# Symmetric continuous subgraph matching with bidirectional dynamic programming

Seunghwan Min, Sung Gwan Park, Kunsoo Park, Dora Giammarresi, Giuseppe Italiano, Wook-Shin Han

▶ **To cite this version:**

Seunghwan Min, Sung Gwan Park, Kunsoo Park, Dora Giammarresi, Giuseppe Italiano, et al.. Symmetric continuous subgraph matching with bidirectional dynamic programming. VLDB 2021 - 47th International Conference on Very Large Data Bases, Aug 2021, Copenhagen, Denmark. pp.1298-1310, 10.14778/3457390.3457395 . hal-03498315

**HAL Id: hal-03498315**
**https://inria.hal.science/hal-03498315**

Submitted on 21 Dec 2021

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Symmetric Continuous Subgraph Matching with Bidirectional Dynamic Programming

Seunghwan Min
Seoul National University
shmin@theory.snu.ac.kr

Sung Gwan Park
Seoul National University
sgpark@theory.snu.ac.kr

Kunsoo Park*
Seoul National University
kpark@theory.snu.ac.kr

Dora Giammarresi†
Università Roma "Tor Vergata"
giammarr@mat.uniroma2.it

Giuseppe F. Italiano†
LUISS University
gitaliano@luiss.it

Wook-Shin Han*
Pohang University of Science and Technology (POSTECH)
wshan@dblab.postech.ac.kr

arXiv:2104.00886v1 [cs.DB] 2 Apr 2021

## ABSTRACT

In many real datasets such as social media streams and cyber data sources, graphs change over time through a graph update stream of edge insertions and deletions. Detecting critical patterns in such dynamic graphs plays an important role in various application domains such as fraud detection, cyber security, and recommendation systems for social networks. Given a dynamic data graph and a query graph, the continuous subgraph matching problem is to find all positive matches for each edge insertion and all negative matches for each edge deletion. The state-of-the-art algorithm TurboFlux uses a spanning tree of a query graph for filtering. However, using the spanning tree may have a low pruning power because it does not take into account all edges of the query graph. In this paper, we present a symmetric and much faster algorithm SymBi which maintains an auxiliary data structure based on a directed acyclic graph instead of a spanning tree, which maintains the intermediate results of bidirectional dynamic programming between the query graph and the dynamic graph. Extensive experiments with real and synthetic datasets show that SymBi outperforms the state-of-the-art algorithm by up to three orders of magnitude in terms of the elapsed time.

## 1 INTRODUCTION

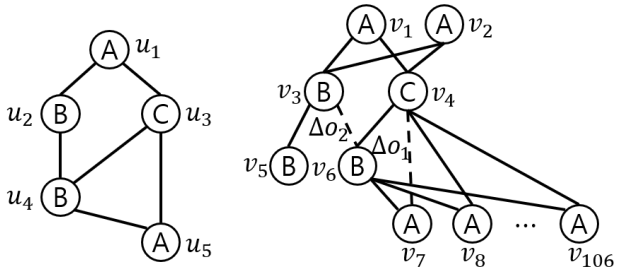
A dynamic graph is a graph that changes over time through a graph update stream of edge insertions and deletions. In the last decade, the topic of massive dynamic graphs has become popular. Social media streams and cyber data sources, such as computer network traffic and financial transaction networks, are examples of dynamic graphs. A social media stream can be modeled as a graph where vertices represent people, movies, or images, and edges represent relationship such as friendship, like, post, etc. A computer network traffic consists of vertices representing IP addresses and edges representing protocols of network traffic [17].

Extensive research has been done for the efficient analysis of dynamic graphs [1, 20, 23, 26, 36]. Among them, detecting critical patterns or events in a dynamic graph is an important issue since it lies at the core of various application domains such as fraud detection [29, 32], cyber security [6, 7], and recommendation systems for social networks [13, 18]. For example, various cyber attacks such as denial-of-service attack and data exfiltration attack can be represented as graphs [6]. Moreover, US communications company Verizon reports that 94% of the cyber security incidents fell into nine patterns, many of which can be described as graph patterns in their study, "2020 Data Breach Investigations Report" [38]. Cyber security applications should detect in real-time that such graph patterns appear in network traffic, which is one of dynamic graphs [7].

In this paper, we focus on the problem of detecting and reporting such graph patterns in a dynamic graph, called *continuous subgraph matching*. Many researchers have developed efficient solutions for continuous subgraph matching [5, 6, 10–12, 18, 19, 28] and its variants [9, 22, 25, 34, 39, 40] over the past decade. Due to the NP-hardness of continuous subgraph matching, Chen et al. [5] and Gao et al. [12] propose algorithms that cannot guarantee the exact solution for continuous subgraph matching. The results of these algorithms may include false positive matches, which is far from being desirable. Since several algorithms such as IncIsoMat [10] and Graphflow [18] do not maintain any intermediate results, these algorithms need to perform subgraph matching for each graph update even if the update does not incur any match of the pattern, which leads to significant overhead. Unlike IncIsoMat and Graphflow, SJ-Tree [6] stores all partial matches for each subgraph of the pattern to get better performance, but this method requires expensive storage space. The state-of-the-art algorithm TurboFlux [19] uses the idea of $\text{Turbo}_{\text{iso}}$ [15] which is one of state-of-the-art algorithms for the subgraph matching problem. It proposes an auxiliary data structure called *data-centric graph* (DCG), which is an updatable graph to store the partial matches for a spanning tree of the pattern graph. TurboFlux uses less storage space for the auxiliary data structure than SJ-Tree and outperforms the other algorithms. According to experimental results, however, TurboFlux has the disadvantage that processing edge deletions is much slower than edge insertions due to the asymmetric update process of DCG.

Previous studies show that what information is stored as intermediate results in an auxiliary data structure is important for solving continuous subgraph matching. An auxiliary data structure should be designed such that it doesn't take long time to update while containing enough information to help detect the pattern quickly (i.e., balancing update time vs. amount of information to keep). It was shown in [14] that the *weak embedding* of a directed acyclic

* Contact author
† Work partially done while visiting Seoul National University



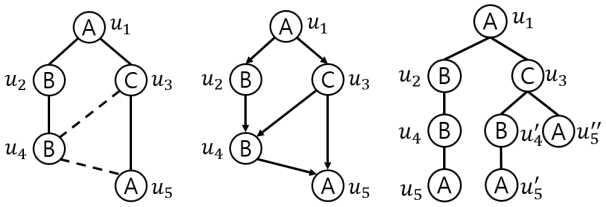

**(a) Query graph $q$** **(b) Dynamic data graph $g$ with an initial data graph $g_0$ and two edge insertions**

**Figure 1: A running example of query graph and data graph for continuous subgraph matching**

graph is more effective in filtering candidates than the embedding of a spanning tree. In this paper we embed the weak embedding into our data structure so that the intermediate results (i.e., weak embeddings of directed acyclic graphs) contain information that helps detect the pattern quickly and can be updated efficiently. We propose an algorithm SymBi for continuous subgraph matching which uses the proposed data structure. Compared to the state-of-the-art algorithm TurboFlux, this is a substantial benefit since directed acyclic graphs have better pruning power than spanning trees due to non-tree edges, while the update of intermediate results is fast. The contributions of this paper are as follows:

- We propose an auxiliary data structure called *dynamic candidate space* (DCS), which maintains the intermediate results of bidirectional (i.e., top-down and bottom-up) dynamic programming between a directed acyclic graph of the pattern graph and the dynamic graph. DCS serves as a complete search space to find all matches of the pattern graph in the dynamic graph, and it enables us to symmetrically handle edge insertions and edge deletions. Also, we propose an efficient algorithm to maintain DCS for each graph update. Rather than recomputing the entire structure, this algorithm updates only a small portion of DCS that changes.
- We introduce a new matching algorithm using DCS that works for both edge insertions and edge deletions. Unlike the subgraph matching problem, in continuous subgraph matching we need to find matches that contain the updated data graph edge. Thus, we propose a new matching order which is different from the matching orders used in existing subgraph matching algorithms. This matching order starts from an edge of the query graph corresponding to the updated data graph edge, and then selects a next query vertex to match from the neighbors of the matched vertices. When selecting the next vertex, we use an *estimate of the candidate size* of the vertex instead of the exact candidate size [14] for efficiency. In addition, we introduce the concept of *isolated vertices* which is an extension of the leaf decomposition technique from [3].

Experiments show that SymBi outperforms TurboFlux by up to three orders of magnitude. In particular, when edge deletions are included in the graph update stream, the performance gap between the two algorithms becomes larger. In an experiment where all query graphs are solved within the time limit by both algorithms,




**(a) Spanning tree $q_T$** **(b) Query DAG $\hat{q}$** **(c) Path tree of $\hat{q}$**

**Figure 2: Spanning tree, DAG, and path tree for the running example**

for example, when the ratio of the number of edge deletions to the number of edge insertions increases from 0% to 10%, the performance improvement of SymBi over TurboFlux increases from 224.61 times to 309.45 times. While the deletion ratio changes from 0% to 10%, the average elapsed time of SymBi increases only 1.54 times, but TurboFlux increases 2.13 times. This supports the fact that SymBi handles edge deletions better than TurboFlux.

The remainder of the paper is organized as follows. Section 2 formally defines the problem of continuous subgraph matching and describes some related work. Section 3 describes a brief overview of our algorithm. Section 4 introduces DCS and proposes an algorithm to maintain DCS efficiently. Section 5 presents our matching algorithm. Section 6 presents the results of our performance evaluation. Finally, we conclude in Section 7.

## 2 PRELIMINARIES

For simplicity of presentation, we focus on undirected, connected, and vertex-labeled graphs. Our algorithm can be easily extended to directed or disconnected graphs with multiple labels on vertices or edges. A *graph* $g$ is defined as $(V(g), E(g), l_g)$, where $V(g)$ is a set of vertices, $E(g)$ is a set of edges, and $l_g : V(g) \to \Sigma$ is a labeling function, where $\Sigma$ is a set of labels. Given $S \subseteq V(g)$, an *induced subgraph* $g[S]$ of $g$ is a graph whose vertex set is $S$ and whose edge set consists of all the edges in $E(g)$ that have both endpoints in $S$.

A *directed acyclic graph* (DAG) $\hat{q}$ is a directed graph that contains no cycles. A *root* (resp., *leaf*) of a DAG is a vertex with no incoming (resp., outgoing) edges. A DAG $\hat{q}$ is a *rooted DAG* if there is only one root (e.g., the rooted DAG in Figure 2b can be obtained by directing edges of $q$ in Figure 1a in such a way that $u_1$ is the root). Its reverse $\hat{q}^{-1}$ is the same as $\hat{q}$ with all of the edges reversed. We say that $u$ is a *parent* of $v$ ($v$ is a *child* of $u$) if there exists a directed edge from $u$ to $v$. An *ancestor* of a vertex $v$ is a vertex which is either a parent of $v$ or an ancestor of a parent of $v$. A *descendant* of a vertex $v$ is a vertex which is either a child of $v$ or a descendant of a child of $v$. A *sub-DAG of $\hat{q}$ rooted at $u$*, denoted by $\hat{q}_u$, is the induced subgraph of $\hat{q}$ whose vertices set consists of $u$ and all the descendants of $u$. The *height* of a rooted DAG $\hat{q}$ is the maximum distance between the root and any other vertex in $\hat{q}$, where the distance between two vertices is the number of edges in a shortest path connecting them. Let $\text{Child}(u)$, $\text{Parent}(u)$, and $\text{Nbr}(u)$ denote the children, parents, and neighbors of $u$ in $\hat{q}$, respectively.

**Definition 2.1.** Given a query graph $q = (V(q), E(q), l_q)$ and a data graph $g = (V(g), E(g), l_g)$, a *homomorphism* of $q$ in $g$ is a mapping $M : V(q) \to V(g)$ such that (1) $l_q(u) = l_g(M(u))$ for every $u \in V(q)$, and (2) $(M(u), M(u')) \in E(g)$ for every $(u, u') \in E(q)$. An

*embedding* of $q$ in $g$ is an injective (i.e., $\forall u, u' \in V(q)$ such that $u \neq u' \Rightarrow M(u) \neq M(u')$) homomorphism.

An embedding of an induced subgraph of $q$ in $g$ is called a *partial embedding* of $q$ in $g$. We say that $q$ is *subgraph-isomorphic* (resp., *subgraph-homomorphic*) to $g$, if there is an embedding (resp., homomorphism) of $q$ in $g$. We use subgraph isomorphism as our default matching semantics. Subgraph homomorphism can be easily obtained by omitting the injective constraint.

**Definition 2.2.** [14] The *path tree* of a rooted DAG $\hat{q}$ is defined as the tree $\hat{q}_T$ such that each root-to-leaf path in $\hat{q}_T$ corresponds to a distinct root-to-leaf path in $\hat{q}$, and $\hat{q}_T$ shares common prefixes of its root-to-leaf paths. Figure 2c shows the path tree of $\hat{q}$ in Figure 2b.

**Definition 2.3.** [14] For a rooted DAG $\hat{q}$ with root $u$, a *weak embedding* $M'$ of $\hat{q}$ at $v \in V(g)$ is defined as a homomorphism of the path tree of $\hat{q}$ in $g$ such that $M'(u) = v$.

**Example 2.1.** We will use the query graph and the dynamic data graph in Figure 1 and the DAG of the query graph in Figure 2b as a running example throughout this paper. For example, $\{(u_3, v_4), (u'_4, v_6), (u'_5, v_7), (u''_5, v_8)\}$ is a weak embedding of $\hat{q}_{u_3}$ (Figure 2b) at $v_4$ in $g_0$ (Figure 1b), where $\hat{q}_{u_3}$ is a sub-DAG of $\hat{q}$ rooted at $u_3$. Note that $u_5$ in $\hat{q}$ is mapped to two different vertices $v_7$ and $v_8$ of $g_0$ via the path tree. If $\Delta o_1$ is inserted to $g_0$, $\{(u_3, v_4), (u'_4, v_6), (u'_5, v_7), (u''_5, v_7)\}$ is a weak embedding (also an embedding) of $\hat{q}_{u_3}$ at $v_4$.

Every embedding of $q$ in $g$ is a weak embedding of $\hat{q}$ in $g$, but the converse is not true. Hence a weak embedding is a necessary condition for an embedding. The weak embedding is a key notion in our filtering.

**Definition 2.4.** A *graph update stream* $\Delta g$ is a sequence of update operations $(\Delta o_1, \Delta o_2, \cdots)$, where $\Delta o_i$ is a triple $(op, v, v')$ such that $v, v' \in V(g)$ and $op$ is the type of the update operation which is one of edge insertion (denoted by +) or edge deletion (denoted by −) of an edge $(v, v')$.

Update operations are defined only as inserting and deleting edges between existing vertices, but inserting new vertices or deleting existing vertices is also easy to handle. We can insert a new vertex $v$ by putting $v$ in $V(g)$ and defining a labeling function for $v$. To delete an existing vertex $v$, we first delete all edges connected to $v$ and then remove $v$ from $V(g)$ and the labeling function.

**Problem Statement.** Given an initial data graph $g_0$, a graph update stream $\Delta g$, and a query graph $q$, the *continuous subgraph matching problem* is to find all positive/negative matches for each update operation in $\Delta g$. For example, given a query graph $q$ and an initial data graph $g_0$ with two edge insertion operations $\Delta o_1 = (+, v_4, v_7)$ and $\Delta o_2 = (+, v_3, v_6)$ in Figure 1, continuous subgraph matching finds 200 positive matches when $\Delta o_2$ occurs.

## 2.1 Related Work

**Labeled Subgraph Matching.** There are many studies for practical subgraph matching algorithms for labeled graphs [2, 3, 8, 14–16, 30, 31, 33, 35, 41], which are initiated by Ullmann's backtracking algorithm [37]. Given a query graph $q$ and a data graph $g$, this algorithm finds all embeddings by mapping a query vertex to a data vertex one by one. Extensive research has been done to improve the backtracking algorithm. Recently, there are many efficient algorithms solving the subgraph matching problem, such as $\text{Turbo}_{\text{iso}}$ [15], CFL-Match [3], and DAF [14].

**Table 1: Frequently used notations**

| Symbol | Description |
|---|---|
| $g$ | Data graph |
| $\Delta g$ | Graph update stream |
| $q$ | Query graph |
| $\hat{q}$ | Query DAG |
| $C(u)$ | Candidate set for query vertex $u$ |
| $M(u)$ | Mapping of $u$ in (partial) embedding $M$ |
| $C_M(u)$ | Set of extendable candidates of $u$ regarding partial embedding $M$ |
| $\text{Nbr}_M(u)$ | Set of matched neighbors of $u$ in $q$ regarding partial embedding $M$ |

$\text{Turbo}_{\text{iso}}$ finds all the embeddings of a spanning tree $q_T$ of $q$ (e.g., solid edges in Figure 2a form a spanning tree of $q$ in Figure 1a) in the data graph. Based on the result, it extracts candidate regions from the data graph that may have embeddings of the query graph, and decides an effective matching order for each candidate region by the *path-ordering* technique. Furthermore, it uses a technique called *neighborhood equivalence class*, which compresses equivalent vertices in the query graph.

CFL-Match also uses a spanning tree for filtering to solve the subgraph matching problem, while it proposes additional techniques to improve $\text{Turbo}_{\text{iso}}$. It focuses on the fact that $\text{Turbo}_{\text{iso}}$ may check the non-tree edges of $q$ too late, and thus result in a huge search space. To handle this issue, it proposes the *core-forest-leaf decomposition* technique, which decomposes the query graph into a core including the non-tree edges, a forest adjacent to the core, and leaves adjacent to the forest. It is shown in [3] that this technique reduces the search space effectively.

DAF proposes a new approach to solve the subgraph matching problem, by building a *query DAG* instead of a spanning tree. It gives three techniques to solve the subgraph matching problem using query DAG, which are dynamic programming on DAG, adaptive matching order with DAG ordering, and pruning by failing sets. It is shown in [14] that the query DAG results in the high pruning power and better matching order. For example, DAF finds that there is no embeddings of $q$ in $g_0$ in Figure 1 without backtracking process, while $\text{Turbo}_{\text{iso}}$ and CFL-Match need backtracking.

**Continuous Subgraph Matching.** Extensive studies have been done to solve continuous subgraph matching, such as IncIsoMat [10], Graphflow [18], SJ-Tree [6], and TurboFlux [19].

IncIsoMat finds a subgraph of a data graph that is affected by a graph update operation, executes subgraph matching to it before and after a graph update operation, and computes the difference between them. The affected range within a data graph is computed based on the diameter of a query graph, where the diameter of a query graph $q$ is defined as the maximum of the length of the shortest paths between arbitrary two vertices in $q$. Since subgraph matching is an NP-hard problem, it costs a lot of time to execute subgraph matching for each graph update operation.

Graphflow uses a worst-case optimal join algorithm [24, 27]. Starting from each query edge $(u, u')$ that matches a graph edge $(v, v')$, it solves the subgraph matching starting from partial embedding $\{(u, v), (u', v')\}$ and incrementally joins the other edges

in the query graph until it gets the set of full embeddings of a query graph. Since it does not maintain any intermediate results, it starts subgraph matching from scratch every time the graph update operation occurs.

SJ-Tree decomposes a query graph $q$ into smaller graphs recursively until each graph consists of only one edge, and build a tree structure called SJ-Tree based on them, where each node in the tree corresponds to a subgraph of $q$. For each node, it stores an intermediate result for subgraph matching between a data graph and a subgraph of $q$ the node represents. When the graph update operation occurs, it updates the intermediate results starting from the leaves of SJ-Tree and recursively perform join operations between the neighbors in SJ-Tree, until it reaches the root of the tree. Since it stores all the intermediate results in an auxiliary data structure, it may cost an exponential space to the size of the query graph.

TurboFlux uses the idea of $\text{Turbo}_{\text{iso}}$, and modifies it to solve continuous subgraph matching efficiently. It maintains an auxiliary data structure called *data-centric graph*, or DCG, to maintain the intermediate results efficiently. For every pair of an edge in the data graph and an edge in a spanning tree of $q$, it stores a filtering information whether the two edges can be matched or not. For each graph update operation, it updates whether each pair of edges in DCG can be used to compose an embedding of a query graph, based on *edge transition model*. It is shown in [19] that TurboFlux is more than two orders of magnitude faster in solving continuous subgraph matching than the previous results. Note that both $\text{Turbo}_{\text{iso}}$ and TurboFlux use a spanning tree of the query graph to filter the candidates, while DAF uses a DAG built from the query graph for filtering.

## 3 OVERVIEW OF OUR ALGORITHM

Algorithm 1 shows the overview of SymBi, which takes a data graph $g$, a graph update stream $\Delta g$, and a query graph $q$ as input, and find all positive/negative matches of $q$ for each update operation in $\Delta g$. SymBi uses three main procedures below.

1. We first build a rooted DAG $\hat{q}$ from $q$. In order to build $\hat{q}$, we traverse $q$ in a BFS order and direct all edges from earlier to later visited vertices. In BuildDAG, we select a vertex as root $r$ such that the DAG has the highest height. Figure 2b shows a rooted DAG $\hat{q}$ built from query graph $q$ in Figure 1a when $u_1$ is the root.
2. BuildDCS is called to build an initial DCS structure by using bidirectional dynamic programming between the rooted DAG $\hat{q}$ and the initial data graph $g_0$ (Section 4.1).
3. For each update operation, we update the data graph $g$ and the DCS structure, and perform continuous subgraph matching. For insertion of edge $e$, we first invoke DCSChangedEdge to compute a set $E_{DCS}$ which consists of updated edges in DCS due to the inserted edge $e$. We also update the data graph by inserting the edge $e$ into $g$ and update the DCS structure with $E_{DCS}$ (Section 4.2). Finally, we find positive matches from the updated DCS and $E_{DCS}$ by calling the backtracking procedure (Section 5). For deletion of edge $e$, we find negative matches first and then update data structures because the information related to $e$ is deleted during the update.

**Algorithm 1:** Continuous Subgraph Matching

**Input:** A data graph $g$, a graph update stream $\Delta g$, and a query graph $q$
**Output:** all positive/negative matches

```
1  q̂ ← BuildDAG(q);
2  DCS ← BuildDCS(g, q̂);
3  foreach Δo ∈ Δg do
4      e ← (Δo.v, Δo.v′);
5      if Δo.op = + then
6          E_DCS ← DCSChangedEdge(g, q, e);
7          InsertEdgeToDataGraph(g, e);
8          DCSInsertionUpdate(DCS, E_DCS);
9          FindMatches(DCS, E_DCS, ∅);
10     if Δo.op = − then
11         E_DCS ← DCSChangedEdge(g, q, e);
12         FindMatches(DCS, E_DCS, ∅);
13         DeleteEdgeFromDataGraph(g, e);
14         DCSDeletionUpdate(DCS, E_DCS);
```

## 4 DCS STRUCTURE

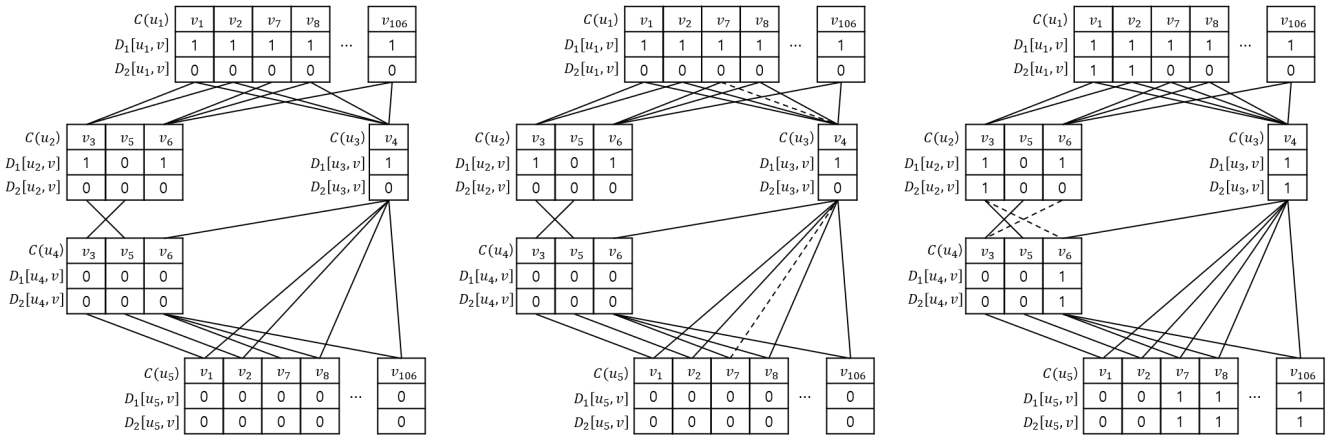
### 4.1 DCS Structure

To deal with continuous subgraph matching, we introduce an auxiliary data structure called the *dynamic candidate space* (DCS) which stores weak embeddings of DAGs as intermediate results that help reduce the search space of backtracking based on the fact that a weak embedding is a necessary condition for an embedding. These intermediate results are obtained through top-down and bottom-up dynamic programming between a DAG of a query graph and a dynamic data graph. Compared to the auxiliary data structure DCG used in TurboFlux, DCS has non-tree edge information which DCG does not have, so it is advantageous in the backtracking process. The auxiliary data structure CS (Candidate Space) which DAF [14] uses to solve the subgraph matching problem does not store intermediate results, and thus it cannot respond efficiently to the update operations.

**DCS Structure.** Given a rooted DAG $\hat{q}$ from $q$ and a data graph $g$, a *DCS* on $\hat{q}$ and $g$ consists of the following.

- For each $u \in V(q)$, a candidate set $C(u)$ is a set of vertices $v \in V(g)$ such that $l_q(u) = l_g(v)$. Let $\langle u, v\rangle$ denote $v$ in $C(u)$.
- For each $u \in V(q)$ and $v \in C(u)$, $D_1[u, v] = 1$ if there exists a weak embedding of sub-DAG $\hat{q}_u^{-1}$ at $v$; $D_1[u, v] = 0$ otherwise.
- For each $u \in V(q)$ and $v \in C(u)$, $D_2[u, v] = 1$ if there exists a weak embedding $M'$ of sub-DAG $\hat{q}_u$ at $v$ such that $D_1[u', v'] = 1$ for every mapping $(u', v') \in M'$; $D_2[u, v] = 0$ otherwise.
- There is an edge $(\langle u, v\rangle, \langle u', v'\rangle)$ between $\langle u, v\rangle$ and $\langle u', v'\rangle$ if and only if $(u, u') \in E(q)$ and $(v, v') \in E(g)$. We say that $\langle u, v\rangle$ is a parent (or child) of $\langle u', v'\rangle$ if $u$ is a parent (or child) of $u'$ in $\hat{q}$.

The DCS structure can be viewed as a labeled graph (labeled with $D_1$ and $D_2$) whose vertices are $\langle u, v\rangle$'s and edges are $(\langle u, v\rangle, \langle u', v'\rangle)$'s. Note that the intermediate results $D_1$ and $D_2$ which



**(a) Initial $DCS_0$** **(b) $DCS_1$ after $\Delta o_1 = (+, v_4, v_7)$ occurs** **(c) $DCS_2$ after $\Delta o_2 = (+, v_3, v_6)$ occurs**

**Figure 3: A running example of DCS structure on DAG $\hat{q}$ in Figure 2b and the dynamic data graph $g$ in Figure 1b**

DCS stores are weak embeddings of sub-DAGs. $D_1$ and $D_2$ store the results of top-down and bottom-up dynamic programming, respectively, which are used to filter candidates. For any embedding $M$ of $q$ in $g$, $D_2[u, v] = 1$ must hold for every $(u, v) \in M$, since a weak embedding of a sub-DAG of $q$ is a necessary condition for an embedding of $q$. From this observation, we need only consider $(u, v)$ pairs such that $D_2[u, v] = 1$ when computing an embedding of $q$ in $g$.

**Example 4.1.** Figure 3 shows the DCS structure on the DAG $\hat{q}$ in Figure 2b and the dynamic data graph $g$ in Figure 1b. Figure 3a shows the initial $DCS_0$ on $\hat{q}$ in Figure 2b and $g_0$ in Figure 1b. Figure 3b and 3c show DCS after $\Delta o_1$ and $\Delta o_2$ occur, respectively. Dashed lines $(\langle u_1, v_7\rangle, \langle u_3, v_4\rangle)$ and $(\langle u_3, v_4\rangle, \langle u_4, v_7\rangle)$ in Figure 3b represent inserted edges due to $\Delta o_1$. Note that multiple edges can be inserted to DCS by one edge insertion to the data graph. In the initial $DCS_0$ (Figure 3a), $C(u_2) = \{v_3, v_5, v_6\}$ because $v_3$, $v_5$, and $v_6$ have the same label as $u_2$, $D_1[u_2, v_3] = 1$ since there exists a weak embedding $M' = \{(u_2, v_3), (u_1, v_1)\}$ of sub-DAG $\hat{q}_{u_2}^{-1}$ at $v_3$, and $D_2[u_2, v_3] = 0$ because there is no weak embedding of sub-DAG $\hat{q}_{u_2}$ at $v_3$. Since there are no $(u, v)$ pairs such that $D_2[u, v] = 1$ in Figure 3b, SymBi reports that there are no positive matches for $\Delta o_1$ without backtracking. In contrast, TurboFlux, which uses the spanning tree in Figure 2a, needs to perform backtracking only to find that there are no positive matches for $\Delta o_1$, because there exists a spanning tree that includes the inserted edge $(v_4, v_7)$ in the data graph.

To compute $D_1$ and $D_2$, we use following recurrences which can be obtained from the definition:

$$D_1[u, v] = 1 \text{ iff } \exists v_p \in C(u_p) \text{ adjacent to } v \text{ such that } D_1[u_p, v_p] = 1 \text{ for every parent } u_p \text{ of } u \text{ in } \hat{q} \quad (1)$$

$$D_2[u, v] = 1 \text{ iff } D_1[u, v] = 1 \text{ and } \exists v_c \in C(u_c) \text{ adjacent to } v \text{ such that } D_2[u_c, v_c] = 1 \text{ for every child } u_c \text{ of } u \text{ in } \hat{q} \quad (2)$$

Based on the above recurrences, we can compute $D_1$ and $D_2$ by dynamic programming in a top-down and bottom-up fashion in DAG $\hat{q}$, respectively. Note that we reverse the parent-child relationship in the first recurrence in order to take only one DAG $\hat{q}$ into account.

**Lemma 4.1.** Recurrences (1) and (2) correctly compute $D_1$ and $D_2$ according to the definition.

**Lemma 4.2.** Given a query graph $q$ and a data graph $g$, the space complexity of the DCS structure and the time complexity of DCS construction are $O(|E(q)| \times |E(g)|)$.

## 4.2 DCS Update

In this subsection, we describe how to update the DCS structure for each update operation. An edge update in a data graph causes insertion or deletion of a set of edges in DCS, and makes changes on $D_1$ and $D_2$. Since the update algorithm works symmetrically for edge insertions and edge deletions, we describe how to update $D_1$ and $D_2$ when an edge is inserted, and then describe what changes when an edge is deleted.

We first explain DCSChangedEdge (Lines 6 and 11 in Algorithm 1) which returns a set of inserted/deleted edges in DCS due to the updated edge $e = (v, v')$. We traverse the query graph and find an edge $(u, u')$ such that $l_q(u) = l_g(v)$ and $l_q(u') = l_g(v')$. We then insert the edge $(\langle u, v\rangle, \langle u', v'\rangle)$ into the set $E_{DCS}$. In Figure 1, DCSChangedEdge returns $\{(\langle u_2, v_3\rangle, \langle u_4, v_6\rangle), (\langle u_2, v_6\rangle, \langle u_4, v_3\rangle)\}$ when $\Delta o_2 = (+, v_3, v_6)$ occurs.

**Edge Insertion.** Now, we focus on updating $D_1$ for the case of edge insertion. Obviously, it is inefficient to recompute the entire process of top-down dynamic programming to update $D_1$ for each update. Instead of computing the whole $D_1$, we want to compute only the elements of $D_1$ whose values may change. To update $D_1$, we start with an edge $(\langle u_p, v_p\rangle, \langle u, v\rangle)$ of $E_{DCS}$ where $\langle u_p, v_p\rangle$ is a parent of $\langle u, v\rangle$. If $D_1[u_p, v_p]$ is 0, then the edge $(\langle u_p, v_p\rangle, \langle u, v\rangle)$ does not affect $D_1[u, v]$ nor its descendants, so we stop the update and move to the next edge of $E_{DCS}$. On the other hand, if $D_1[u_p, v_p]$ is 1, then $D_1$ of $\langle u, v\rangle$ and its descendants may be changed due to this edge. First, we compute $D_1[u, v]$. If $D_1[u, v]$ changes from 0 to 1, then we repeat this process for the children of $\langle u, v\rangle$ until $D_1$ has no changes. Next, we try to update $D_1$ with the next edge in $E_{DCS}$.

**Example 4.2.** When $\Delta o_2$ occurs, we update $D_1$ in Figure 3b with a set $E_{DCS} = \{(\langle u_2, v_3\rangle, \langle u_4, v_6\rangle), (\langle u_2, v_6\rangle, \langle u_4, v_3\rangle)\}$. As mentioned above, we start with the edge $(\langle u_2, v_3\rangle, \langle u_4, v_6\rangle)$. Since $D_1[u_2, v_3]$ is 1, we recompute $D_1[u_4, v_6]$ and it changes from 0 to 1 because now every parent of $u_4$ has a candidate adjacent to $\langle u_4, v_6\rangle$ whose $D_1$ value is 1. Since $D_1[u_4, v_6]$ becomes 1, we iterate for the children of $\langle u_4, v_6\rangle$ (e.g., $\langle u_5, v_7\rangle, \ldots, \langle u_5, v_{106}\rangle$), and then we stop updating $D_1$ because $u_5$ has no children. Now, we update $D_1$ with the next edge $(\langle u_2, v_6\rangle, \langle u_4, v_3\rangle)$. Similarly to the previous case, we recompute $D_1[u_4, v_3]$, but it remains 0 because there are no edges between $\langle u_4, v_3\rangle$ and $\langle u_3, v_4\rangle$. So, we stop the update with $(\langle u_2, v_6\rangle, \langle u_4, v_3\rangle)$. Since $E_{DCS}$ has no more edges, we finish the update and obtain $D_1$ in Figure 3c.

We can see that there are two cases that $\langle u_p, v_p\rangle$ affects $D_1[u, v]$:

(i) If $D_1[u_p, v_p] = 1$ and an edge between $\langle u_p, v_p\rangle$ and $\langle u, v\rangle$ is inserted.
(ii) If $D_1[u_p, v_p]$ changes from 0 to 1 and there is an edge between $\langle u_p, v_p\rangle$ and $\langle u, v\rangle$.

In both of these cases, we say that $\langle u_p, v_p\rangle$ is an *updated parent* of $\langle u, v\rangle$. In Example 4.2, $\langle u_2, v_3\rangle$ is an updated parent of $\langle u_4, v_6\rangle$ from case (i) and $\langle u_4, v_6\rangle$ is an updated parent of its children from case (ii).

However, the above method has redundant computations in two aspects. First, if $\langle u, v\rangle$ has $n$ updated parents then the above method computes $D_1[u, v]$ $n$ times in the worst case. Second, to compute $D_1[u, v]$, we need to reference the non-updated parents of $\langle u, v\rangle$ even if they do not change during the update. To handle these issues, we store additional information between $\langle u, v\rangle$ and its parents. When $\langle u_p, v_p\rangle$ becomes an updated parent of $\langle u, v\rangle$, instead of computing $D_1[u, v]$ from scratch, we update the information of $\langle u, v\rangle$ related to $\langle u_p, v_p\rangle$, and then update $D_1[u, v]$ using the stored information.

We store the aforementioned information using two additional arrays, $N^1_{u,v}[u_p]$ and $N^1_P[u, v]$:

- $N^1_{u,v}[u_p]$ stores the number of candidates $v_p$ of $u_p$ such that there exists an edge $(\langle u_p, v_p\rangle, \langle u, v\rangle)$ and $D_1[u_p, v_p] = 1$, where $u_p$ is a parent of $u$. For example, $N^1_{u_2,v_3}[u_1] = 2$ in Figure 3b because $v_1$ and $v_2$ in $C(u_1)$ satisfy the condition. By definition of updated parents and $N^1_{u,v}[u_p]$, we can easily update $N^1_{u,v}[u_p]$ while updating DCS: when $\langle u_p, v_p\rangle$ becomes an updated parent of $\langle u, v\rangle$, we increase $N^1_{u,v}[u_p]$ by 1.
- $N^1_P[u, v]$ stores the number of parents $u_p$ of $u$ such that $N^1_{u,v}[u_p] \neq 0$. When $N^1_{u,v}[u_p]$ changes from 0 to 1 during the update, we increase $N^1_P[u, v]$ by 1. We can update $D_1[u, v]$ using $N^1_P[u, v]$ from the following equation obtained from Recurrence (1) in Section 4.1:

$$D_1[u, v] = 1 \text{ if and only if } N^1_P[u, v] = |\text{Parent}(u)|.$$

Back to the situation in Example 4.2, we increase $N^1_{u_4,v_6}[u_2]$ by 1 instead of recomputing $D_1[u_4, v_6]$. Since $N^1_{u_4,v_6}[u_2]$ becomes 1 from 0, we increase $N^1_P[u_4, v_6]$ from 1 to 2. Because $N^1_P[u_4, v_6] = |\text{Parent}(u_4)| = 2$ now holds, $D_1[u_4, v_6]$ becomes 1. Thus, we can update $D_1$ correctly without redundant computations.

The revised method solves the two problems described earlier. The first problem is solved in two aspects. First, the revised method still performs the update for $\langle u, v\rangle$ as many times as the number of updated parents of $\langle u, v\rangle$. However, when there is an updated parent of $\langle u, v\rangle$, we update the corresponding arrays and $D_1[u, v]$ in constant time. So, during the $D_1$ update, the total computational cost to update $D_1[u, v]$ is proportional to the number of updated parents of $\langle u, v\rangle$. Second, even if we update $D_1[u, v]$ more than once, $\langle u, v\rangle$ affects its children at most once because $\langle u, v\rangle$ affects its children only when $D_1[u, v]$ changes from 0 to 1. The second problem is solved because now we update only the information of $\langle u, v\rangle$ related to its updated parents (i.e., $N^1_{u,v}$ and $N^1_P[u, v]$) during the update process.

Similarly with the $D_1$ update, we can define *updated child*, $N^2_{u,v}$, and $N^2_C$ to update $D_2$ efficiently in a bottom-up fashion.

- $N^2_{u,v}[u']$ stores the number of candidates $v'$ of $u'$ such that there exists an edge $(\langle u', v'\rangle, \langle u, v\rangle)$ and $D_2[u', v'] = 1$, where $u'$ is a neighbor of $u$.
- $N^2_C[u, v]$ stores the number of children $u_c$ of $u$ such that $N^2_{u,v}[u_c] \neq 0$.

While we need to define $N^2_{u,v}$ only for the children of $u$ in the $D_2$ update, we define $N^2_{u,v}$ for every neighbor of $u$ in order to use it in the backtracking process (Section 5.2). The difference between the $D_1$ update and the $D_2$ update arises from the condition that $D_1[u, v]$ should be 1 in order for $D_2[u, v]$ to be 1. There is one more case where $D_2[u, v]$ changes from 0 to 1, except when it changes due to its updated children. If $D_1[u, v]$ becomes 1 and $N^2_C[u, v] = |\text{Child}(u)|$ already holds, $D_2[u, v]$ changes from 0 to 1. For example, $D_2[u_5, v_7]$ in Figure 3c changes to 1 after $D_1[u_5, v_7]$ changes to 1, and then $\langle u_5, v_7\rangle$ becomes an updated child of its parents.

Algorithm 2 shows the process of updating $D_1$, $D_2$ and the additional arrays for edge insertion. There are two queues $Q_1$ and $Q_2$ which store $\langle u, v\rangle$ such that $D_1[u, v]$ and $D_2[u, v]$ changed from 0 to 1, respectively. DCSInsertionUpdate performs the update process described above for each inserted edge $(\langle u_1, v_1\rangle, \langle u_2, v_2\rangle)$ in $E_{DCS}$. Suppose that $\langle u_1, v_1\rangle$ is a parent of $\langle u_2, v_2\rangle$. Lines 5-8 describe the update by case (i) of updated parents and updated children. It invokes the following two algorithms. InsertionTopDown (Algorithm 3) updates $N^1_{u_c,v_c}[u]$, $N^1_P[u_c, v_c]$, and $D_1[u_c, v_c]$ when $\langle u, v\rangle$ is an updated parent of $\langle u_c, v_c\rangle$. Also, when $D_1[u_c, v_c]$ becomes 1, it pushes $\langle u_c, v_c\rangle$ into $Q_1$ and check the condition $(N^2_c[u, v] = |\text{Child}(u)|)$ to see if $D_2[u_c, v_c]$ can change to 1. InsertionBottomUp (Algorithm 4) works similarly. Lines 11-14 (or Lines 15-18) perform the update process of case (ii) of updated parents (or updated children) until $Q_1$ (or $Q_2$) is not empty.

Now we show that Algorithm 2 correctly updates $D_1$ and $D_2$ for the edge insertion.

**Lemma 4.3.** If we have a correct DCS, and edges in $E_{DCS}$ are inserted into DCS by running Algorithm 2, the DCS structure is still correct after the insertion.

**Edge Deletion.** We can update DCS for edge deletion with a small modification of the previous method. The first case of the updated parent (or updated child) is changed to when an edge is deleted and the second case is changed to when $D_1[u_p, v_p]$ (or $D_2[u_p, v_p]$) changes from 1 to 0. Next, if $D_2[u, v] = 1$ and $D_1[u, v]$ becomes 0 during the $D_1$ update, then $D_2[u, v]$ also changes to 0.

**Algorithm 2:** DCSInsertionUpdate($DCS, E_{DCS}$)

```
1  Q1, Q2 ← empty queue;
2  foreach (⟨u1, v1⟩, ⟨u2, v2⟩) ∈ E_DCS do
3      if ⟨u2, v2⟩ is a parent of ⟨u1, v1⟩ then
4          swap(⟨u1, v1⟩, ⟨u2, v2⟩);
5      if D1[u1, v1] = 1 then
6          InsertionTopDown(⟨u1, v1⟩, ⟨u2, v2⟩);
7      if D2[u2, v2] = 1 then
8          InsertionBottomUp(⟨u2, v2⟩, ⟨u1, v1⟩);
9      if D2[u1, v1] = 1 then
10         N^2_{u2,v2}[u1] ← N^2_{u2,v2}[u1] + 1
11     while Q1 ≠ ∅ do
12         ⟨u, v⟩ ← Q1.pop;
13         foreach ⟨uc, vc⟩ which is a child of ⟨u, v⟩ do
14             InsertionTopDown(⟨u, v⟩, ⟨uc, vc⟩);
15     while Q2 ≠ ∅ do
16         ⟨u, v⟩ ← Q2.pop;
17         foreach ⟨up, vp⟩ which is a parent of ⟨u, v⟩ do
18             InsertionBottomUp(⟨u, v⟩, ⟨up, vp⟩);
19         foreach ⟨uc, vc⟩ which is a child of ⟨u, v⟩ do
20             N^2_{uc,vc}[u] ← N^2_{uc,vc}[u] + 1
```

**Algorithm 3:** InsertionTopDown($\langle u, v\rangle, \langle u_c, v_c\rangle$)

```
1  if N^1_{uc,vc}[u] = 0 then
2      N^1_P[uc, vc] ← N^1_P[uc, vc] + 1 ;
3      if N^1_P[uc, vc] = |Parent(uc)| then
4          D1[uc, vc] ← 1;
5          Q1.push(⟨uc, vc⟩);
6          if N^2_C[uc, vc] = |Child(uc)| then
7              D2[uc, vc] ← 1;
8              Q2.push(⟨uc, vc⟩);
9  N^1_{uc,vc}[u] ← N^1_{uc,vc}[u] + 1
```

**Algorithm 4:** InsertionBottomUp($\langle u, v\rangle, \langle u_p, v_p\rangle$)

```
1  if N^2_{up,vp}[u] = 0 then
2      N^2_C[up, vp] ← N^2_C[up, vp] + 1 ;
3      if D1[up, vp] = 1 and N^2_C[up, vp] = |Child(up)| then
4          D2[up, vp] ← 1;
5          Q2.push(⟨up, vp⟩);
6  N^2_{up,vp}[u] ← N^2_{up,vp}[u] + 1
```

**Lemma 4.4.** Let $P$ be the set of DCS vertices $\langle u, v\rangle$ such that $D_1[u, v]$ or $D_2[u, v]$ is changed during the update. Then the time complexity of the DCS update is $O(\sum_{p\in P} \deg(p) + |E_{DCS}|)$, where $\deg(p)$ is the number of edges connected to $p$. Also, the space complexity of the DCS update excluding DCS itself is $O(|E(q)|\times|V(g)|)$.

In the worst case, almost all $D_1[u, v]$ and $D_2[u, v]$ in DCS may be changed and the time complexity becomes $O(|E(q)| \times |E(g)|)$, so there is no difference from recomputing DCS from scratch. In Section 6, however, we will show that there are very few changes in $D_1[u, v]$ or $D_2[u, v]$ in practice, so our proposed update method is efficient.

# 5 BACKTRACKING

In this section, we present our matching algorithm to find all positive/negative matches in the DCS structure. Our matching algorithm works regardless of the cases of edge insertion and edge deletion.

## 5.1 Backtracking Framework

We find matches by gradually extending a partial embedding until we get an (full) embedding of $q$ in $g$. We extend a partial embedding by matching an *extendable* vertex of $q$, which is defined as below.

**Definition 5.1.** Given a partial embedding $M$, a vertex $u$ of query graph $q$ is called *extendable* if $u$ is not mapped to a vertex of $g$ in $M$ and at least one neighbor of $u$ is mapped to a vertex of $g$ in $M$.

Note that the definition of extendable vertices is different from that of DAF [14], which requires *all* parents of $u$ to be mapped to a vertex of $g$ while we require only *one* neighbor of $u$. The difference occurs because DAF has a fixed query DAG and a root vertex for backtracking, while our algorithm has to start backtracking from an arbitrary edge.

We start by mapping one edge from $E_{DCS}$, since we need only find matches including at least one updated edge in $E_{DCS}$. Until we find a full embedding, we recursively perform the following steps. First, we find all extendable vertices, and choose one vertex among them according to the *matching order*. Once we decide an extendable vertex $u$ to match, we compute its *extendable candidates*, which are the vertices in the data graph that can be matched to $u$. Formally, we define an extendable candidate as follows.

**Definition 5.2.** Given a query vertex $u$, a data vertex $v$ is its *extendable candidate* if $v$ satisfies the following conditions:

1. $D_2[u, v] = 1$ (i.e., it is not filtered in the DCS structure)
2. For all matched neighbors $u'$ of $u$, $(M(u'), v) \in E(g)$

The set of extendable candidates of $u$ is denoted by $C_M(u)$. Finally, we extend the partial embedding by matching $u$ to one of its extendable candidates and continue the process.

Algorithm 5 shows the overall backtracking process. This algorithm is invoked with $M = \emptyset$ in Algorithm 1. For each edge $(\langle u, v\rangle, \langle u', v'\rangle)$ in $E_{DCS}$, we start backtracking in Lines 6-11 only when $D_2[u, v] = D_2[u', v'] = 1$ (i.e., none of the pairs are filtered). We recursively extend a partial embedding in Lines 13-19. If we get a full embedding, we output it as a match in Line 2. UpdateEmbedding and RestoreEmbedding maintain additional values related to the matching order, every time a new match is augmented to $M$ (i.e., $M$ is updated) or an existing match is removed from $M$ (i.e., $M$ is restored). In Section 5.3, we explain what these functions do in more detail.

**Algorithm 5:** FindMatches(DCS, $E_{DCS}$, $M$)

**Input:** DCS, $E_{DCS}$, and a partial embedding $M$
**Output:** all positive/negative matches including an edge in $E_{DCS}$

```
if |M| = |V(q)| then
    Report M as a match;
else if |M| = 0 then
    foreach (⟨u, v⟩, ⟨u', v'⟩) ∈ E_DCS do
        if D_2[u, v] = 1 and D_2[u', v'] = 1 then
            M ← {(u, v), (u', v')};
            UpdateEmbedding(M, u);
            UpdateEmbedding(M, u');
            FindMatches(DCS, E_DCS, M);
            RestoreEmbedding(M, u');
            RestoreEmbedding(M, u);
else
    u ← next vertex according to the matching order;
    Compute C_M(u);
    foreach v ∈ C_M(u) do
        M' ← M ∪ {(u, v)};
        UpdateEmbedding(M, u);
        FindMatches(DCS, E_DCS, M');
        RestoreEmbedding(M, u);
```

## 5.2 Computing Extendable Candidates

According to Definition 5.2, the set of extendable candidates $C_M(u)$ of an extendable vertex $u$ is defined as follows:

$$C_M(u) = \{v : D_2[u, v] = 1, \forall u' \in \mathrm{Nbr}_M(u), (v, M(u')) \in E(g)\},$$

where $\mathrm{Nbr}_M(u)$ represents the set of matched neighbors of $u$.

We can compute the extendable candidates of $u$ based on the above equation. Even though we can compute $C_M(u)$ by naively iterating through all data vertices $v$ with $D_2[u, v] = 1$ and checking whether $(M(u'), v) \in E(g)$ for all matched neighbors $u'$ of $u$, there can be a large number of $v$'s with $D_2[u, v] = 1$, and thus it costs a lot of time to iterate through them.

Here we check the conditions in an alternative order to reduce the number of iterations. Given a vertex $u' \in \mathrm{Nbr}_M(u)$, we define a set $S_{u'} = \{v \in C(u) : D_2[u, v] = 1, (v, M(u')) \in E(g)\}$. Among the vertices $u' \in \mathrm{Nbr}_M(u)$, we find a vertex with the smallest $|S_{u'}|$ and call it $u_{min}$. We can see that the definition of $|S_{u'}|$ matches the definition of $N^2_{u',M(u')}[u]$ in Section 4.2, since the existence of an edge $(\langle u, v\rangle, \langle u', M(u')\rangle)$ is equivalent to the existence of an edge $(v, M(u'))$ if $(u, u') \in E(q)$ and $v \in C(u)$. Therefore, we have $|S_{u'}| = N^2_{u',M(u')}[u]$ and thus $u_{min}$ is the vertex $u' \in \mathrm{Nbr}_M(u)$ with smallest $N^2_{u',M(u')}[u]$.

Once we compute $u_{min}$, we compute $S_{u_{min}}$ by iterating through the neighbors $v$ of $M(u_{min})$ and checking whether $v \in C(u)$ and $D_2[u, v] = 1$. By using $S_{u_{min}}$, we rewrite $C_M(u)$ as follows:

$$C_M(u) = \{v \in S_{u_{min}} : \forall u' \in \mathrm{Nbr}_M(u) \backslash \{u_{min}\}, (v, M(u')) \in E(g)\}.$$

Based on the equation, we compute $C_M(u)$ by iterating through the vertices $v$ in $S_{u_{min}}$ and checking whether the edge $(M(u'), v)$ exists for every $u' \in \mathrm{Nbr}_M(u) \backslash \{u_{min}\}$. Note that we need only iterate through $S_{u_{min}}$, which has a considerably smaller size than the number of vertices $v$ with $D_2[u, v] = 1$ in usual. Algorithm 6 shows an algorithm to compute $C_M(u)$.

**Algorithm 6:** Computing $C_M(u)$

**Input:** DCS, a data graph $g$, a query graph $q$, and an extendable query vertex $u$
**Output:** A set of extendable candidates $C_M(u)$

```
Nbr_M(u) ← a set of matched neighbors of u in q;
u_min ← u' ∈ Nbr_M(u) with smallest N^2_{u',M(u')}[u];
S_{u_min} ← {v ∈ V(g) : D_2[u, v] = 1, (M(u_min), v) ∈ E(g)};
C_M(u) ← {v ∈ S_{u_min} : forall u' ∈ Nbr_M(u), (M(u'), v) ∈ E(g)};
```

## 5.3 Matching Order

We select a matching order that can reduce the search space (and thus the backtracking time). We choose a matching order based on the size of extendable candidates, and thus it can be adaptively changed during the backtracking process.

In the case of the subgraph matching problem, it is known that the *candidate-size order* is an efficient way [14], which chooses the extendable vertex with smallest $|C_M(u)|$. We basically follow the candidate-size order with an approximation for speed-up. Even though we can compute the exact size of extendable candidates every time we need to decide which extendable vertex to match as in DAF [14], it costs a high overhead in our case because the definition of extendable vertices is different. In the case of DAF, a vertex $u$ is extendable only when all parents of $u$ are matched. Therefore, the extendable candidates of an extendable vertex $u$ do not change. In contrast, in our algorithm, a vertex $u$ is extendable if at least one of neighbors of $u$ is matched. Therefore, $C_M(u)$ may change even if $u$ remains extendable, when an unmatched neighbor of $u$ becomes matched. As a result, our algorithm has more frequent changes in $C_M(u)$, and a higher overhead of maintaining it when compared to DAF.

To handle this issue, we use an estimated size of extendable candidates which can be maintained much faster, and we compute $C_M(u)$ only when all neighbors of $u$ are matched (i.e., when $C_M(u)$ no longer changes). Here we use the fact that in Algorithm 6, the size of $C_M(u)$ is bounded by $|S_{u_{min}}| = N^2_{u_{min},M(u_{min})}[u]$. We use this value as an estimated size of extendable candidates, since it provides an upper bound and approximation of $|C_M(u)|$, and it can be easily maintained when we update or restore partial embeddings. In a more formal way, we define $E(u)$, an *estimated size of extendable candidates* of $u$, as follows.

$$E(u) = \min_{u' \in \mathrm{Nbr}_M(u)} \{N^2_{u',M(u')}[u]\}$$

Note that for every extendable candidate $u$, $\mathrm{Nbr}_M(u)$ is not empty by definition, and thus $E(u)$ is well-defined.

Every time match $(u, v)$ occurs, we iterate through the neighbors $u'$ of $u$ in $q$, and update $E(u')$ for them. Since we have to restore a partial embedding later, we store the old $E(u')$ every time the update occurs, and we revert to the old $E(u')$ when the partial embedding is restored. These are done in UpdateEmbedding and RestoreEmbedding in Algorithm 5.

**Example 5.1.** Let's consider an edge insertion operation $\Delta o_2$ in Figure 1. As shown in Figure 3c, we get $E_{DCS} = \{(\langle u_2, v_3\rangle, \langle u_4, v_6\rangle), (\langle u_2, v_6\rangle, \langle u_4, v_3\rangle)\}$. We begin with an edge $(\langle u_2, v_3\rangle, \langle u_4, v_6\rangle)$, which results in a partial embedding $M = \{(u_2, v_3), (u_4, v_6)\}$. There are three extendable vertices at this point, which are $u_1$, $u_3$ and $u_5$. We compare the estimated sizes of extendable candidates. Since $E(u_3) = N^2_{u_4,v_6}[u_3] = 1$ is the smallest compared to $E(u_1) = 2$ and $E(u_5) = 100$, we choose $u_3$ as the next query vertex to match. We get $C_M(u_3) = \{v_4\}$ by Algorithm 6 and extend the partial embedding to $M = \{(u_2, v_3), (u_4, v_6), (u_3, v_4)\}$. Now we choose the next extendable vertex between $u_1$ and $u_5$. We again compare the estimated size of extendable candidates, and match $u_1$ first. Finally, we match $u_5$ with its extendable candidates, and output the matches. For DCS edge $(\langle u_2, v_6\rangle, \langle u_4, v_3\rangle)$, we skip finding matches since $D_2[u_2, v_6] = D_2[u_4, v_3] = 0$.

## 5.4 Isolated Vertices

In this subsection, we describe the leaf decomposition technique from [3] and introduce our new idea, called *isolated vertices*.

For a query graph $q$, its *leaf* vertices are defined as the vertices that have only one neighbor. The main idea of leaf decomposition is to postpone matching the leaf vertices of $q$ until all other vertices are matched. If we match a non-leaf vertex first, its unmatched neighbors can be the new extendable vertices (if none of their neighbors were matched before), or have their extendable candidates pruned (if at least one of their neighbors were matched before), both of which may lead to a smaller search space. These advantages do not apply when we match a leaf vertex first, since there are no unmatched neighbors if the leaf vertex is an extendable vertex.

Based on the above properties, we define *isolated vertices* as follows.

**Definition 5.3.** For a query graph $q$, a data graph $g$, and a partial embedding $M$, an *isolated vertex* is an extendable vertex in $q$, where all of its neighbors are mapped in $M$.

Note that postponing matching the isolated vertices enjoy the advantages of leaf decomposition, since isolated vertices have no unmatched neighbors and thus have the same properties as leaves in the context of leaf decomposition. Also note that every extendable leaf vertex is also an isolated vertex by definition, but the converse is not true. For example, consider a partial embedding $M = \{(u_2, v_3), (u_3, v_4), (u_4, v_6)\}$ in Figure 1. Even though there are no leaf vertices in Figure 1a, there are two isolated vertices, $u_1$ and $u_5$. Therefore, the notion of isolated vertices fully includes the leaf decomposition technique, and extends it further.

By combining the discussions in Section 5.3. and 5.4., we use the following matching order.

1. Backtrack if there exists an isolated vertex $u$ such that all data vertices in $C_M(u)$ have already matched.
2. If there exists at least one non-isolated extendable vertex in $q$, we choose the non-isolated extendable vertex $u$ with smallest $E(u)$.
3. If every extendable vertex is isolated, we choose the extendable vertex $u$ with smallest $E(u)$.

# 6 PERFORMANCE EVALUATION

In this section, we present experimental results to show the effectiveness of our algorithm SymBi. Since TurboFlux [19] outperforms the other existing algorithms (e.g., IncIsoMat [10], SJ-Tree [6], Graphflow [18]), we only compare TurboFlux and SymBi. Experiments are conducted on a machine with two Intel Xeon E5-2680 v3 2.50GHz CPUs and 256GB memory running CentOS Linux. The executable file of TurboFlux was obtained from the authors.

**Datasets.** We use two datasets referred to as LSBench and Netflow which are commonly used in previous works [6, 19]. LSBench is synthetic RDF social media stream data generated by the Linked Stream Benchmark data generator [21]. We generate three different sizes of datasets with 0.1, 0.5, and 2.5 million users with default settings of Linked Stream Benchmark data generator, and use the first dataset as a default. This dataset contains 23,317,563 triples. Netflow is a real dataset (CAIDA Internet Anonymized Traces 2013 Dataset [4]) which contains anonymized passive traffic traces obtained from CAIDA. Netflow consists of 18,520,759 triples. We split 90% of the triples of a dataset into an initial graph and 10% into a graph update stream.

**Queries.** We generate query graphs by random walk over schema graphs. To generate various types of queries, we use schema graphs instead of data graphs to randomly select edge labels regardless of edge distribution [19]. For each dataset, we set four different query sizes: 10, 15, 20, 25 (denoted by "G10", "G15", "G20", and "G25"). The size of a query is defined as the number of triples. We exclude queries that have no matches for the entire graph update stream. Also, we use the queries used in [19] (denoted by "T3", "T6", "T9", "T12", "G6", "G9", and "G12"), where "T" stands for tree and "G" stands for graph (having cycles). We generate 100 queries for each dataset and query size. One experiment consists of a dataset and a query set of 100 query graphs with a same size.

**Performance Measurement.** We measure the elapsed time of continuous subgraph matching for a dataset and a query graph for the entire update stream $\Delta g$. The preprocessing time (e.g., time to build the initial data graph and the initial auxiliary data structure) is excluded from the elapsed time. Since continuous subgraph matching is an NP-hard problem, some query graphs may not finish in a reasonable time. To address this issue, we set a 2-hours time limit for each query. If some query reaches the time limit, we record the query processing time of that query as 2 hours. We say that a query graph is *solved* if it finishes within 2 hours. To evaluate an algorithm regarding a query set, we report the average elapsed time, the number of solved query graphs, and the average peak memory usage of the program using the "ps" utility.

## 6.1 Experimental Results

The performance of SymBi was evaluated in several aspects: (1) varying the query size, (2) varying the insertion rate, (3) varying the deletion rate, and (4) varying the dataset size. Table 2 shows the parameters of the experiments. Values in boldface in Table 2 are used as default parameters. The insertion rate is defined as the ratio of the number of edge insertions to the number of edges in the original dataset before splitting. Thus, 10% insertion rate means the entire graph update stream we split. Also, the deletion rate is defined as the ratio of the number of edge deletions to the number

of edge insertions in the graph update stream. For example, if the deletion rate is 10%, one edge deletion occurs for every ten edge insertions. For an edge deletion, we randomly choose an arbitrary edge in the data graph at the time the edge deletion occurs, and delete it.

**Table 2: Experiment settings**

| Parameter | Value Used |
|---|---|
| Datasets | LSBench, Netflow |
| Query size | **G10**, G15, G20, G25, T3, T6, T9, T12, G6, G9, G12 |
| Insertion rate | 2, 4, 6, 8, **10** |
| Deletion rate | **0**, 2, 4, 6, 8, 10 |
| Dataset size | **0.1**, 0.5, 2.5 million users (LSBench) |

**Efficiency of DCS Update.** Before we compare our results with TurboFlux, we first show the efficiency of our DCS update algorithm as described in Lemma 4.4. Table 3 shows the number of updated vertices and the number of visited edges in DCS per update operation for Netflow and LSBench. "DCS $|V|$" and "DCS $|E|$" denote the average number of vertices and the average number of edges of the DCS structure. "Updated vertices" denotes the average number of $\langle u, v\rangle$'s such that $D_1[u, v]$ or $D_2[u, v]$ changes per update operation and "Visited edges" denotes the average number of DCS edges visited during the update. This result shows that the portion of DCS we need to update is extremely small compared to the size of DCS.

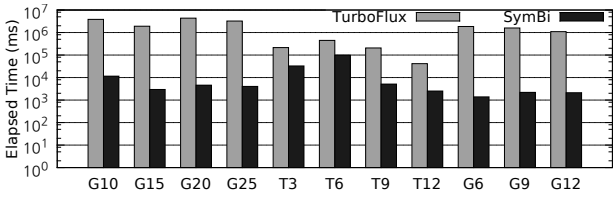
**Table 3: The number of updated vertices and visited edges in DCS per update operation (top: Netflow, bottom: LSBench)**

| Query set | DCS $\|V\|$ | DCS $\|E\|$ | Updated vertices | Visited edges |
|---|---|---|---|---|
| G10 | 30744014 | 9725255 | 0.033 | 4.634 |
| G15 | 45975850 | 16480557 | 0.052 | 9.655 |
| G20 | 58217388 | 19570587 | 0.052 | 10.177 |
| G25 | 76564119 | 25310130 | 0.024 | 12.396 |
| T3 | 12459580 | 3850193 | 11.219 | 36.284 |
| T6 | 21804265 | 7888574 | 5.382 | 13.534 |
| T9 | 31148950 | 12044573 | 2.378 | 12.951 |
| T12 | 40493635 | 16118240 | 2.596 | 15.999 |
| G6 | 18689370 | 5583268 | 0.048 | 2.168 |
| G9 | 28034055 | 8868896 | 0.072 | 4.048 |
| G12 | 37378740 | 12295665 | 0.069 | 6.093 |
| G10 | 51111071 | 6872525 | 0.203 | 1.997 |
| G15 | 75233829 | 10584646 | 0.225 | 3.006 |
| G20 | 83517886 | 10721872 | 0.141 | 2.536 |
| G25 | 125354981 | 18676312 | 0.184 | 4.573 |
| T3 | 18339548 | 2104435 | 0.104 | 0.463 |
| T6 | 32198412 | 4072171 | 0.07 | 0.773 |
| T9 | 46421983 | 5999443 | 0.065 | 1.085 |
| T12 | 60489250 | 8021626 | 0.055 | 1.335 |
| G6 | 29332857 | 2918077 | 0.042 | 0.675 |
| G9 | 42410206 | 4969639 | 0.03 | 1.001 |
| G12 | 56008565 | 6764759 | 0.029 | 1.298 |

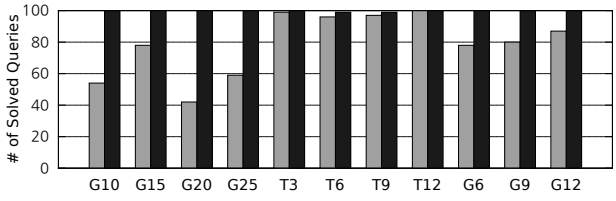
**Varying the query size.** First, we vary the number of triples in query graphs. We set the insertion rate to 10% and the deletion rate to 0% (i.e., no edge deletion in the graph update stream).

Figure 4a shows the average elapsed time for performing continuous subgraph matching for Netflow. When calculating the average elapsed time, we exclude queries that no algorithms can solve within the time limit. SymBi outperforms TurboFlux regardless of query sizes. Specifically, SymBi is 333.13 ~ 947.02 times faster than TurboFlux in our generated queries (G10 ~ G25), 4.54 ~ 16.49 times in tree queries (T3 ~ T12), and 516.48 ~ 1336.24 times in graph queries (G6 ~ G12). The performance gap between SymBi and TurboFlux is larger for graph queries than tree queries. The reason for this is that TurboFlux does not take into account non-tree edges for filtering, whereas SymBi consider all edges for filtering.

Figure 4b shows the number of queries solved by two algorithms. SymBi solves most queries for every query set (except 1 query in the T6 query set and 1 query in the T9 query set), while TurboFlux has query sets that contains many unsolved queries. Specifically, TurboFlux solves only 42 queries while SymBi solves all queries for G20.




**(a) Average elapsed time (in milliseconds)**

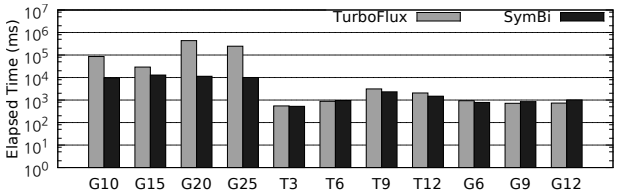



**(b) Solved queries**

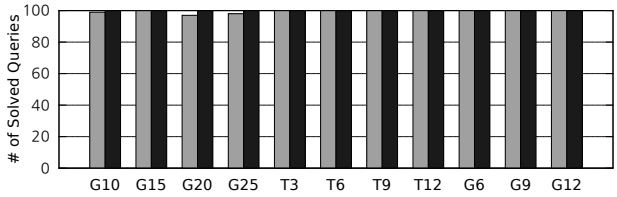
**Figure 4: Varying query size for Netflow**

Figure 5 shows the performance results for LSBench. In Figure 5a, SymBi outperforms TurboFlux by 2.26~38.35 times in our generated queries. The reason for the lesser performance gap over Netflow is that LSBench has 45 edge labels, and thus it is an easier dataset to solve than Netflow with 8 edge labels. Also, there is almost no difference in performance for the queries used in [19]. Unlike our generated queries, most queries from [19] are solved in less than one second. For theses cases, SymBi takes most of the elapsed time to update the data graph or auxiliary data structures that takes polynomial time, which is difficult to improve. In Figure 5b, SymBi solves all queries within the time limit, while TurboFlux reach the time limit for 1, 3 and 2 queries in G10, G20 and G25, respectively.

**Varying the deletion rate.** Next, we vary the deletion rate of the graph update stream. We fix the query set to G10 and the insertion rate to 10%, and vary the deletion rate from 0% to 10% in 2% increments.




(a) Average elapsed time (in milliseconds)



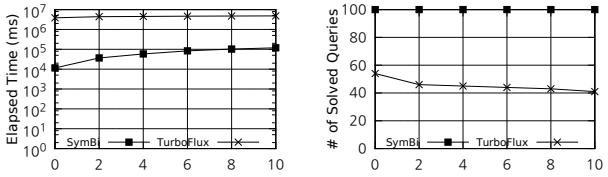

(b) Solved queries

**Figure 5: Varying query size for LSBench**

Figure 6 represents the performance results for Netflow. In Figure 6b, SymBi solves all queries for all deletion rates, but as the deletion rate increases from 0% to 10%, the number of solved queries of TurboFlux decreases from 54 to 41. Figure 6a shows that the average elapsed time of TurboFlux is almost flat, but the average elapsed time of SymBi increases as the deletion rate increases (i.e., the performance improvement of SymBi over TurboFlux is 333.13 times for 0% deletion rate and it decreases to 40.44 times as the deletion rate increases to 10%). The decrease in the performance gap stems from queries that TurboFlux cannot solve, but SymBi solves within the time limit. Figure 7 helps to understand this phenomenon. Figure 7a and 7b show the elapsed time of all queries for each algorithm with deletion rate 0% and 10%, respectively. Queries on the x-axis of Figure 7a and 7b are sorted in ascending order based on the elapsed time of TurboFlux when the deletion rate is 10%. In Figure 7a and 7b, there are many queries for which TurboFlux reaches the time limit. As the deletion rate increases from 0% (Figure 7a) to 10% (Figure 7b), the elapsed time of TurboFlux for these queries does not increase further beyond the time limit (2 hours), while the elapsed time of SymBi increases. This reduces the performance gap between two algorithms.

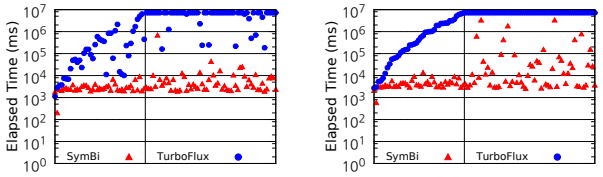
Considering this issue, we focus on 41 queries that TurboFlux solves within the time limit in all deletion rates (queries on the left side of the vertical line in Figure 7). When we measure the average elapsed time with these 41 queries, the performance ratio between two algorithms increases from 224.61 times to 309.45 times as the deletion rate increases from 0% to 10%. While the deletion rate changes from 0% to 10%, the average elapsed time of SymBi increases only 1.54 times, but the average elapsed time of TurboFlux increases 2.13 times. When two algorithms are compared, therefore, the number of solved queries as well as the average elapsed time are important measures.

The performance results for LSBench are shown in Figure 8. Figure 8b shows that SymBi solves all queries while TurboFlux solves 99 queries for all deletion rates. Figure 8a shows that as the deletion rate increases, the performance ratio between two algorithms increases (8.84 to 16.30 times). Similarly to the previous one, considering only the 99 queries that both algorithms solve,



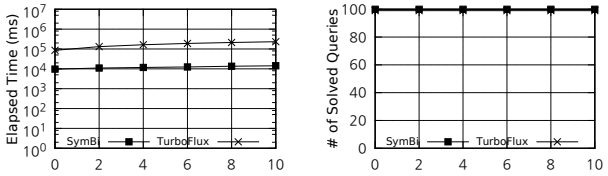

(a) Average elapsed time (in milliseconds) (b) Solved queries

**Figure 6: Varying deletion rate for Netflow**




(a) Deletion rate 0% (b) Deletion rate 10%

**Figure 7: Elapsed time of all queries for each algorithm with deletion rate 0% and 10%**




(a) Average elapsed time (in milliseconds) (b) Solved queries

**Figure 8: Varying deletion rate for LSBench**

SymBi only increases 1.52 times when the deletion rate is 10% compared to 0%, but TurboFlux increases 11.70 times. This shows that SymBi handles the edge deletion case better than TurboFlux.

In order to further analyze why SymBi processes queries better than TurboFlux as the deletion rate increases, we divide the elapsed time when the deletion rate is 10% into four types: update/backtracking time for edge insertion, and update/backtracking time for edge deletion. Since the number of insertion operations and that of deletion operations are different, we measure the elapsed time per operation by dividing the elapsed time by the number of operations. Table 4 shows the results for Netflow and LSBench. It is noteworthy that the update time of TurboFlux for edge deletion is much slower than that for edge insertion, while those of SymBi are quite similar. As noted in Section 1, this happens because the DCG update process of TurboFlux is more complex for edge deletion than for edge insertion.

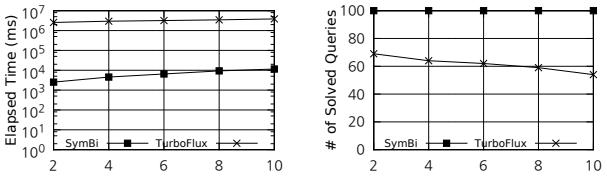
**Varying the insertion rate.** To test the effect of the insertion rate, we use the G10 query set and vary the insertion rate from 2% to 10% in 2% increments. Note that the size of the initial graph does not change from 90% of the original dataset.

Figure 9 represents the results using Netflow for varying insertion rates. Figure 9b shows that SymBi solves all queries for all insertion rates, while the number of solved queries of TurboFlux decreases from 69 to 54 as the insertion rate increases. In Figure 9a, SymBi outperforms TurboFlux regardless of the insertion rate. However, as before, the performance gap between two algorithms

**Table 4: Average update and backtracking time per operation in microseconds (top: Netflow, bottom: LSBench)**

| | TurboFlux | | SymBi | |
|---|---|---|---|---|
| | Update | Backtracking | Update | Backtracking |
| **Ins** | 6.44 | 202.48 | 0.93 | 0.41 |
| **Del** | 1867.39 | 2086.18 | 1.68 | 4.20 |
| **Ins** | 1.13 | 4.32 | 0.47 | 3.44 |
| **Del** | 599.82 | 21.72 | 0.68 | 17.32 |

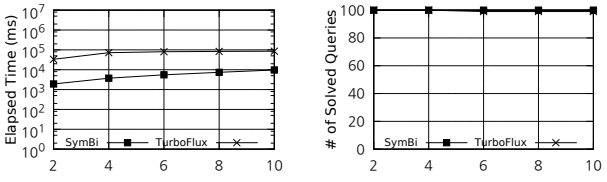
decreases as the insertion rate increases due to the queries that TurboFlux cannot solve. As in Figure 7, when we measure the average elapsed time with 54 queries that both algorithms solve within the time limit in all insertion rates, the performance ratio increases from 95.32 times to 276.60 times as the insertion rate increases from 2% to 10%.




**(a) Average elapsed time (in milliseconds)** **(b) Solved queries**

**Figure 9: Varying insertion rate for Netflow**

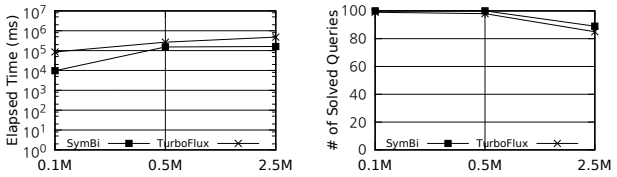
Figure 10 shows the results for LSBench. The performance ratio between two algorithms is the largest at 19.30 times when the insertion rate is 4%. As one query reaches the time limit for TurboFlux at 6% insertion rate, the performance gap starts to decrease from 6% insertion rate. Nevertheless, SymBi is 8.84 times faster than TurboFlux at 10% insertion rate.




**(a) Average elapsed time (in milliseconds)** **(b) Solved queries**

**Figure 10: Varying insertion rate for LSBench**

**Varying the dataset size.** We measure the performance for different LSBench dataset sizes: 0.1, 0.5, and 2.5 million users. The size of the initial data graph increases from 20,988,361 triples (0.1M users) to 525,446,784 triples (2.5M users). As shown in the experiment of varying the insertion rate, the number of triples in the graph update stream affects the elapsed time. To test only the effect of the dataset size, we set the same number of triples in the three graph update streams. We fix the number of triples in the graph update streams as 10% of the triples of the first dataset (0.1M users). In Figure 11, as the dataset size increases, the elapsed time of both algorithm generally increases and the number of solved queries decreases. SymBi is consistently faster and solves more queries than TurboFlux regardless of the dataset sizes.




**(a) Average elapsed time (in milliseconds)** **(b) Solved queries**

**Figure 11: Varying dataset size**

**Memory usage.** Figure 12 demonstrates the average peak memory of each program for varying the dataset size (the results for the other experiments are similar). Here, peak memory is defined as the maximum of the virtual set size (VSZ) in the "ps" utility output. This shows that SymBi uses a slightly less memory than TurboFlux regardless of the dataset sizes.

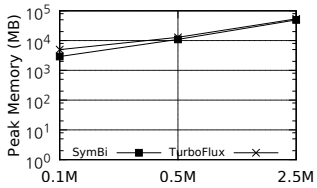



**Figure 12: Average peak memory (in MB)**

## 7 CONCLUSION

In this paper, we have studied continuous subgraph matching, and proposed an auxiliary data structure called dynamic candidate space (DCS) which stores the intermediate results of bidirectional dynamic programming between a query graph and a dynamic data graph. We further proposed an efficient algorithm to update DCS for each graph update operation. We then presented a matching algorithm that uses DCS to find all positive/negative matches. Extensive experiments on real and synthetic datasets show that SymBi outperforms the state-of-the-art algorithm by up to several orders of magnitude. Parallelizing our algorithm including both intra-query parallelism and inter-query parallelism is an interesting future work.

## 8 ACKNOWLEDGMENTS

S. Min, S. G. Park, and K. Park were supported by Institute of Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2018-0-00551, Framework of Practical Algorithms for NP-hard Graph Problems). W.-S. Han was supported by Institute of Information communications Technology Planning Evaluation(IITP) grant funded by the Korea government(MSIT) (No. 2018-0-01398, Development of a Conversational, Self-tuning DBMS). G. F. Italiano was partially supported by MIUR, the Italian Ministry for Education, University and Research, under PRIN Project AHeAD (Efficient Algorithms for HArnessing Networked Data).

## REFERENCES


[1] Ehab Abdelhamid, Mustafa Canim, Mohammad Sadoghi, Bishwaranjan Bhattacharjee, Yuan-Chi Chang, and Panos Kalnis. 2017. Incremental Frequent Subgraph Mining on Large Evolving Graphs. *IEEE Transactions on Knowledge and Data Engineering* 29, 12 (2017), 2710–2723.

[2] 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*. 1447–1462.

[3] 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*. 1199–1214.

[4] CAIDA. 2013. *The CAIDA UCSD Anonymized Internet Traces 2013.* Retrieved September 3, 2020 from https://www.caida.org/data/passive/passive_2013_dataset.xml

[5] Lei Chen and Changliang Wang. 2010. Continuous Subgraph Pattern Search over Certain and Uncertain Graph Streams. *IEEE Transactions on Knowledge and Data Engineering* 22, 8 (2010), 1093–1109.

[6] Sutanay Choudhury, Lawrence Holder, George Chin, 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*. 157–168.

[7] 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.

[8] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE transactions on pattern analysis and machine intelligence* 26, 10 (2004), 1367–1372.

[9] Grace Fan, Wenfei Fan, Yuanhao Li, Ping Lu, Chao Tian, and Jingren Zhou. 2020. Extending Graph Patterns with Conditions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 715–729.

[10] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental Graph Pattern Matching. *ACM Transactions on Database Systems (TODS)* 38, 3 (2013), 1–47.

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

[12] Jun Gao, Chang Zhou, Jiashuai Zhou, and Jeffrey Xu Yu. 2014. Continuous Pattern Detection over Billion-Edge Graph Using Distributed Framework. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 556–567.

[13] 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. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1379–1380.

[14] 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 SIGMOD*. 1429–1446. https://doi.org/10.1145/3299869.3319880

[15] 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.

[16] 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*. 405–418.

[17] Cliff Joslyn, Sutanay Choudhury, David Haglin, Bill Howe, Bill Nickless, and Bryan Olsen. 2013. Massive Scale Cyber Traffic Analysis: A Driver for Graph Database Research. In *First International Workshop on Graph Data Management Experiences and Systems*. 1–6.

[18] 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.

[19] 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.

[20] 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.

[21] 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*. Springer, 300–312.

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

[23] Andrew McGregor. 2014. Graph Stream Algorithms: A Survey. *ACM SIGMOD Record* 43, 1 (2014), 9–20.

[24] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1692–1704.

[25] Jayanta Mondal and Amol Deshpande. 2014. EAGr: Supporting Continuous Ego-centric Aggregate Queries over Large Dynamic Graphs. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of data*. 1335–1346.

[26] 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)*. IEEE, 1387–1388.

[27] Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *ACM SIGMOD Record* 42, 4 (2014), 5–16.

[28] Andrea Pugliese, Matthias Bröcheler, V. S. Subrahmanian, and Michael Ovelgönne. 2014. Efficient Multiview Maintenance under Insertion in Huge Social Networks. *ACM Transactions on the Web (TWEB)* 8, 2 (2014), 1–32.

[29] 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. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.

[30] Xuguang Ren and Junhu Wang. 2015. Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs. *Proceedings of the VLDB Endowment* 8, 5 (2015), 617–628.

[31] Carlos R Rivero and Hasan M Jamil. 2017. Efficient and scalable labeled subgraph matching using SGMatch. *Knowledge and Information Systems* 51, 1 (2017), 61–87.

[32] Gorka Sadowski and Philip Rathle. 2014. Fraud Detection: Discovering Connections with Graph Databases. *White Paper-Neo Technology-Graphs are Everywhere* 13 (2014).

[33] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proceedings of the VLDB Endowment* 1, 1 (2008), 364–375.

[34] Chunyao Song, Tingjian Ge, Cindy Chen, and Jie Wang. 2014. Event Pattern Matching over Graph Streams. *Proceedings of the VLDB Endowment* 8, 4 (2014), 413–424.

[35] 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.

[36] 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*. 1481–1496.

[37] Julian R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *Journal of the ACM (JACM)* 23, 1 (1976), 31–42.

[38] Verizon. 2020. *Data Breach Investigations Report.* Retrieved September 3, 2020 from https://enterprise.verizon.com/resources/reports/2020-data-breach-investigations-report.pdf

[39] Lefteris Zervakis, Vinay Setty, Christos Tryfonopoulos, and Katja Hose. 2019. Efficient Continuous Multi-Query Processing over Graph Streams. *arXiv preprint arXiv:1902.05134* (2019).

[40] Qianzhen Zhang, Deke Guo, Xiang Zhao, and Aibo Guo. 2019. On Continuously Matching of Evolving Graph Patterns. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 2237–2240.

[41] Peixiang Zhao and Jiawei Han. 2010. On Graph Query Optimization in Large Networks. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 340–351.


# A APPENDIX

## A.1 Proofs of Lemmas

**Proof of Lemma 4.1.** We prove the lemma for $D_1$ by induction in a top-down fashion. For each $\langle u, v\rangle$ in DCS, we divide the cases whether $u$ is the root of DAG $\hat{q}$ or not.

When $u$ is the root of DAG $\hat{q}$ (i.e., base cases), the lemma holds trivially since $v \in C(u)$ and thus $u$ and $v$ both have the same label.

When $u$ is not the root of DAG $\hat{q}$ (i.e., inductive case), let's assume that $D_1[u_p, v_p]$ is correctly computed for every parent $u_p$ of $u$ and $v_p \in C(u_p)$. Now we show that Recurrence (1) correctly computes $D_1[u, v]$, i.e., there exists a weak embedding of $\hat{q}_u^{-1}$ at $v$ if and only if $\exists v_p \in C(u_p)$ adjacent to $v$ such that $D_1[u_p, v_p] = 1$ for every parent $u_p$ of $u$ in $\hat{q}$.

First we show the 'only if' part. Let's assume that there exists a weak embedding $M'$ of $\hat{q}_u^{-1}$ at $v$. For each parent $u_p$ of $u$ in $\hat{q}$ (which is also a child of $u$ in $\hat{q}^{-1}$), we can get a weak embedding of $\hat{q}_{u_p}^{-1}$ at $M'(u_p)$ by removing the nodes not in $\hat{q}_{u_p}^{-1}$ from $M'$. Therefore, $D_1[u_p, M'(u_p)] = 1$ holds by inductive hypothesis. Furthermore,

$M'(u_p) \in C(u_p)$ and $M'(u_p)$ is adjacent to $M'(u) = v$ because $M'$ is a weak embedding of $\hat{q}_u^{-1}$ and $u_p$ is adjacent to $u$. Therefore we proved the statement.

Now we show the converse. If there exist $\exists v_p \in C(u_p)$ adjacent to $v$ such that $D_1[u_p, v_p] = 1$ for every parent $u_p$ of $u$ in $\hat{q}$, there is a weak embedding $M'_{u_p}$ of $\hat{q}_{u_p}^{-1}$ at $v_p$ for each $u_p$ by inductive hypothesis. Now we can build a weak embedding $M'$ of $\hat{q}_u^{-1}$ at $v$, by building a tree which has $v$ as a root, and $M'_{u_p}$'s as subtrees under $v$. Therefore we proved the statement.

We can similarly prove that $D_2$ is also correctly computed by Recurrence (2).

**Proof of Lemma 4.2.** We need $O(|V(q)| \times |V(g)|)$ space to store $\langle u, v\rangle$'s in the DCS structure, $O(|V(q)| \times |V(g)|)$ space to store $D_1$ and $D_2$, and $O(|E(q)| \times |E(g)|)$ space to store edges. Hence we need $O(|E(q)| \times |E(g)|)$ space for the DCS structure in total.

We can build the vertices and edges in the DCS structure in $O(|E(q)| \times |E(g)|)$ time by traversing through the vertices and edges in $q$ and $g$. Now we consider $D_1$ and $D_2$. In order to compute $D_1[u, v]$, we have to traverse through all parents $\langle u_p, v_p\rangle$ of $\langle u, v\rangle$. Since we traverse through all edges once, we need $O(|E(q)| \times |E(g)|)$ time to compute $D_1$. Similarly, we can compute $D_2$ with the same time complexity.

**Proof of Lemma 4.3.** We first consider $D_1$ here, since we can deal with the case of $D_2$ similarly to $D_1$.

In order to prove that $D_1[u, v]$ is correctly updated after the insertion for every $\langle u, v\rangle$, we need only prove that $N^1_{u,v}[u_p]$'s are correctly updated for all parents $u_p$ of $u$. It is because every time $N^1_{u,v}[u_p]$ is updated in Line 9 in Algorithm 3, we update $N^1_p[u, v]$ and $D_1[u, v]$ following their definitions in Lines 1-8 in Algorithm 3.

Now we prove that $N^1_{u,v}[u_p]$'s are correctly updated by induction on $u$, in a topological order on $\hat{q}$.

If $u$ is the root of $\hat{q}$ (i.e., base case), the statement is trivial since since there are no parents $u_p$ of the root $u$.

If $u$ is not the root of $\hat{q}$ (i.e., inductive cases), let's assume that $D_1[u_p, v_p]$'s are correctly updated for every parents of $u_p$ and $v_p \in C(u_p)$. Now we show that $N^1_{u,v}[u_p]$'s are correctly updated. If $N^1_{u,v}[u_p]$ has to be updated from 0 to 1, it means that a new edge $(\langle u, v\rangle, \langle u_p, v_p\rangle)$ is inserted to DCS for some $v_p$, or $D_1[u_p, v_p]$ was updated from 0 to 1 for some $v_p$. In the former case, $N^1_{u,v}[u']$ is updated properly in Line 6 in Algorithm 2. In the latter case, since $D_1[u_p, v_p]$ was properly updated from 0 to 1 by inductive hypothesis, Line 4 in Algorithm 3 was called with $D_1[u_p, v_p]$, and thus $\langle u, v\rangle$ is pushed to $Q_1$ in Line 5. Thus, $N^1_{u,v}[u']$ is updated properly in Line 14 of Algorithm 2, when $\langle u_p, v_p\rangle$ is popped from $Q_1$. Therefore, Algorithm 2 updates $N^1_{u,v}[u']$ every time it's necessary, and thus it has the correct value after Algorithm 2 is finished.

**Proof of Lemma 4.4.** The DCS update process for edge deletion is similar to the DCS update process for edge insertion, so we will only show the time complexity of the update process for edge insertion (Algorithm 2). First, Lines 3-10 of Algorithm 2 are executed $|E_{DCS}|$ times. Since InsertionTopDown (Algorithm 3) and InsertionBottomUp (Algorithm 4) take a constant time, the total execution time of Lines 3-10 is $O(|E_{DCS}|)$. Next, the while loop of Lines 11-14 (or Lines 15-20) takes a time proportional to the number of children of $\langle u, v\rangle$ (or the number of parent of $\langle u, v\rangle$ plus the number of children of $\langle u, v\rangle$), which is equal to or less than the number of connected edges of $\langle u, v\rangle$. Since the while loop is executed for $\langle u, v\rangle$ where $D_1[u, v]$ or $D_2[u, v]$ changes, the total execution time of Lines 11-20 is proportional to the sum of the number of edges connected to $\langle u, v\rangle$ where $D_1[u, v]$ or $D_2[u, v]$ changes. Hence, the time complexity of the DCS update is $O(\sum_{p \in P} \deg(p) + |E_{DCS}|)$.

We need $O(|E(q)| \times |V(g)|)$ space to store $N^1_{u,v}[u_p]$'s because there are $|E(q)|$ edges for $(u, u_p)$ and $|V(g)|$ vertices for $v$. Also, we need $O(|V(q)| \times |V(g)|)$ space to store $N^1_p[u, v]$'s. Similarly, we need $O(|E(q)| \times |V(g)|)$ space for $N^2_{u,v}[u']$ and $O(|V(q)| \times |V(g)|)$ for $N^2_C[u, v]$. Hence the space complexity of the DCS update excluding DCS itself is $O(|E(q)| \times |V(g)|)$.

## A.2 Extensions of Our Algorithm

In this section, we explain how to extend SymBi to handle edge-labeled graphs and directed graphs.

**Edge-labeled Graph.** To deal with edge-labeled graphs, our algorithm needs to be modified as follows. First, when constructing DCS, edge labels should be considered. Specifically, for the edge $(\langle u, v\rangle, \langle u', v'\rangle)$ in DCS to exist, not only edges $(u, u')$ and $(v, v')$ must exist, but the edge labels of both must also be the same. Also, when computing $D_1[u, v]$ (or $D_2[u, v]$) through recurrences, it is necessary to verify that the label of $(u, u_p)$ (or $(u, u_c)$) and the label of $(v, v_p)$ (or $(v, v_c)$) should be the same. Next, when computing $E_{DCS}$ in DCSChangedEdge, only edges of the query graph with the same edge label as $(v, v')$ should be included. Finally, when computing $S_{u_{min}}$ in Algorithm 6, we must include vertex $v$ such that the label of $(u, u_{min})$ and the label of $(v, M(u_{min}))$ are the same. Similarly, the edge label must be considered when computing $C_M(u)$.

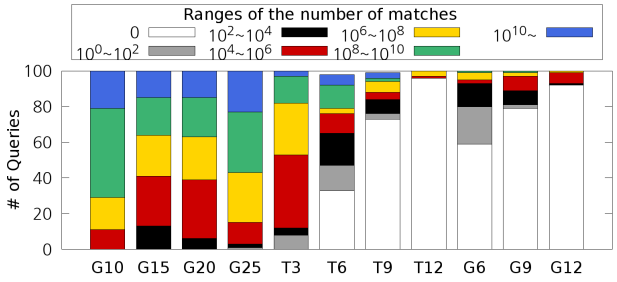
**Directed Graph.** Suppose that we are given a directed data graph and a directed query graph. To create a parent-child relationship of query vertices required in DCS, we regard the directed query graph as an undirected graph and build a rooted DAG as in the paper. The query DAG created in this way is independent of the actual direction of the edge of the query graph. We use the query DAG to order the query vertices when constructing or updating DCS in a top-down or bottom-up fashion. When we map an edge $(u, u')$ of the query graph and an edge $(v, v')$ of the data graph, we need to check that the directions of the two edges match, as in the edge-labeled graph.

## A.3 Distribution of the Number of Matches

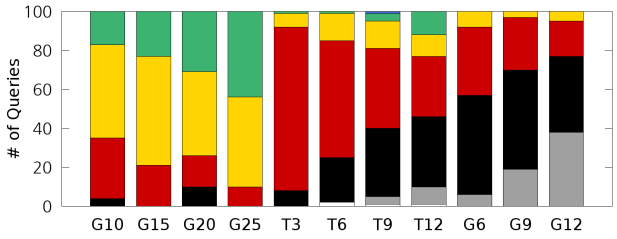
Stacked bar charts in Figure 13 represent the distribution of the number of matches for queries we test. The bar for each query set is made up of seven sub-bars. The color of a sub-bar represents the range of the number of matches as shown in the legend of Figure 13, and the size of a sub-bar represents the number of queries for which the number of matches is in that range. Sub-bars are stacked in order from the smallest number of matches to the largest number of matches from the bottom. For example, the bar for the G10 query set shows that there are 11 queries with $10^4 \sim 10^6$ matches, 18 queries with $10^6 \sim 10^8$ matches, 50 queries with $10^8 \sim 10^{10}$ matches, and 21 queries with more than $10^{10}$ matches. In Figure 13a, two queries in the T6 query set bar and one in the T9 query set bar are missing because neither algorithm can solve them within the time limit. Also, the stacked bars in Figure 13 show different results from

the stacked bars of [19], since [19] uses graph homomorphism as matching semantics while we use graph isomorphism as matching semantics.

Figure 13 shows that our generated queries have a larger number of matches than queries from [19]. Also, Figure 13 shows that the Netflow queries have more matches than the LSBench queries. This supports the fact that the LSBench queries are easier to solve than the Netflow queries as mentioned above.




**(a) Netflow query sets**




**(b) LSBench query sets**

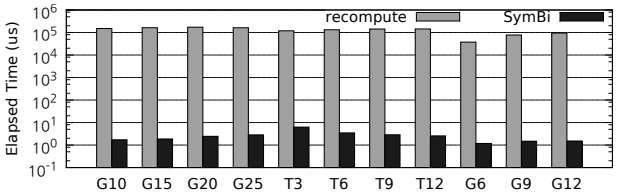
**Figure 13: Distribution of the number of matches for queries**

## A.4 Effect of Our Techniques

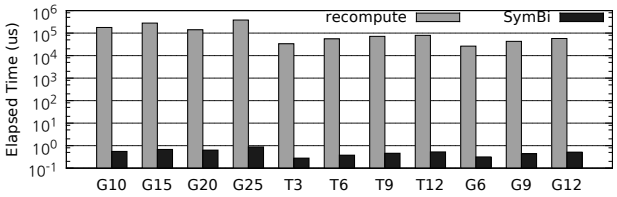
**Effect of DCS Update.** To see the effect of the proposed DCS update method, we compare the elapsed time of the proposed method with the elapsed time of recomputing from scratch. We also measure the number of updated vertices and the number of visited edges in DCS during the DCS update, which are presented in Section 6.1. Since recomputing from scratch takes a long time, we limit the number of update operations to 1000.

Figure 14 shows the average DCS update time per update operation (i.e., average DCS update time / 1000) for Netflow and LSBench. Our proposed update method is faster than recomputing from scratch by up to four orders of magnitude for Netflow and five orders of magnitude for LSBench.

**Effect of Estimated Size.** To evaluate the effect of the estimated size of extendable candidates, we compare the estimated size of extendable candidates and the exact size of extendable candidates through two scores. Also, we compare the elapsed time when using the estimated size and the elapsed time when using the exact size.




**(a) Netflow query sets**




**(b) LSBench query sets**

**Figure 14: Average elapsed time per update operation (in microseconds)**

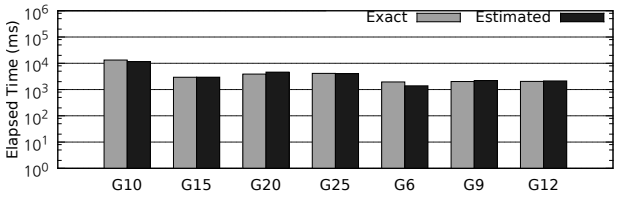
To define the two scores $S_1$ and $S_2$ for a query graph and a dataset, we consider the query vertex $u$ selected to match according to the estimated candidate size order for each partial embedding $M$ in the search process. To show how similar the estimated size and the exact size are, we define the first score $S_1$ as the sum of $|C_M(u)|$ of the vertices that we consider divided by the sum of $E(u)$ of the same vertices. Next, the second score $S_2$ is defined as the number of partial embeddings in which the estimated candidate size order and the exact candidate size order select the same query vertex $u$ (i.e., both $E(u)$ and $|C_M(u)|$ are the smallest among extendable vertices) divided by the number of partial embeddings in the search process.

Table 5 shows the average scores of 100 queries in a query set for Netflow and LSBench. We exclude tree-shaped query sets because extendable vertices in a tree-shaped query have only one matched neighbor. The average $S_1$ score for the 7 query sets is 0.434 for Netflow and 0.874 for LSBench. The average $S_2$ score is 0.830 for Netflow and 0.966 for LSBench. The $S_1$ score indicates that depending on the dataset, there may be differences between the estimated size and the exact size. However, the $S_2$ score shows that in most cases of both datasets, the estimated candidate size order chooses the same vertex as the exact candidate size order. Although the computational overhead of the estimated size of extendable candidates is negligible, the estimated candidate size order works almost like the exact candidate size order.

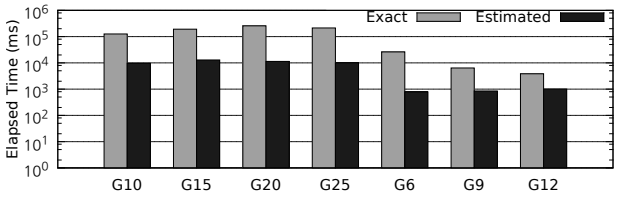
**Table 5: Estimated size $E(u)$ vs. exact size $|C_M(u)|$. $S_1$: $\sum |C_M(u)| / \sum E(u)$, $S_2$: proportion that the estimated candidate size order is the same as the exact candidate size order. (top: Netflow, bottom: LSBench)**

| Score | G10 | G15 | G20 | G25 | G6 | G9 | G12 |
|---|---|---|---|---|---|---|---|
| $S_1$ | 0.567 | 0.535 | 0.445 | 0.509 | 0.308 | 0.332 | 0.341 |
| $S_2$ | 0.869 | 0.831 | 0.811 | 0.863 | 0.772 | 0.842 | 0.822 |
| $S_1$ | 0.939 | 0.950 | 0.891 | 0.955 | 0.754 | 0.807 | 0.823 |
| $S_2$ | 0.998 | 0.991 | 0.899 | 0.975 | 0.969 | 0.973 | 0.959 |

Figure 15 represents the average elapsed time when using the estimated size and the exact size. In Figure 15b, the algorithm using the estimated size outperforms the algorithm using the exact size for all query sets in LSBench. In particular, the algorithm using the exact size failed to solve 1 and 2 queries in G10 and G20, respectively, while the algorithm using the estimated size solves all queries. This is because the estimated size in LSBench works almost the same as the exact size, but there is no overhead for updating $C_M(u)$.



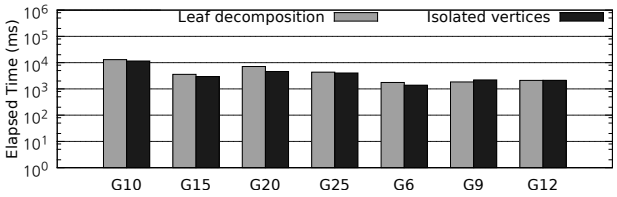

**(a) Average elapsed time (in milliseconds) for Netflow query sets**



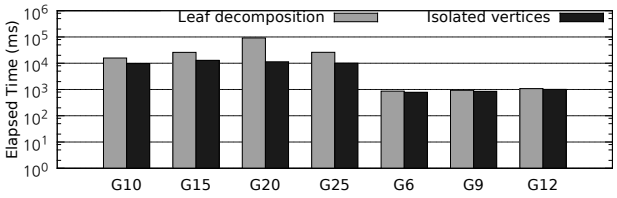

**(b) Average elapsed time (in milliseconds) for LSBench query sets**

**Figure 15: Estimated size vs. exact size**

**Effect of Isolated Vertex.** To test the effect of isolated vertices, we compare the elapsed time when using the isolated vertex technique and the elapsed time when not using it (i.e., just using the leaf decomposition technique in [3]). Figure 16 shows the results. Since the isolated vertex technique and the leaf decomposition technique are the same in a tree-shape query, we exclude tree-shaped query sets. Due to the characteristics of the datasets, the generated queries are sparse. Because of this, there are not many situations where a query vertex becomes an isolated vertex without being a leaf, so the performance of the two techniques is similar in many query sets. Nevertheless, when using the isolated vertex technique, it is 1.63, 2.01, 8.07, and 2.59 times faster on query sets G10, G15, G20, and G25 of LSBench, respectively.




**(a) Average elapsed time (in milliseconds) for Netflow query sets**




**(b) Average elapsed time (in milliseconds) for LSBench query sets**

**Figure 16: Isolated vertex vs. leaf decomposition**