be expressed as a set of one-to-one matching pairs $\{(u \leftrightarrow f(u))\}$. All the matches are denoted by M.

Definition 2.2 (Streaming Graph). *A graph G is called a streaming graph if it changes dynamically following a sequence of update operations ΔG including four kinds of updates, i.e., vertex addition/deletion and edge addition/deletion.*

Since the query graph in our task is connected, the newly added vertex, clearly an isolated vertex in the data graph, cannot be included in any subgraph matching result. The vertex deletion only happens to isolated vertices, and the deletion of a non-isolated vertex can be taken as deleting all its connected edges. Hence, we just consider edge addition and deletion by convention [32].

Problem Statement 1. *(Continuous Subgraph Matching, shorted as CSM). Given a query graph Q, a data graph G, and a graph update stream ΔG, the continuous subgraph matching task is to find the*

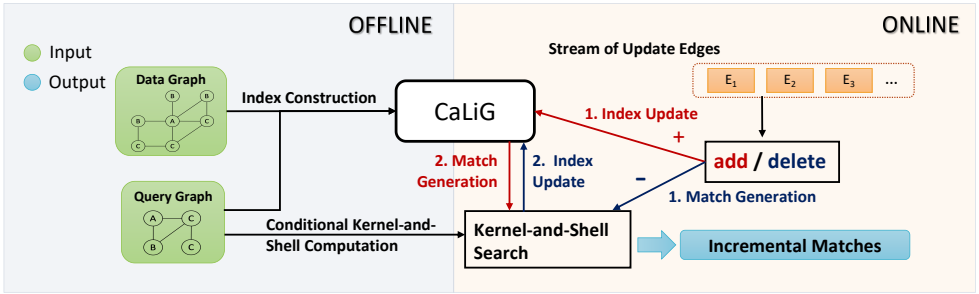


Fig. 3. Overview of our approach.

incremental matches (i.e., newly added or decreased subgraph matches) of Q for each update operation in ΔG.

Example 2.1. *Let us consider the data graph in Figure 1(b). Assume that there are two update operations, i.e., edge deletion $e(v_4, v_6)$ followed by edge addition $e(v_2, v_6)$. The decreased match is the left in Figure 1(c), owing to deleting the edge $e(v_4, v_6)$. After adding the edge $e(v_2, v_6)$, a new match (right in Figure 1(c)) emerges.*

2.2 Overview of Our Approach

Figure 3 presents the framework of the proposed approach, consisting of the offline indexing phase and online querying phase.

In the offline phase, we develop a novel index **Ca**ndidate **Li**ghting **G**raph, shorted as CaLiG. The index structure CaLiG is constructed according to the input query graph Q and data graph G.

Definition 2.3 (Matching Pair). *A pair of vertices u and v form a matching pair, shorted as (u, v)-MP, if $L(u) = L(v)$. Each (u, v)-MP has a lighting state "ON" or "OFF", indicating whether v can match u or not, respectively. Let (u, v)-MP.state denote the lighting state.*

CaLiG organizes candidate matching pairs in a vertex-pair graph, where each node represents a pair (u, v) of query vertex u and data vertex v. CaLiG imports the lighting state ("ON" or "OFF") to indicate whether $v \in V_G$ matches $u \in V_Q$ or not, i.e., whether v is a candidate of u. Benefiting from capturing all the connecting relations in both the query graph and data graph, it is easy to maintain tighter candidates and support incremental match generation. In addition, we compute a particular conditional kernel set and a shell set for each edge in the query graph.

In the online phase, the system responds differently according to the received update operation. For the edge addition, the index CaLiG is updated first to obtain new candidates, and then the subgraph matching is performed (as marked using red lines in Figure 3); For edge deletion, the subgraph matching is conducted first, and then CaLiG is updated (as marked using deep blue lines in Figure 3).

CaLiG update. To support incremental subgraph matching for subsequent updates, the index CaLiG needs to be maintained in real-time. Either adding or deleting an edge may lead to candidate updates and CaLiG changes including both the structure and the lighting states. Moreover, the changed state of one pair could propagate to other matching pairs, starting recursive propagations over CaLiG.

Incremental match. In order to complete the incremental match over the streaming graph efficiently, we design a new matching paradigm, kernel-and-shell search (KSS) in the subgraph matching phase. KSS divides the vertices of the query graph into conditional kernel vertices and

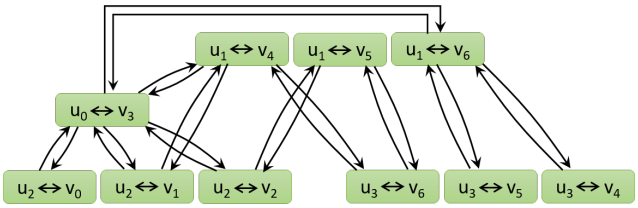


Fig. 4. CaLiG for the query graph and data graph in Figure 1, where the lighting states of all MPs are "ON" (as filled with green color).

shell vertices. It first takes candidates of the update edge as initially partial matches and then expands the matches to cover all the kernel vertices. Finally, the incremental matches can be produced easily by joining candidates of the shell vertices without any backtrackings.

3 CALIG: CANDIDATE LIGHTING GRAPH

To reduce the intermediate results (i.e., failures and backtracking in the search), we resort to a carefully designed index CaLiG.

3.1 CaLiG Structure

According to the definition of subgraph isomorphism, the vertex $v \in V_G$ matching $u \in V_Q$ or not depends on whether neighbors of v matches neighbors of u. Therefore, CaLiG organizes the matching pairs (u, v)-MP as a graph over which the lighting states can be easily maintained.

Definition 3.1 (CaLiG Index). *The CaLiG index for Q and G is a directed graph where each node represents an MP. There is a directed edge from (u_i, v_j)-MP to (u_k, v_l)-MP if it holds that*

1) $e(u_i, u_k) \in E_Q$ and $e(v_j, v_l) \in E_G$; and,
2) (u_i, v_j)-MP is "ON" or (u_i, v_j)-MP is turned "OFF" after (u_k, v_l)-MP.

In the CaLiG, we use the term "node" to distinguish vertices in Q or G. Note that the proposed index is designed for continuous subgraph matching and the lighting state of each matching pair may be updated according to the graph updates. Initially, the lighting states of all nodes in CaLiG are "ON", as all matching pairs are not pruned at first.

Example 3.1. *Figure 4 shows the CaLiG Index constructed based on the query graph and data graph in Figure 1. The query graph has 1 A vertex, 1 B vertex, and 2 C vertices, while the data graph has 1 A vertex, 3 B vertices, and 3 C vertices, so CaLiG contains $1 \times 1 + 1 \times 3 + 2 \times 3 = 10$ nodes. Since $e(u_1, u_2) \in E_Q$, $e(v_1, v_4) \in E_G$, and (u_1, v_4)-MP is "ON", there is an edge from (u_1, v_4)-MP to (u_2, v_1)-MP.*

Algorithm 1 illustrates the process of constructing CaLiG and refining the lighting states of matching pairs (turning OFF the nodes that do meet the matching requirements). First, we generate the matching pairs (nodes) of CaLiG (lines 2-5), where the initial lighting state of each matching pair is ON (line 4). Next, we generate the edges in CaLiG (lines 6-9). For each node (u, v)-MP in CaLiG, we traverse all neighbors of u in Q and all neighbors of v in G. If the pair of neighbors (u', v') forms a matching pair in CaLiG, an edge from (u', v')-MP to (u, v)-MP is added as all the matching pairs are "ON" initially. Last, we need to initialize the structure of CaLiG by updating the lighting states (line 10), which will be introduced in detail in Section 3.3. Note that after CaLiG is initialized, the incremental matches if any must be embedded in all ON-state nodes.

3.2 Lighting State Computation

Based on the definition of subgraph isomorphism (Definition 2.1), we know that v matches u only if v's neighbors match u's neighbors as well. The existing filter rules examine the existence of candidates for the neighbors of each query vertex, but ignore the injective requirement necessary for subgraph isomorphism. Although each neighbor of u can find a candidate in v's neighbors, v's neighbors may still fail to match u's neighbors when one neighbor of v is taken as candidates of multiple neighbors of u, violating the injective requirement. Thus, to compute the lighting state of nodes in CaLiG, we introduce a bipartite graph for each matching pair.

Definition 3.2 (Bigraph for (u, v)-MP). *A bigraph for (u, v)-MP, denoted by $BI(u, v)$, is a bipartite graph with two disjoint sets of vertices $N_Q(u)$ and $N_G(v)$, where there is an edge between $u_i \in N_Q(u)$ and $v_j \in N_G(v)$ if (u_i, v_j)-MP is an in-neighbor of (u, v)-MP in the CaLiG.*

For a matching pair (u, v), to determine whether v matches u, we can just consider the in-neighbors of (u, v)-MP in CaLiG, rather than examine all the vertex pairs $\{(u_i, v_j) | u_i \in N_Q(u) \wedge v_j \in N_G(v) \wedge L(u_i) = L(v_j)\}$.

Example 3.2. *Let us consider (u_1, v_6)-MP and (u_1, v_4)-MP in Figure 1. Their bigraphs are presented in Figure 5. The neighbor u_2 of u_1 does not have any candidates, which means when we take $(v_6 \leftrightarrow u_1)$ as a partial match, no candidates will be available for u_2, and this partial matching will fail. That is, v_6 should not be a candidate of u_1. Thus, the lighting state of the (u_1, v_6)-MP should be "OFF". Clearly, precise lighting states of MPs benefit tight candidates, improving the time efficiency of finding incremental matches.*

Based on subgraph isomorphism, v's neighbors can match u's neighbors only if there is an injective matching for $BI(u, v)$.

Definition 3.3 (Injective Matching). *Given a bigraph with two disjoint sets of vertices X and Y, there is an injective matching if each vertex u in X is matched against a vertex in Y via an edge and all the edges in the matching are independent of each other, i.e., all edges do not share vertices.*

For a (u, v)-MP, if its bigraph does not have an injective matching for $N_Q(u)$, at least one vertex $u_i \in N_Q(u)$ cannot be matched. We can set the lighting state "OFF" safely.

Algorithm 1: ConstructCaLiG(G, Q)

Input: A data graph G, a query graph Q
Output: A candidate lighting graph *CaLiG*

```
1  CaLiG ← GenerateEmptyCaLiG();
2  for each (u, v) ∈ (V_Q, V_G) do
3      if L(v) = L(u) then
4          (u, v)-MP.state ← ON;
5          add a node (u, v)-MP into CaLiG;
6  for each (u, v)-MP ∈ CaLiG do
7      for each (u', v') ∈ (N_Q(u), N_G(v)) do
8          if (u', v')-MP ∈ CaLiG then
9              add an edge from (u', v')-MP to (u, v)-MP;
10 IndexInitialization(CaLiG);
11 return CaLiG;
```

(a) No injective matching in the bigraph $BI(u_1, v_6)$ of (u_1, v_6)-MP.

(b) Having an injective matching in the bigragh $BI(u_1, v_4)$ of (u_1, v_4)-MP.

Fig. 5. Bigraphs for (u_1, v_6)-MP and (u_1, v_4)-MP respectively, where blue edges indicate an injective matching.

Lemma 3.1. *The bigraph as defined in Definition 3.2 is sufficient to determine the lighting state of the matching pair (u, v)-MP. Given a (u, v)-MP, its lighting state is "OFF" only if there is no injective matching for $N_Q(u)$ in the bigraph $BI(u, v)$; Otherwise, the state is "ON".*

Proof. The edges of the bigraph represent in-neighbors of (u, v)-MP in CaLiG. According to Definition 3.1, no matter what the state of (u, v)-MP is, the non-in-neighbors of (u, v)-MP are definitely OFF. An OFF-state node means it does not belong to any match, so we do not need to take the OFF-state node into account when we determine the state of (u, v)-MP. At least one vertex $u_i \in N_Q(u)$ cannot be matched if $BI(u, v)$ does not have any injective matching for $N_Q(u)$, leading to the OFF state of (u, v)-MP. □

Example 3.3. *As shown in Figure 5(a), $BI(u_1, v_6)$ does not have any injective matching, so we change the state of (u_1, v_6)-MP to OFF. In Figure 5(b), there is an injective matching as marked in blue lines, so the state of (u_1, v_4)-MP remains ON.*

Computing the injective matching for a bigraph (also called bipartite graph) could employ Hopcroft–Karp algorithm [21] with the time complexity of $O(\sqrt{|V|}|E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the bigraph. In practice, it just matters whether such an injective matching exists or not rather than what the injective matching is. There are two straightforward cases in which no injective matching exists.

-Case 1: One vertex u_i in $N_Q(u)$ has no incident edge in the bipartite graph, meaning has no candidate.

-Case 2: The vertices in $N_G(v)$ having at least one incident edge in the bipartite graph is less than $|N_Q(u)|$.

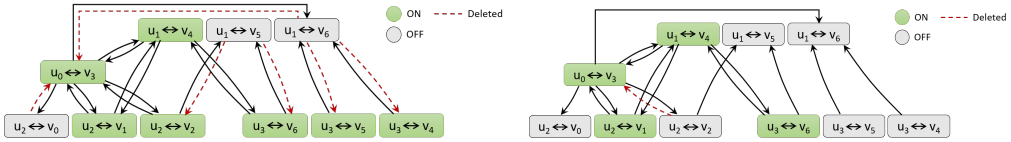
In the second case, we constrain the vertices having at least one incident edge since each edge represents a relation of candidate and the bipartite graph may change with the stream of edge updates.

Computing the injective matching is closely related to Hall's marriage theorem [15].

Theorem 3.2. *(Hall's Marriage Theorem [15]) Given a bigraph with two disjoint sets of vertices X and Y, for a subset W of X, let $N(W)$ denote the neighbors of W, i.e., a subset of Y in which vertices are connected to some vertices of W. There is an injective matching if and only if for every subset W of X we have $|W| < |N(W)|$. In other words, every subset W of X has sufficiently many neighbors in Y.*

The two cases above are indeed special cases of Hall's marriage theorem, where $W = \{u_i\}$ in Case 1 and $W = X$ in Case 2.

Lemma 3.3. *The vertex pairs connected by the edges in $BI(u, v)$ form a subset of $\{(u_i, v_j) | u_i \in N_Q(u) \wedge v_j \in N_G(v) \wedge L(u_i) = L(v_j)\}$.*



(a) Step 1. Turn off all nodes that do not have any injective matchings.

(b) Step 2. OFF-Propagation.

Fig. 6. CaLiG Initialization.

Proof. The proof is straightforward according to definitions of CaLiG (Definition 3.1) and bigraph (Definition 3.2). □

Generally, the time cost of computing an injective matching can be reduced if the bigraph of (u, v)-MP contains fewer edges. It is ideal to just keep the necessary edges corresponding to in-neighbors of (u, v)-MP in CaLiG. The in-neighbors of (u, v)-MP could be further reduced through a propagation mechanism next.

3.3 CaLiG Initialization

In CaLiG, the lighting state of a node is determined by the states of its neighbors. When the lighting state of a node (u, v)-MP is turned "OFF", i.e., v is pruned from the candidate set of u, the neighbors of (u, v)-MP would be affected. If one of the neighbors is turned "OFF", its neighbors would be affected recursively. Such a process is called OFF-state propagation which refines the candidates in CaLiG initialization.

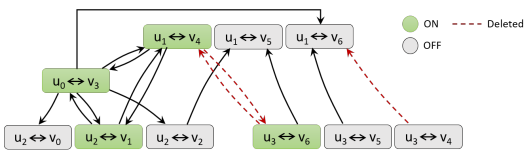
OFF-State Propagation. By using Lemma 3.1, the lighting state of each Matching Pair can be determined initially (i.e., the first round checking). The OFF-state matching pairs cannot be included in any subgraph matches. Therefore, the edges corresponding to these OFF-state matching pairs in the bigraphs of other matching pairs can be deleted safely, which may further turn off a set of matching pairs. The procedure proceeds iteratively until no matching pairs can be newly turned off. Finally, the ON-state matching pairs straightforwardly produce the candidates. The index CaLiG finishes initialization, ready for handling graph updates.

Algorithm 2 outlines the initialization process, where the procedure in line 6 (Algorithm 3) can be viewed as propagation on CaLiG. Let $Out_{CaLiG}(u, v)$ denote out-neighbors of (u, v)-MP in CaLiG. For each node (u', v')-MP whose lighting state is ON (line 2), we remove the edge from (u, v)-MP to (u', v')-MP (line 3) as (u, v)-MP will not contribute to the state of (u', v')-MP. Then we check the updated bigraph $BI(u', v')$. If (u', v')-MP is turned off, the procedure OFF-Propagation will be invoked recursively (lines 5-7).

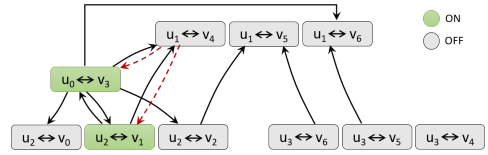
Algorithm 2: IndexInitialization(*CaLiG*)

Input: A candidate lighting graph *CaLiG*

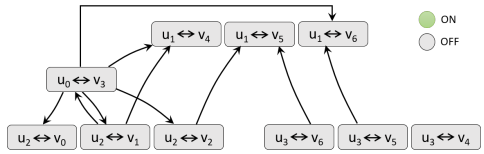
1 **for** *each* (u, v)*-MP* $\in$ *CaLiG* **do**
2 build a bigraph $BI(u, v)$ for (u, v)-MP;
3 **for** *each* (u, v)*-MP* $\in$ *CaLiG* **do**
4 **if** (u, v)*-MP.state* $=$ *ON and* $BI(u, v)$ *has no injective matching* **then**
5 (u, v)-MP.state $\leftarrow$ *OFF*;
6 OFF-Propagation(*CaLiG*, (u, v)-MP);



(a) Step 1. Delete all update edges in CaLiG.



(b) Step 2. Two nodes are turned off.



(c) Step 3. OFF-Propagation.

Fig. 7. CaLiG update for deleting the edge (v_4, v_6).

Example 3.4. *Let us consider the CaLiG in Figure 4. Since the bigraphs of (u_2, v_0)-MP, (u_1, v_5)-MP, and (u_1, v_6)-MP do not have any injective matching, they will be turned off in the first round. At the same time, we remove the out-going edges (red dotted edges) as shown in Figure 6(a). In the second round, as shown in Figure 6(b), (u_2, v_2)-MP, (u_3, v_5)-MP and (u_3, v_4)-MP are turned off. In this round, only the edge from (u_2, v_2)-MP to (u_0, v_3)-MP is deleted as (u_0, v_3)-MP is the only ON-state out-neighbor. Finally, no node can be further turned off and only nodes (u_0, v_3)-MP, (u_2, v_1)-MP, (u_3, v_6)-MP, and (u_1, v_4)-MP are ON. In fact, these four matching pairs can just form a subgraph match of the query graph.*

The lighting process, i.e., CaLiG initialization, only needs to propagate to the "ON"-state nodes. For each ON node (u, v)-MP, its OFF in-neighbors have been deleted when the in-neighbors are turned off in the previous round, so the remaining matching pairs are all ON nodes. Then the state of (u, v)-MP will be re-checked by finding the injective matching. In short, the process of OFF propagation will only propagate among ON-state nodes, and the nodes that have been previously turned off will not be checked repeatedly.

Compared with SymBi. SymBi builds a query DAG and refines the candidates based on weak embeddings. It focuses on the existence (i.e., mapping from $N_Q(u)$ to $N_Q(v)$) such that the reused candidates could not be excluded. In contrast, CaLiG computes the injective matching from $N_Q(u)$ to $N_Q(v)$) and propagates along the edges in CaLiG.

Hence, *the candidates computed by CaLiG are equal to or tighter than SymBi.*

Algorithm 3: OFF-Propagation(*CaLiG*, (u, v)-MP)

Input: *CaLiG* and a matching pair (u, v)-MP that was turned off in the previous round

```
1 for each (u', v')-MP ∈ Out_CaLiG(u, v) do
2     if (u', v')-MP.state = ON then
3         delete the edge from (u, v)-MP to (u', v')-MP;
4         update BI(u', v');
5         if BI(u', v') has no injective matching then
6             (u', v')-MP.state ← OFF;
7             OFF-Propagation(CaLiG, (u', v')-MP);
```

Lemma 3.4. *The candidates computed by CaLiG are equal to or tighter than SymBi.*

Proof. SymBi proposes an index $D_i[u, v]$ to maintain the dynamic candidate space. The recurrence process is described as:

- $D_1[u, v] = 1$ iff $\exists v_p$ adjacent to v such that $D_1[u_p, v_p] = 1$ for every parent u_p of u in DAG $\hat{q}$;
- $D_2[u, v] = 1$ iff $D_1[u, v] = 1$ and $\exists v_c$ adjacent to v such that $D_1[u_c, v_c] = 1$ for every parent u_p of u in DAG $\hat{q}$.

SymBi considers $\{v_i | D_2[u, v] = 1\}$ as the candidates of query vertex u. A tighter filter rule could be obtained by extending the number of filtering iterations from only 2 to until the candidate space converges and extending the vertex needed to check from parent/children to all the neighbors, i.e.

- The vertex v in $Cand(u)$ iff v_n adjacent to v such that v_n in $Cand(u)$ for every neighbor u_p of u in q.

which is the third constraint of CaLiG candidate space. Then The candidates computed by CaLiG are at least equal to SymBi. □

Complexity Analysis. The number of nodes in CaLiG is $O(|V_G| \times |V_Q|)$ and the number of edges is $O(|E_G| \times |E_Q|)$. Correspondingly, the bigraph of node (u, v)-MP in CaLiG is a complete bipartite graph in the worst case, and the storage cost is $O(|N_Q(u)| \times |N_G(v)|)$, where $|N_Q(u)|$ and $|N_G(v)|$ are degrees of $u \in Q$ and $v \in G$, respectively. Thus, the overall space cost is $O(|E_G| \times |E_Q| + \sum_{(u,v) \in CaLiG} |N_Q(u)| \times |N_G(v)|) = O(|E_Q| \times |E_G|)$. In the CaLiG initialization, each edge in CaLiG is visited at most once and the injective matching is computed for the incident nodes with the time cost $O(d_Q^{2.5})$, where d_Q is the maximum vertex degree of Q. Thus, the overall time complexity is $O(|E_Q| \times |E_G| + |E_Q| \times |E_G| \times d_Q^{2.5}) = O(|E_Q| \times |E_G| \times d_Q^{2.5})$.

4 DYNAMIC UPDATE OF CALIG

Generally, it is expected to produce tight candidates, taking as little time as possible. Benefiting from CaLiG, it is easy to update candidates by simply exploring CaLiG from the incident nodes of newly added or deleted edges. The core idea is that edge addition or deletion may cause the state alteration of some nodes, which would further propagate over CaLiG. To resolve the updates, we present how to address edge deletion (in Section 4.1) and edge addition (in Section 4.2), respectively.

4.1 Edge Deletion

Deleting an edge from the data graph G may make some candidates fail to match the query vertices (turning off some matching pairs in CaLiG), decreasing the subgraph matches. When one edge is deleted from G, we first delete all the related edges from CaLiG and adopt the OFF-Propagation to refine the candidates by updating the lighting states.

To be specific, let $e(v_1, v_2)$ denote the data edge (the edge in data graph G) to be deleted. Since $e(v_1, v_2)$ may be a candidate of any query edge (the edge in query graph Q) $e(u_1, u_2)$ with the same label, it will result in edge deletions in CaLiG. The process is similar to CaLiG initialization and resorts to the OFF-Propagation (Algorithm 3). The details of handling edge deletions are outlined in Algorithm 4. First, it deletes the edges between (u_1, v_1)-MP and (u_2, v_2)-MP from CaLiG (lines 2-3). Then we check the two affected nodes (u_1, v_1)-MP and (u_2, v_2)-MP. If they are turned off due to the deleted edge $e(v_1, v_2)$, the OFF propagation will be invoked to further refine other nodes.

Example 4.1. *Take Figure 1 as an example, where the edge (v_4, v_6) is deleted. As shown in Figure 7(a), we delete (1) the edges between (u_1, v_4)-MP and (u_3, v_6)-MP; and (2) the edge from (u_3, v_4)-MP to (u_1, v_6)-MP. The nodes (u_1, v_4)-MP and (u_3, v_6)-MP are originally ON, so we perform bigraph checking separately. As shown in Figure 7(b), neither of the nodes has an injective matching in the bigraph due*

to edge deletion, so they will be turned off and trigger the OFF-Propagation. Finally, all nodes will eventually be turned off, as shown in Figure 7(c), indicating that there is no match for the query.

Complexity Analysis. Let D represent the set of nodes turned off because of updating. For each node in D, we compute the injective matching for the bigraph $BI(u, v)$ and propagate the updating processing to its neighbors. Therefore, the time complexity of updating CaLiG for deleting an edge is $O(\sum_{(u,v)\in D}(|N_{CaLiG}(u, v)| + |N_Q(u)|^{2.5}))$, where $N_{CaLiG}(u, v)$ is the set of neighbors of (u, v)-MP in CaLiG. It seems that the time cost is a little bit high, but it is sensible considering the benefit of reducing cumbersome backtrackings, that is, updating CaLiG is cost-effective.

4.2 Edge Addition

Adding an edge to the data graph brings opportunities for turning on the OFF-state matching pairs. Based on CaLiG, it is not difficult to deliver the new candidates. Intuitively, the updating process for edge addition is similar to that for edge deletion. We first add all the related edges to GaLiG, and detect new candidates by ON-Propagation. Before presenting the details of addressing edge addition, two proprieties regarding CaLiG are introduced as follows.

Lemma 4.1. *For any node (u, v)-MP in CaLiG, all of its in-neighbors are either ON or turned OFF after (u, v)-MP.*

Proof. Assume (u', v')-MP is an in-neighbor of (u, v)-MP and it is turned off before (u, v)-MP (i.e., (u, v)-MP is ON when (u', v')-MP has been turned off). By the OFF-Propagation, the edge from (u', v')-MP to (u, v)-MP will be deleted once (u', v')-MP is turned off. Thus (u', v')-MP cannot be the in-neighbor of (u, v)-MP. □

As we mentioned in Section 3.1, the lighting states of all nodes are initialized to ON, and then gets updated by OFF-Propagation in the offline phase. However, this propagation process is irreversible, that is to say, if the turning off of (u_1, v_1)-MP leads to the turning off of (u_2, v_2)-MP, the turning on of (u_2, v_2)-MP because of edge addition will not turn (u_1, v_1)-MP on since (u_1, v_1)-MP has been turned off when (u_2, v_2)-MP is still ON.

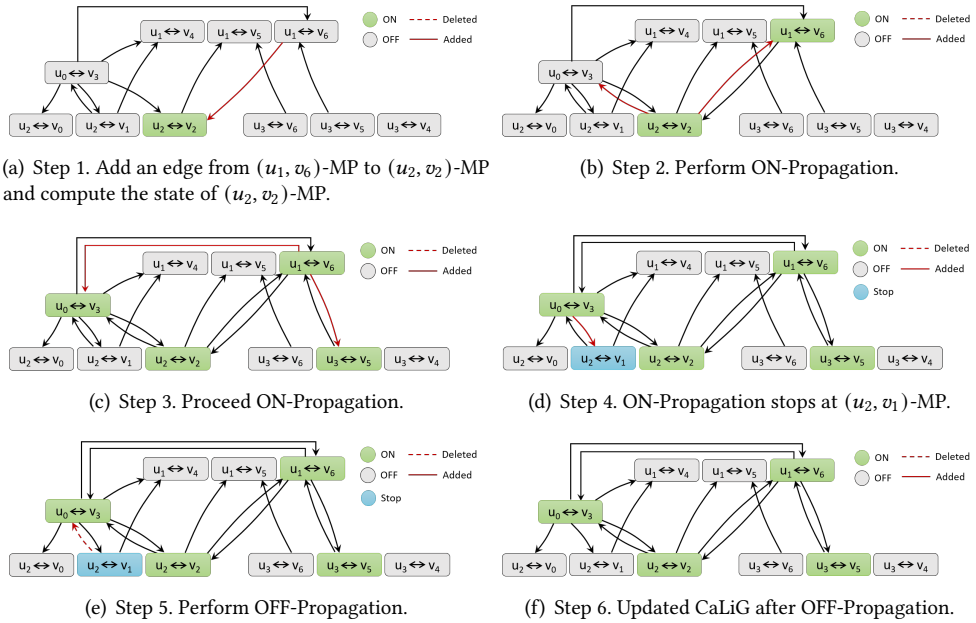
Lemma 4.2. *If there is an edge from the OFF-state (u, v)-MP to the OFF-state (u', v')-MP, the turning on of (u, v)-MP will not cause the turning on of (u', v')-MP.*

Proof. Assuming turning on (u, v)-MP causes turning on (u', v')-MP, (u', v')-MP would not be turned off before (u, v)-MP because (u, v)-MP was ON when computing its lighting state of (u', v')-MP, contradicting that (u', v')-MP was turned off before (u, v)-MP. □

Algorithm 4: UpdateCaLiGForDel(*CaLiG*, $e(v_1, v_2)$)

Input: *CaLiG* and an updated edge $e(v_1, v_2)$ to delete

```
1 for each e(u1, u2) ∈ E_Q do
2     if L(u1) = L(v1) and L(u2) = L(v2) then
3         delete edges between (u1, v1)-MP and (u2, v2)-MP from CaLiG;
4         if (u1, v1)-MP.state = ON and BI(u1, v1) has no injective matching then
5             (u1, v1)-MP.state ← OFF;
6             OFF-Propagation(CaLiG, (u1, v1)-MP);
7         if (u2, v2)-MP.state = ON and BI(u2, v2) has no injective matching then
8             (u2, v2)-MP.state ← OFF;
9             OFF-Propagation(CaLiG, (u2, v2)-MP);
```



(a) Step 1. Add an edge from (u_1, v_6)-MP to (u_2, v_2)-MP and compute the state of (u_2, v_2)-MP.

(b) Step 2. Perform ON-Propagation.

(c) Step 3. Proceed ON-Propagation.

(d) Step 4. ON-Propagation stops at (u_2, v_1)-MP.

(e) Step 5. Perform OFF-Propagation.

(f) Step 6. Updated CaLiG after OFF-Propagation.

Fig. 8. CaLiG updates for adding the edge (v_2, v_6).

Let $e(v, v')$ denote the edge to add. According to Lemma 4.1 and Lemma 4.2, to determine whether turning on matching pairs (u, v)-MP or (u', v')-MP, where $L(u) = L(v)$ and $L(u') = L(v')$, we only need to check its in-neighbors in CaLiG rather than considering all the vertex pairs $\{(u_i, v_j) | u_i \in N_Q(u) \wedge v_j \in N_G(v) \wedge L(u_i) = L(v_j)\}$ or $\{(u_i, v_j) | u_i \in N_Q(u') \wedge v_j \in N_G(v') \wedge L(u_i) = L(v_j)\}$. The nodes that are not in-neighbors of (u, v)-MP or (u', v')-MP in CaLiG were turned off before (u, v)-MP or (u', v')-MP. Likewise, they will not be turned on even if (u, v)-MP or (u', v')-MP is turned on.

Algorithm 5: UpdateCaLiGForAdd($CaLiG$, $e(v_1, v_2)$)

Input: $CaLiG$ and an updated edge $e(v_1, v_2)$ to add

```
1  for each e(u1, u2) ∈ E_Q do
2      if L(v1) = L(u1) and L(v2) = L(u2) then
3          add an edge from (u2, v2)-MP to (u1, v1)-MP;
4          if (u1, v1)-MP.state = OFF and BI(u1, v1) has an injective matching then
5              (u1, v1)-MP.state ← ON;
6          if (u1, v1)-MP.state = ON then
7              StopSet ← ON-Propagation(CaLiG, (u1, v1)-MP);
8              while StopSet is not empty do
9                  (u, v)-MP ← StopSet.pop();
10                 OFF-Propagation(CaLiG, (u, v)-MP);
```

Algorithm 6: ON-Propagation($CaLiG$, (u, v)-MP)

Input: $CaLiG$ and a matching pair (u, v)-MP that was turned on in the previous round
Output: A set of stopping nodes S

```
1  S ← ∅;
2  for each (u', v')-MP ∈ In_CaLiG(u, v) do
3      add an edge from (u, v)-MP to (u', v')-MP;
4      if (u', v')-MP.state = OFF then
5          if BI(u', v') has an injective matching then
6              (u', v')-MP.state ← ON;
7              S ← S ∪ ON-Propagation(CaLiG, (u', v')-MP);
8          else
9              S ← S ∪ (u', v')-MP;
10 return S;
```

Algorithm 5 depicts the process of handling edge additions based on CaLiG. It involves two procedures ON-Propagation (Algorithm 6) and OFF-Propagation. Different from handling edge deletions, it just needs to add an edge from (u_2, v_2)-MP to (u_1, v_1)-MP and computes the lighting state of (u_1, v_1)-MP (lines 2-5), since its turning on will propagate to (u_2, v_2)-MP through the ON-Propagation. If the state of (u_1, v_1)-MP is OFF, the state of (u_2, v_2)-MP will not change no matter whether it is ON or OFF. For each ON-state (u_1, v_1)-MP, the procedure ON-Propagation is invoked to try to turn on more nodes. Note that before ON-Propagation we add an edge from (u_2, v_2)-MP to (u_1, v_1)-MP (line 3 in Algorithm 5), which indicates that we progressively take (u_2, v_2)-MP as an ON-state node. Hence, some nodes may be falsely turned on due to the propagation. When a node is not turned on by ON-Propagation, it is recorded as a stopping node. All the stopping nodes constitute the set, called $StopSet$. The OFF-Propagation is performed for each node in $StopSet$ to ensure that all the nodes that are newly turned on conforming to subgraph isomorphism.

Algorithm 6 presents the details of ON propagation. Based on Lemma 4.2, it only needs to propagate to its in-neighbor (u', v')-MP. If (u', v')-MP is OFF and its corresponding bigraph has an injective matching, we can turn (u', v')-MP off and invoke the ON-Propagation recursively. Otherwise, the node will be added into S, the set of stopping nodes.

Example 4.2. *Take Figure 8 as an example, where the edge (v_2, v_6) is added to the data graph, as shown by the red line in Figure 1(c). Firstly, as shown in Figure 8(a), an edge is added from (u_1, v_6)-MP to (u_2, v_2)-MP in CaLiG. Then (u_2, v_2)-MP is turned on. Secondly, we will perform ON-Propagation towards (u_2, v_2)-MP's in-neighbors as shown in Figure 8(b), where the nodes (u_1, v_6)-MP and (u_0, v_3)-MP are originally OFF. Only (u_2, v_2)-MP is turned on as it has an injective matching. Then the propagation continues to try to turn on its OFF-state in-neighbors. Correspondingly, (u_0, v_3)-MP and (u_3, v_5)-MP are turned on as shown in Figure 8(c). When propagating to (u_2, v_1)-MP, it is not turned on and marked as a stopping node as shown in Figure 8(d). Finally, the OFF-Propagation is invoked starting from the stopping node (u_2, v_1)-MP, but (u_0, v_3)-MP remains ON-state after the OFF-Propagation (Figure 8(e)). The final updated CaLiG index is depicted in Figure 8(f).*

Complexity Analysis. Let D and A represent the sets of nodes turned off and turned on because of edge addition, respectively. The time complexity for edge addition is almost the same as that for edge deletion. The difference is we need to take the procedure of turning nodes in A into consideration.

The overall time complexity of updating CaLiG by adding an edge is $O(\sum_{(u,v)\in(D\cup A)} |N_{CaLiG}(u,v)| + |N_Q(u)|^{2.5})$, where $N_{CaLiG}(u,v)$ is the set of neighbors of (u,v)-MP.

5 KSS-BASED SUBGRAPH MATCHING

For continuous subgraph matching, we need to find all the new matches due to edge addition or the decreased matches due to edge deletion. It is clear that the changed matches must contain the updated edges. In detail, for each update edge $e(v_i, v_j)$, we need to find two connected ON-state nodes (u_k, v_i)-MP and (u_l, v_j)-MP in CaLiG that contain v_i and v_j, respectively. These two nodes will be taken as a partial match which can be extended to a complete match. Thus, the widely used backtracking search could be utilized. Such a backtracking search enumerates candidates of each query vertex one by one and examines constraints for the subgraph isomorphism at each step. It performs intensive backtracks over the candidates to construct matches by trying to integrate the candidates, which incurs expensive time costs.

Most existing algorithms accelerate the backtracking by adjusting the matching order. In contrast, we seek a more powerful backtracking framework in this section. It is noticeable that the matches for the degree-one query vertices are independent, where the "independent" means that computing matches for one query vertex would not affect the matches of another. Therefore, the matches of such vertices could be acquired together instead of exploring them one by one.

Thus, we develop a novel backtracking search framework, called Kernel-and-Shell Search (KSS), for continuous subgraph matching. KSS decomposes the query vertices into kernel and shell vertices, making it straightforward to deliver the incremental matches by joining the partial matches for kernel vertices and the candidate of shell vertices.

5.1 Kernel and Shell Vertex

To identify the independent vertices as discussed above, KSS defines the kernel set as the connected vertex cover, and the shell set is the complementary set.

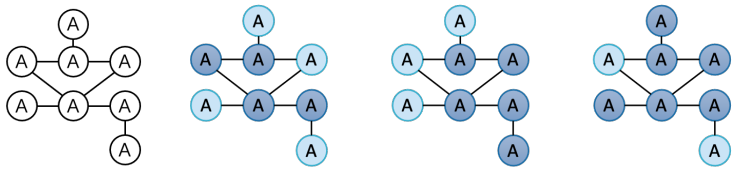
Definition 5.1 (Kernel Set and Shell Set). *Given a query graph Q, its kernel set, also known as connected vertex cover, is a set of connected vertices s.t. each edge in Q has at least one vertex in the set. Each vertex in the kernel set is called a kernel vertex. The shell set is the complementary set of the kernel set, where vertices are independent of each other and each of them is called a shell vertex.*

The shell vertices are naturally independent by definition. Since the query graph is fixed in CSM, the corresponding kernel and shell vertices could be computed and stored in advance.

Example 5.1. *Figure 9(b) lists three different kernel sets and shell sets for the query graph in 9(b), where vertices in dark blue form the kernel set and the other vertices form the shell sets.*

Given a query graph Q and its kernel set and shell set, once the match of kernel set is determined, the candidates for each shell vertex are obtained by checking its adjacent matched vertex. As there is no edge dependency among shell vertices, the matches can be produced straightforwardly by joining the candidates of shell vertices, without any failing backtracks. Therefore, a good decomposition should have as many shell vertices as possible, indicating as few kernel vertices as possible. As discussed above, the desired incremental matches must contain the update edge $e(v_i, v_j)$ and the search starts from $e(v_i, v_j)$. Hence, the subgraph induced by the kernel set should contain the edge $e(u_k, u_l)$ such that the matching pairs (u_k, v_i)-MP and (u_l, v_j)-MP are ON in CaLiG.

Definition 5.2 (Conditional Kernel Set). *Given a query graph Q and an edge $e(u_k, u_l) \in E_Q$, the conditional kernel set, denoted by CKS, is the kernel set that contains vertices u_k and u_l.*



(a) Query graph. (b) Three kernel sets and shell sets.

Fig. 9. An example of kernel vertices and shell vertices.

Lemma 5.1. *Given a query graph Q, we get a new query graph Q' by adding a virtual vertex u_0 and connecting u_0 with all vertices in Q. If $S \cup \{u_0\}$ is a minimum kernel set of Q', S is a minimum kernel set of Q, where $S \subseteq V_Q$.*

Proof. Suppose S is not a minimum kernel set of V_Q and there is another kernel set S' of Q that is smaller than S. $S' \cup \{u_0\}$ will be a smaller kernel set of Q', which leads to a contradiction. □

Theorem 5.2. *Computing the minimum conditional kernel set (MCKS) for a query graph Q and an edge $e(u_k, u_l) \in E_Q$ is NP-hard.*

Proof. The proof is achieved by reducing the NP-hard minimum connected vertex cover (MCVC) problem [13, 29].

Let Q' denote the graph obtained by adding a virtual vertex u_0 and connecting u_0 with all vertices in Q. For each edge in Q', we compute its MCKS. Let S' represent the minimum MCKS among the $|E_{Q'}|$ MCKSs. S' is an MCVC (i.e., a minimum kernel set) of Q', which can be proved by contradiction as follows.

Suppose S' is not the MCVC of Q'. There must be another MCVC of Q', denoted by S, such that $|S'| > |S|$. (1) When S does not contain u_0, all the nodes V_Q constitute the MCVC S of Q', i.e., $S = V_Q$. If u_0 is not contained in S', $|S'| \leq |V_Q|$, which contradicts $|S'| > |S|$. If u_0 is contained in S', $S' = S^* \cup \{u_0\}$ such that $S^* \geq S$. Thus, we have S'=$V_Q \cup \{u_0\}$. As $S = V_Q$, S is the MCKS for Q' and an edge $e(u_x, u_y)$, where $e(u_x, u_y) \in E_{Q'}$. It contradicts that S' is the smaller MCKS for Q'. (2) When S contains u_0, at least one neighbor u_i of u_0 will be contained in S as the kernel set is connected. S is the MCKS for Q' and the edge $e(u_0, u_i)$, which contradicts that S' is the minimum one among all the possible edges.

If $u_0 \in S'$, $S' \setminus u_0$ is a minimum kernel set of Q according to Lemma 5.1; Otherwise, S' is a minimum kernel set of Q. It is clear that the procedure of constructing the solution to MCVC is in polynomial time. Thus, if MCKS can be solved in polynomial time, the MCVC problem can be solved in polynomial time, which contradicts the NP-hardness of the MCVC problem. □

Given the MCKS is NP-hard to compute, a greedy algorithm is proposed in practice. It adds u_k and u_l into CKS and removes u_k, u_l, and their neighbors from Q. Then it enumerates a maximal collection of disjoint odd cycles in the remaining Q'. Let t denote the number of odd cycles, C_i denote the vertices of i-th cycle, and $C = \bigcup_i^t C_i$. It computes the optimal vertex cover of the remaining bipartite graph $Q' \setminus C$, denoted as W. If $C \cup W \cup \{e(u_k, u_l)\}$ is connected, return $C \cup W \cup \{e(u_k, u_l)\}$; otherwise, we apply approximation algorithms for Steiner tree to obtain one connected result.

5.2 Kernel-and-Shell Search

Powered by kernel and shell vertices, we develop a novel backtracking search framework. When an edge is added or deleted, our method would first initialize partial matches based on the updated edge (Algorithm 7), and then invoke the kernel and shell search (Algorithm 8). For kernel vertices,

Algorithm 7: FindMatches($CaLiG$, $e(v_1, v_2)$, $Kernel$, $Shell$)

Input: $CaLiG$, an updated edge $e(v_1, v_2)$, the conditional kernel set K, and shell set S for $e(v_1, v_2)$
Output: Incremental matches due to $e(v_1, v_2)$

```
1 for each u1 ∈ Q do
2     if L(v1) = L(u1) and (u1, v1)-MP.state = ON then
3         m[u1] ← v1;
4         for each u2 ∈ N_Q(u1) do
5             if (u2, v2)-MP ∈ In_CaLiG(u1, v1) then
6                 m[u2] ← v2;
7                 return KSS(m, K, S).
```

Algorithm 8: $KSS(m, K, S)$

Input: The partial match m, conditional kernel set K, and shell set S
Output: Incremental matches due to $e(v_1, v_2)$

```
1 if m.size < |K| then
2     th ← m.size;
3     u ← K[th];
4     Cand(u) ← generate u's candidates;
5     for each v ∈ Cand(u) do
6         m' ← m;
7         m'[u] ← v;
8         KSS(m', K, S);
9 else
10    for each u ∈ S do
11        Cand(u) ← generate u's candidates;
12    return m ⋈_{u∈S} Cand(u).
```

KSS computes the partial matches through the existing backtracking. The incremental matches can be reported immediately by joining the partial matches and the candidates of shell vertices, without any unnecessary backtrackings.

Algorithm 7 presents the process of finding incremental matches based on kernel-and-shell search (KSS). Given the update edge (v_1, v_2), as a matching pair (u_1, v_1)-MP must be ON (lines 2-3). The matching pair (u_2, v_2)-MP is then determined from the in-neighbors of (u_1, v_1)-MP (lines 4-6). Thus, we have found partial match $\{(u_1 \leftrightarrow v_1, u_2 \leftrightarrow v_2)\}$, denoted as m. Next, we match the remaining query vertices in the order of kernel vertex first and then shell vertex by invoking the procedure *KSS*.

Algorithm 8 describes how to search matches for conditional kernel and shell vertices. First, it determines the query vertex to match according to the kernel set (lines 2-3), and then generates candidates for the vertex. Finally, it selects a candidate as the partial match to drive the matching process (lines 5-8). After all the kernel vertices have been matched, we generate candidates for all shell vertices and join the candidates with the partial match to report the incremental matches

Table 3. Data graphs.

Dataset	Abb.	|V|	|E|	Average degree
Lastfm	lfm	7,624	27,806	7.3
Facebook	fbm	22,470	170,823	15.2
Email	em	36,692	183,831	10.0
Github	gh	37,700	289,003	15.3
Deezer	dz	41,773	125,826	6.0
Twitch	tw	168,114	6,797,557	80.9
Skitter	sk	1,696,415	11,095,298	13.1
Netflow	nf	3,114,895	16,668,683	10.7

(lines 10-12). The process is very efficient as we produce the incremental matches by a simple join operation without checking any constraints or backtracks.

Candidate generation. Let $Cand(u)$ denote the candidates of query vertex u. The vertex v in $Cand(u)$ meets the constraints:

1) v has not been used in the partial match.
2) The node (u, v)-MP in CaLiG is ON.
3) v is a neighbor of v' if $v' \in Cand(u')$ and $u' \in N_Q(u)$.

To generate $Cand(u)$ for vertex u, two steps are conducted.

Step 1. For each $u_i \in N_Q(u)$ that been matched to v_i, we build a set $C_i = \{v' | (u', v')\text{-MP} \in In_{CaLiG}(u_i, v_i)$ and (u', v')-MP is ON$\}$. Then we intersect all of these sets C_i to get a candidate set for u.

Step 2. We remove the data vertices that have already been used in the partial matching from the candidate set. After this step, we obtain the final candidate set of u.

Pruning in advance. KSS computes the partial matches for kernel vertices first, which facilitates match generation while incurring new problems. For example, when we determine a partial match of all kernel vertices, the candidate set of a shell vertex may be empty, but we can only find this failed partial match when generating candidates for that shell vertex. Hence, we employ a pruning strategy that detects such failures as early as possible. For each shell vertex u, once all its neighbors have been matched, we try to find a candidate for u in advance. If no candidate vertex is found for u, it is safe to rule out the current partial match. Otherwise, it proceeds to find matches of the remaining kernel vertices.

Complexity Analysis. For KSS, only the kernel part is matched by backtracking search, the worst-case complexity is induced to $O(|V_G|^{|K|})$, where $|K|$ is the size of kernel vertices, much smaller than traditional backtracking with the time complexity $O(|V_G|^{|V_Q|})$.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate our proposed method, denoted by CaLiG, and compare it with two state-of-the-art algorithms TurboFlux [24] and SymBi [32].

6.1 Experimental Settings

Data graphs. Table 3 lists the graphs used in experiments that are downloaded from SNAP [28], except Netflow (downloaded from CAIDA [1]). We randomly sample 5% data edges as the streaming edges, with the ratio of deleted edges over added edges being 2:1.

Table 4. Average elapsed time.

Dataset	Elapsed Time (ms)			$\|\Delta m\|$
	TurboFlux	SymBi	CaLiG	
Lastfm	49,341.9	13,848.3	**641.6**	7,626,438
Facebook	2,510,220.9	1,745,489.0	**94,174.2**	443,817,336
Email	2,836,668.9	1,873,555.0	**82,417.3**	338,293,270
Github	4,177,493.4	3,411,865.8	**189,060.7**	1,679,993,941
Deezer	368,280.5	99,535.6	**4,669.0**	14,327,860
Twitch	1,957,440.8	1,107,821.3	**51,247.5**	464,933,035
Skitter	1,460,160.6	321,649.7	**9,447.0**	130,260,509
Netflow	727,494.2	119,694.9	**122.3**	48,801

Query graphs. For each data graph, we sample 7 groups of subgraphs, denoted by $Q3$, $Q4$, $Q6$, $Q8$, $Q10$, $Q12$, $Q14$, and $Q16$, as query graphs by varying the number of vertices from 3 to 16. Each group Qi contains 50 query graphs.

Metrics. The elapsed time is reported in milliseconds. Due to the NP-hardness of subgraph matching, some queries may take an extremely long time. By convention, we set a timeout of 20 minutes for each query. If one query cannot finish within the time limit, it is called *uncompleted*. To evaluate an algorithm generally, we report the average elapsed time, as well as the completion rate, the average peak memory usage, and the match density.

All the algorithms are implemented in C++ and evaluated on a Linux Server equipped with Intel(R) Xeon(R) CPU E5-2640 @ 2.60GHz and 128G RAM.

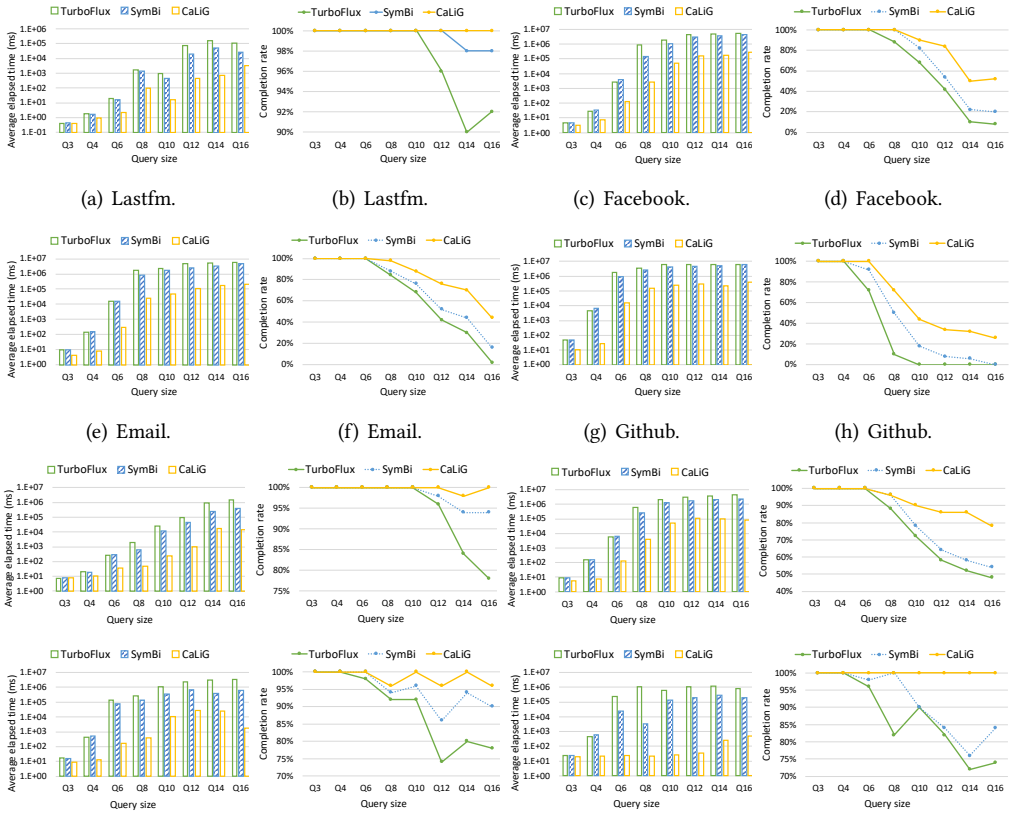
6.2 Experimental Results

Overall Performance. Table 4 reports the average elapsed time of 400 query graphs on 8 data graphs respectively. Table 4. As is observed, It shows that CaLiG runs much faster than TurboFlux and SymBi in all graphs, achieving significant speedups over TurboFlux (from 22.10x to 5963.07x) and SymBi (from 18.05x to 978.69x) respectively. The last column $|\Delta m|$ represents the number of incremental matches. We can find that $|\Delta m|$ ranges from millions to more than 1 billion. The larger the number of results, the more time it takes.

Varying the size of query graph. The effect of query size (i.e., the number of vertices) is evaluated by varying $|V_Q|$ from 3 to 16. Figure 10 shows the detailed results, where each dataset has two sub-figures representing the average elapsed time and the completion rate for each group of queries in the same size. We can see that CaLiG exhibits significant advantages over the competitors.

For all data graphs, CaLiG always approximately outperforms SymBi one to three orders of magnitude faster in terms of time efficiency, while achieving a higher completion rate. The average speedup on the group of queries $Q12$ reaches 5472x (on Netflow). Moreover, CaliG achieves better speedups over TurboFlux. As the query size grows, the time cost increases as well. For those larger queries, the completion rate of CaLiG is significantly higher.

Varying the size of data graph. To evaluate the scalability, we randomly generate 7 data graphs of different sizes through LSBench [27]: 0.1, 0.2, 0.5, 1, 2, 5, and 10 million vertices, while the other characters of the data graphs are the same. Two groups of query graphs with 8 and 12 vertices are used in this experiment. The results are shown in Figure 11(a). The dotted lines represent the results of the 8-vertex query graphs, and the solid lines represent the results of the 12-vertex query graphs. Compared with TurboFlux and SymBi, CaLiG is consistently faster regardless of the size of



(a) Lastfm. (b) Lastfm. (c) Facebook. (d) Facebook.

(e) Email. (f) Email. (g) Github. (h) Github.

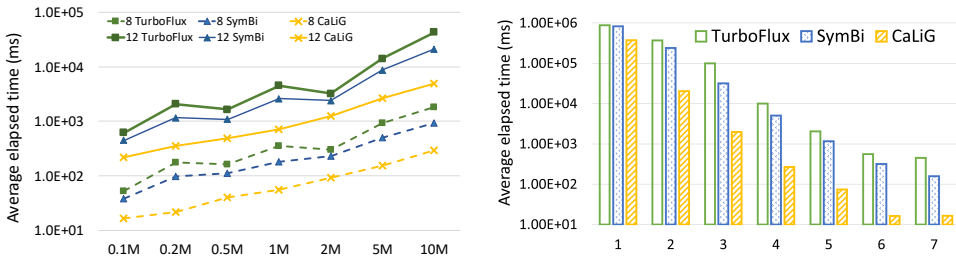
(m) Skitter. (n) Skitter. (o) Netflow. (p) Netflow.

Fig. 10. Varying query graph in different data graphs.

the data graph. In addition, the elapsed time of all algorithms tends to increase with the growth of the dataset size.

Varying the number of vertex labels for data graph. We evaluate the effect number of labels by randomly assigning labels to vertices. Totally 7 varied graphs, with the number of labels 1, 2, 3, 4, 5, 7, and 9, are generated for each original data graph. Figure 11(b) presents the results on the data graph *Deezer*. We can see that the CaLiG algorithm is 2.2 to 19.7 times faster than SymBi, and 2.4 to 50.5 times faster than TurboFlux. The speedup increases with the growth of the number of labels. In addition, we can find that the number of labels affects the elapsed time. Basically, the time cost is inversely proportional to the number of labels, that is, the more labels, the less elapsed time for continuous subgraph matching.

Varying the scale of streaming data. We explore how the stream size affects our algorithm next. Two sets of experiments are set up respectively, one set to test the mixed update flow, that is, there are deleted edges and added edges in the update flow, and the ratio of them is 2:1. In the other set, only deleted edges are allowed in the stream. The two sets of update streams are randomly sampled from the original data edges, and their scales are set to 2%, 4%, 6%, 8%, and 10% of the number of data edges, respectively. Figure 12(a) reports the results on the data graph *Email*. It shows that the performance is linear to the scale of update streams, whether it is a mixed stream or



(a) Varying the scale of data graph.

(b) Varying the number of vertex labels.

Fig. 11. Varying number of vertex labels and the scale of the data graph.

Table 5. Match Density under different query sizes

Method	Dataset	4	6	8	10	12	14
TurboFlux	*dz*	0.530	0.240	0.040	0.048	0.021	0.0001
	lfm	1.858	1.190	0.152	0.005	0.002	0.002
	sk	4.562	1.514	0.214	0.123	0.119	0.009
SymBi	*dz*	0.557	0.392	0.252	0.110	0.024	0.0009
	lfm	2.012	1.265	0.509	0.371	0.104	0.270
	sk	**35.83**	12.59	1.243	0.567	0.183	0.070
CaLiG	*dz*	**2.139**	**22.88**	**102.8**	**52.83**	**10.79**	**77.05**
	lfm	**9.834**	**199.7**	**65.36**	**422.4**	**156.8**	**1028**
	sk	30.05	**2617**	**156588**	**504142**	**113778**	**357529**

an edge-deletion stream, indicating good scalability of CaLiG. In addition, we can also find that CaLiG exhibits a similar performance in processing edge additions and edge deletions.

Match Density. To discuss the effectiveness of the backtracking search, match density (MD) is defined as

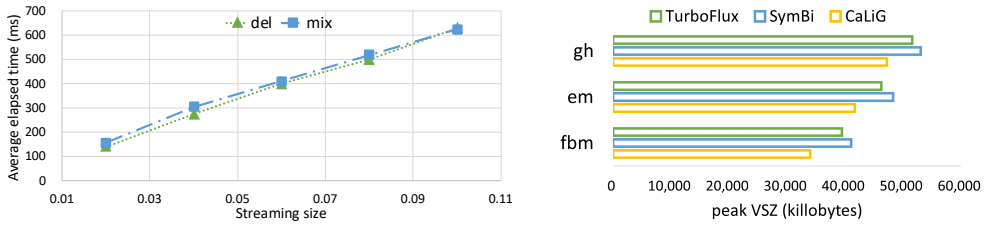
$$\text{MD} = \frac{\text{\# of Incremental Matches}}{\text{\# of Backtrackings}}$$

With higher MD, the same number of matches could be found in fewer backtrackings, and in less time. Table 5 lists the results on *dz*, *lfm*, and *sk*. The match density of CaLiG is larger than that of TurboFlux and SymBi up to 7 orders of magnitude: one backtracking in TurboFlux or SymBi fails to generate one match for most queries, while CaLiG could generate thousands of matches in one backtracking. Moreover, with the growth of query size, MDs of TurboFlux and SymBi decrease, but the MD of CaLiG increases at the same time, demonstrating CaLiG would have a more significant advance in solving large queries.

Memory usage. We evaluate the memory usage by reporting the peak memory that is defined as the maximum of the virtual set size (VSZ) in the "ps" utility output. Figure 12(b) gives the results on graphs *Facebook*, *Email*, and *Github* for query graphs with 8 vertices. CaLiG has a smaller memory usage than TurboFlux and SymBi on all the graphs.

Ablation test. We evaluate the effect of each individual technique proposed in this paper. The methods we used for comparison include

(1) *CaLiG*, the complete version of the proposed algorithm;



(a) Varying the scale of streaming data.

(b) Memory usage.

Fig. 12. Varying the scale of streaming data and Memory usage.

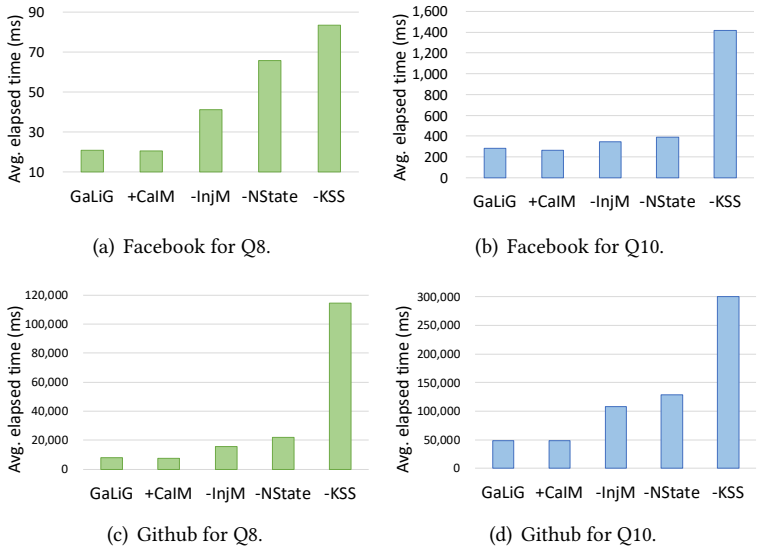
(2) *+CaIM*, caching the computed injective matching to avoid some unnecessary recomputation. If the update edge is not involved in the cached injective matching, it is not required to recompute the injective matching;
(3) *-InjM*, disabling the injective matching checking, but only considering whether each vertex in the $N_Q(u)$ has a connecting edge in the bigraph;
(4) *-NState*, disabling the lighting state update and checking;
(5) *-KSS*, replacing the KSS technique with the way of matching one expanded vertex at a time.

Figure 13 presents the results on Facebook and Github, where $Q8$ and $Q10$ represent the query graph that has 8 and 10 vertices, respectively. We can observe that the improvement is marginal by caching the injective matching as the cost of computing the injective matching is much smaller than the overall cost. In contrast, the elapsed time will significantly increase when other techniques are disabled. Among them, KSS exhibits the most superiority, achieving 4.0X~14.7X speedups on average, since incremental match generation dominates the total cost.

7 RELATED WORK

Streaming graph engines. The recent research on streaming graph engines has provided vertex or edge-centric programming models for iterative incremental graph computation. Kineograph [8] and GraphInc [7] are two systems that enable incremental computation for monotonic algorithms like Shortest Path. Chronos [18] assumes streaming graph data is stored persistently and simultaneously computes general graph algorithms on several snapshots. Auxo [17] is complementary to Chronos and could be used as its data source. GraphBolt [30] proposes a generalized incremental model to handle non-monotonic algorithms like Belief Propagation, but involves more overheads than KickStarter for monotonic algorithms. GraphS [34] designs a real-time streaming system called GraphS for cycle detection. RisGraph [12] targets per-update analysis to provide low latency and detailed information in comparison. However, the systems above do not pay attention to the problem of continuous subgraph matching.

Subgraph matching. Subgraph matching is a fundamental requirement for graph databases and has been studied extensively in recent decades. QuickSI [35] follows the direct enumeration framework, which directly explores the data graph to enumerate all results. Most state-space based representation models (that is, each state represents an intermediate result) use this framework, e.g., Ri [5], VF2++ [22], Graphql [20], CFL[4], CECI[3], and DP-ISO[16]. Due to the effective filtering and sorting methods, they have achieved great improvements in overall performance. However, these algorithms focus on the problem in a static data graph. For streaming graphs, it is unacceptable for the real-time requirement to perform subgraph matching over the graphs prior to and after updating respectively.



(a) Facebook for Q8. (b) Facebook for Q10.

(c) Github for Q8. (d) Github for Q10.

Fig. 13. Effect of different components of CaLiG.

Continuous subgraph matching. IncIsoMat [11] and Graphflow [23] expand matches from the updated edges without any additional index structures, showing limited real-time performance. SJ-Tree [10] materializes all the partial matches of subgraphs, leading to a huge storage and index update cost. TurboFlux [24] employs a concise structure DCG, a complete multigraph, to represent intermediate results, reducing the storage cost. Similarly, CEPDG [40] employs TreeMat to store the partial matches of trees. It supports changes in both pattern graphs and data graphs. However, TurboFlux and CEPDG exploit a spanning tree of the query graph to filter the candidates, leaving the non-tree edges to be maintained by DCG. SymBi [32] introduces an auxiliary data structure DCS to store weak embeddings of directed acyclic graphs as intermediate results. Compared to DCG used in TurboFlux, DCS considers both tree and non-tree edges in a DAG of the query graph. Nevertheless, the index constructed by SymBi is not cost-effective enough, that is, the candidates could be tightened to archive less total cost.

8 CONCLUSION

In this paper, we propose a cost-effective index CaLiG for continuous subgraph matching. CaLiG yields tighter candidates than previous methods with low cost, contributing to considerable speed up for match generation. To further accelerate incremental generation, we develop a novel subgraph matching paradigm, called KSS. With the partial matches of kernel vertices, KSS can produce incremental matches by just joining the candidates of shell vertices without any backtrackings. The empirical experiments demonstrate the efficiency of our proposed method.

ACKNOWLEDGMENTS

This work was supported by the Research Grants Council of Hong Kong, China (No. 14203618, No. 14202919, and No. 14205520). Weiguo Zheng is the corresponding author.

REFERENCES

[1] [n. d.]. Anonymized Internet Traces 2013. https://catalog.caida.org/details/dataset/passive_2013_pcap.

[2] Ehab Abdelhamid, Mustafa Canim, Mohammad Sadoghi, Bishwaranjan Bhattacharjee, Yuan-Chi Chang, and Panos Kalnis. 2018. Incremental Frequent Subgraph Mining on Large Evolving Graphs. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1767–1768.

[3] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1447–1462.

[4] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1199–1214. https://doi.org/10.1145/2882903.2915236

[5] 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, SUPPL7 (22 April 2013). https://doi.org/10.1186/1471-2105-14-S7-S13 Funding Information: This article is published as part of a supplement. The publication costs for this article were funded by PO grant - FESR 2007-2013 Linea di intervento 4.1.1.2, CUP G23F11000840004..

[6] Yazan Boshmaf, Ildar Muslukhov, Konstantin Beznosov, and Matei Ripeanu. 2011. The Socialbot Network: When Bots Socialize for Fame and Money. In *Proceedings of the 27th Annual Computer Security Applications Conference*. 93–102.

[7] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. 2012. Facilitating Real-Time Graph Mining. In *Proceedings of the Fourth International Workshop on Cloud Data Management*. Association for Computing Machinery, 1–8.

[8] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the Pulse of a Fast-Changing and Connected World. In *Proceedings of the 7th ACM European Conference on Computer Systems*. Association for Computing Machinery, 85–98.

[9] Sutanay Choudhury, Lawrence Holder, George Chin, Abhik Ray, Sherman Beus, and John Feo. 2013. StreamWorks: A System for Dynamic Graph Search. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1101–1104.

[10] Sutanay Choudhury, Lawrence B. Holder, George Chin Jr., Khushbu Agarwal, and John Feo. 2015. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015*. 157–168.

[11] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental Graph Pattern Matching. *ACM Trans. Database Syst.* 38, 3, Article 18 (sep 2013), 47 pages.

[12] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. *RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-Millisecond Per-Update Analysis at Millions Ops/s*. 513–527.

[13] Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman amp; Co., USA.

[14] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. 2014. Real-Time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 7, 13 (aug 2014), 1379–1380.

[15] P. Hall. 1935. On Representatives of Subsets. *Journal of the London Mathematical Society* s1-10, 1 (1935), 26–30. https://doi.org/10.1112/jlms/s1-10.37.26 arXiv:https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/jlms/s1-10.37.26

[16] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *SIGMOD 2019 - Proceedings of the 2019 International Conference on Management of Data*. 1429–1446.

[17] Wentao Han, Kaiwei Li, Shimin Chen, and Wenguang Chen. 2019. Auxo: a temporal graph management system. *Big Data Mining and Analytics* 2, 1 (2019), 58–71. https://doi.org/10.26599/BDMA.2018.9020030

[18] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Vijayan Prabhakaran, Wenguang Chen, Enhong Chen, Fan Yang, and Lidong Zhou. 2014. Chronos: A Graph Engine for Temporal Graph Analysis. In *EuroSys*.

[19] Huahai He and A.K. Singh. 2006. Closure-Tree: An Index Structure for Graph Queries. In *22nd International Conference on Data Engineering (ICDE'06)*. 38–38. https://doi.org/10.1109/ICDE.2006.37

[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 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 405–418. https://doi.org/10.1145/1376616.1376660

[21] John E. Hopcroft and Richard M. Karp. 1971. A N5/2 Algorithm for Maximum Matchings in Bipartite. In *Proceedings of the 12th Annual Symposium on Switching and Automata Theory (Swat 1971) (SWAT '71)*. IEEE Computer Society, USA, 122–125. https://doi.org/10.1109/SWAT.1971.1

[22] Alpár Jüttner and Péter Madarasi. 2018. VF2++—An improved subgraph isomorphism algorithm. *Discrete Applied Mathematics* 242 (2018), 69–81. https://doi.org/10.1016/j.dam.2018.02.018 Computational Advances in Combinatorial

Optimization.
[23] 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.
[24] 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.
[25] Seongyun Ko, Taesung Lee, Kijae Hong, Wonseok Lee, In Seo, Jiwon Seo, and Wook-Shin Han. 2021. *ITurboGraph: Scaling and Automating Incremental Graph Analytics*. Association for Computing Machinery, 977–990.
[26] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 249–263.
[27] 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 *The Semantic Web – ISWC 2012*, Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 300–312.
[28] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[29] Yuchao Li, Wei Wang, and Zishen Yang. 2019. The Connected Vertex Cover Problem in K-Regular Graphs. *J. Comb. Optim.* 38, 2 (aug 2019), 635–645. https://doi.org/10.1007/s10878-019-00403-3
[30] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*. Association for Computing Machinery, Article 25.
[31] Andrew McGregor. 2014. Graph Stream Algorithms: A Survey. *SIGMOD Rec.* (may 2014), 9–20.
[32] 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.
[33] Mohammad Hossein Namaki, Keyvan Sasani, Yinghui Wu, and Tingjian Ge. 2017. BEAMS: Bounded Event Detection in Graph Streams. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 1387–1388.
[34] 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.* (aug 2018), 1876–1888.
[35] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (dec 2017), 420–431.
[36] Dennis Shasha, Jason T. L. Wang, and Rosalba Giugno. 2002. Algorithmics and Applications of Tree and Graph Searching. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. 39–52.
[37] Chunyao Song, Tingjian Ge, Cindy Chen, and Jie Wang. 2014. Event Pattern Matching over Graph Streams. *Proc. VLDB Endow.* 8, 4 (dec 2014), 413–424.
[38] Nan Tang, Qing Chen, and Prasenjit Mitra. 2016. Graph stream summarization: From big bang to big crunch. In *SIGMOD 2016 - Proceedings of the 2016 International Conference on Management of Data*. 1481–1496.
[39] Alex Hai Wang. 2010. Don't follow me: Spam detection in Twitter. In *2010 International Conference on Security and Cryptography (SECRYPT)*. 1–10.
[40] Qianzhen Zhang, Deke Guo, Xiang Zhao, and Xi Wang. 2021. Continuous matching of evolving patterns over dynamic graph data. *World Wide Web* 24, 3 (2021), 721–745.

Received April 2022; revised July 2022; accepted August 2022