World Wide Web (2021) 24:721–745
https://doi.org/10.1007/s11280-020-00860-5

Continuous matching of evolving patterns over dynamic graph data

Qianzhen Zhang[1] · Deke Guo[1] · Xiang Zhao[1] · Xi Wang[1]

Received: 8 September 2020 / Revised: 5 November 2020 / Accepted: 21 December 2020 /
Published online: 13 April 2021

Abstract

Nowadays, the scale of various graphs soars rapidly, which imposes a serious challenge to develop processing and analytic algorithms. Among them, graph pattern matching is the one of the most primitive tasks that find a wide spectrum of applications, the performance of which is yet often affected by the size and dynamicity of graphs. In order to handle large dynamic graphs, incremental pattern matching is proposed to avoid re-computing matches of patterns over the entire data graph, hence reducing the matching time and improving the overall execution performance. Due to the complexity of the problem, little work has been reported so far to solve the problem, and most of them only solve the graph pattern matching problem under the scenario of the data graph varying alone. In this article, we are devoted to a more complicated but very practical graph pattern matching problem, *continuous matching of evolving patterns over dynamic graph data*, and the investigation presents a novel algorithm CEPDG for continuously pattern matching along with changes of both pattern graph and data graph. Specifically, we propose a concise representation TreeMat of partial matching solutions, which can help to avoid re-computing matches of the pattern and speed up subsequent matching process. In order to enable the updates of data graph and pattern graph, we propose an incremental maintenance strategy, to efficiently maintain the intermediate results. Moreover, we conceive an effective model for estimating step-wise cost of pattern evaluation to drive the matching process. Extensive experiments verify the superiority of CEPDG.

Keywords Dynamic graph · Subgraph matching · Incremental algorithm

This article is an extension of our earlier published work [16] in ACM CIKM 2019.

✉ Deke Guo
dekeguo@nudt.edu.cn

✉ Xiang Zhao
xiangzhao@nudt.edu.cn

Qianzhen Zhang
850806464@qq.com

1 The Science and Technology on Information Systems Engineering Laboratory, National University of Defense Technology, Changsha, 410073, China

1 Introduction

In recent years, graph analysis plays an increasingly important role in the area of data analytics [14, 15]. Graph pattern matching is one of the most fundamental problems in graph analytics. Given a pattern graph P and a large data graph G, graph pattern matching is to find all subgraph isomorphic of P in G, which has a wide range of applications such as fraud detection and cyber security.

However, graphs are dynamic in nature [11], which continuously evolve over the time. A dynamic graph is defined by an initial graph and a graph update stream of edge insertions and edge deletions. Identifying and monitoring critical patterns in a dynamic graph is important in various application domains [6] such as fraud detection, cyber security, and emergency response, etc. For example, cyber security applications should detect cyber intrusions and attacks in computer network traffic as soon as they appear in the data graph[3]. Most of the previous works only solve the subgraph matching problem under the scenario of the data graph varying alone. But it is common that pattern graph will also evolve along with the time when data graph is updated. For example, in cyberthreats surveillance, one could predict upcoming malicious activities and determine the ultimate goal of an adversary by concealing and supplementing selective edges of attacking patterns, respectively [16].

The aforementioned two update scenarios motivate us to investigate a new problem, *continuous matching of evolving patterns over dynamic graph data*. Formally, given an initial data graph G_0, an initial pattern graph P_0, a graph update stream $(\Delta g_1, \Delta g_2, \Delta p_3, \Delta p_4, \cdots)$ consisting of edge insertions and deletions of the data graph and pattern graph, $G_i = G_{i-1} \oplus \Delta g_i$ (resp. $P_i = P_{i-1} \oplus \Delta p_i$), and $M(P, G)$ denotes the set of subgraph matching results between G and P. Here, $\oplus$ means that Δg_i (resp. Δp_i) is applied to G_{i-1} (resp. P_{i-1}). Then the *continuous matching of evolving patterns over dynamic graph data* problem is to report $M(P_{i-1} \oplus \Delta p_i, G_{i-1})$ (resp. $M(P_{i-1}, G_{i-1} \oplus \Delta g_i)$) when each update operation Δp_i (resp. Δg_i) occurs. A naïve method to solve this problem is to repetitively execute pattern matching for each update to the data graph and pattern graph. Nonetheless, this can be prohibitively costly due to the extensive involvement of expensive subgraph isomorphism tests [8].

To address the challenge, efforts to support incremental graph pattern matching for dynamic data graph seemed to enjoy some success. In [5], INCISOMAT extracts the subgraph of data graph that can be affected by each update operation and conducts subgraph matching for the extracted subgraph to get the new matches by performing the set difference. GraphFlow [9] applies a worst-case optimal join algorithm called *Generic Join* to incrementally evaluate subgraph matching for each update. SJ-TREE [2] uses a left-deep tree, where an internal node in SJ-TREE corresponds to a subgraph containing more than two connected query vertices, and a leaf node corresponds to a subgraph containing two adjacent query vertices. TurboFlux is the state-of-the-art algorithm for continuous subgraph matching [10], which employs a data-centric indicate representation of intermediate results, namely, DCG, in the sense that the query pattern P is embedded into the data graph G. TurboFlux can obtain a higher performance than above algorithms. However, it only considers the update operations of data graph and is no longer applicable on both update scenarios; to put it in our context, TurboFlux has to re-compute DCG when the updates occur on the pattern graph, which can be detrimental.

These problem of existing methods motivated us to develop a fully-fledged framework, namely, CEPDG, to achieve fast pattern matching under the variations of both data graph and pattern graph. To the best of our knowledge, this is among the first attempts to conduct pattern matching under the situation of data graph and pattern graph varying simultaneously. In summary, we make the following contributions:

- We introduce a concise representation TreeMat of partial solutions, which can help to avoid executing subgraph pattern matching repeatedly for edge updates on the data graph and pattern graph;
- In order to enable frequent updates on the data graph, we propose a vertex state transition strategy, to efficiently maintain the intermediate results.
- We devise an execution model to efficiently and incrementally maintain the representation during edge updates on the pattern graph, which are compatible with the algorithm proposed for data graph very well.
- We conceive an effective cost model for estimating step-wise cost of pattern matching.

Comprehensive empirical study verifies the efficiency of the proposed algorithm and techniques.

Organization Section 2 formulates the problem, and presents the overview of the proposed framework. Section 3 introduces a novel representation of intermediate results called the TreeMat and proposes the incremental maintenance strategy. Section 4 explains the algorithms of CEPDG in detail. Experimental results and analyses are reported in Section 5. A brief overview of related work follows immediately in Section 6. Section 7 concludes the paper.

2 Preliminaries and framework

In this section, we first introduce several essential notions and formalize the continuous matching of evolving patterns over dynamic graph data problem. Then, we overview the proposed solution.

2.1 Preliminaries

We focus on a labeled undirected graph $g = (V, E, L)$. Here, V is the set of vertices, $E \in V \times V$ is the set of edges, and L is a labeling function that assigns a label l to each $v \in V$. Each vertex has only one label, representing the attribute of the node. Note that, our techniques can be readily extended to handle directed graphs.

Definition 1 (Graph update stream) A graph update stream Δo is a sequence of update operations $(\Delta o_1, \Delta o_2, \cdots)$, where Δo_i is a triple $\langle op, v_i, v_j \rangle$ such that $op = \{I, D\}$ is the type of operations, with I and D representing edge insertion and deletion of an edge $\langle v_i, v_j \rangle$.

A dynamic graph abstracts an initial graph g and an update stream Δo. g transforms to g' after applying Δo to g. Here, g represents a data graph or pattern graph. Note that, insertion of a vertex can be represented by a set of edge insertions, similarly, deletion of a vertex can be considered as a set of edge deletions.

Definition 2 (Subgraph isomorphism) Given a pattern graph $P = (V_P, E_P, L_P)$, a data graph $G = (U_G, E_G, L_G)$, P is isomorphism to G if there is a bijective function between them, such that: (1) $\forall v \in V_P$, $L_P(v) = L_G(f(v))$; and (2) $\forall (v_i, v_j) \in E_P$, $(f(v_i), f(v_j)) \in E_G$, where $f(v)$ is the vertex in G to which v is mapped.

Definition 3 (Problem statement) Given a pattern graph $P = (V_P, E_P, L_P)$, a data graph $G = (U_G, E_G, L_G)$, and a graph update stream Δo, the continuous matching of evolving patterns over dynamic graph data problem is to continuously return occurrences of P in G when the updates in Δo occur on the pattern graph P or data graph G.

Frequently used notations are summarized in Table 1.

2.2 Overview of solution

In this subsection, we overview the proposed solution, which is referred as CEPDG(Continuous matching of Evolving Patterns over Dynamic Graph data). Specially, we are to address two technical challenges:

- Update operation needs to be efficient such that the intermediate results can be maintained incrementally.
- Pattern matching needs to be efficient such that the number of intermediate results is minimized.

The former corresponds to *update handling* phase, while the latter challenge corresponds to the *query evaluation* phase.

Algorithm 1 shows the outline of CEPDG, which takes an initial pattern graph P_0, an initial data graph G_0 and a graph update stream Δo as input, and find the matching results of P in G when necessary. We first select a root vertex v_r (Line 1). Then we extract from the pattern graph P_0 a *structural tree* P_T based on v_r, walking a spanning tree by breadth-first search, and removing non-tree edges from P_0.(Line 2). The purpose is to execute fast query evaluation by leveraging tree structure [8], i.e., we handle the edges in the query tree first, and then, the non-tree edges.

In particular, to perform continuous subgraph matching, we construct an auxiliary data structure, namely, TreeMat, based on P_T to store the matching results of the structural tree,

Table 1 Notations

Notations	Description
P and G	Pattern graph and data graph
V_P / E_P	The vertex set / the edge set of P
U_G / E_G	The vertex set / the edge set of G
u / v	A vertex in G / a vertex in P
Δo	The graph update stream
Δp / Δg	Updates for P / updates for G
P_T	A generated spanning tree of P
v_r	The root vertex of P_T
match(v)	The set of vertices $\{u\}$ that map to v in some embedding to P_T
cand(v)	The candidates of v
$N_v^{v_p}(u)$	The set of vertices $\{u'\}$ in cand(v) such that $\langle u', u\rangle$ matches $\langle v_p, v\rangle$
$N(v)$ / $deg(v)$	The set of visited neighbors of v in P_T / the total degree of v
M_i	The set of embedding for the subgraph of P induced by $(v_1, ..., v_i)$
r_i	The non-tree edges that connect v_i
d_i^j	The vertices in match(v_i) joinable with an embedding in M_{i-1}

which is able to provide guidance to generate answers with light computation overhead (Line 3). During a graph update stream, when an update comes, we amend the auxiliary data structure first, and then calculate the matching results if necessary (Lines 4–13). For example, on update o of data graph, we first match o to corresponding edges in P_T, and then incrementally maintain the intermediate results in TreeMat (Lines 6–9). On update o of pattern graph, we incrementally maintain TreeMat directly (Lines 10–12). After that, we call subgraphSearch to obtain the matching results if output requested (Line 13). The design and rationale for auxiliary data structure maintenance is given, as well as the algorithm details are given in the subsequent sections, respectively.

Algorithm 1 CEPDG.

```
Input : $P_0$ is the initial pattern graph; $G_0$ is the initial data graph;
        $\Delta o = (\Delta o_1, \Delta o_2, \cdots)$ is the graph update stream.
1  $v_r \leftarrow$ chooseRootVertex($P_0$);
2  $P_T \leftarrow$ constructTree($P_0$, $v_r$);
3  TreeMat $\leftarrow$ constructTreeMat($G_0$, $P_T$);
4  while $\Delta o$ is not empty do
5      $o \leftarrow \Delta o$.pop();
6      if $o$ is an update on $G_0$ then
7          foreach edge $e$ of $P_T$ that matches $o$ do
8              if $o$ is an insertion then G-insertEval($o$, TreeMat) ;
9              else G-deleteEval($o$, TreeMat) ;
10     if $o$ is an update on $P_0$ then
11         if $o$ is an insertion then P-insertEval($o$, TreeMat) ;
12         else P-deleteEval($o$, TreeMat) ;
13     if output requested then subgraphSearch(TreeMat) ;
```

Root Vertex Selection Intuitively, we favor the root vertex to have a small number of candidates and to have a large degree; fewer candidates means fewer partial embeddings being generated, while larger degree means more chance to prune partial embeddings at early stages. In order to minimize the number of matching data vertices for root vertex v_r, chooseRootVertex first selects a pattern edge $\langle v, v' \rangle$ which has the smallest number of matching data edges. Between v and v', chooseRootVertex chooses a pattern vertex that has a smaller number of matching data vertices. Finally, if there is a tie, chooseRootVertex chooses a pattern vertex having a larger degree.

3 Incremental maintenance of intermediate results

The central idea of update handling is to employ a delicate data structure to store and incrementally maintain partial solutions.

3.1 A concise representation

There has been a long tradition in graph community to harness a tree structure for fast pattern matching/search [1, 8]. We also follow this tradition, and conceive a succinct data structure for keeping partial solutions. P_T is constructed by removing the edges that are not in the spanning tree, i.e., *non-tree edges*, if P contains cycles. The vertices in P are partitioned according to their levels in the spanning tree where the level of a vertex in P_T is its depth compared to the root vertex of P_T.

To keep partial solutions, we offer a concise representation named TreeMat, which comprises matching vertices to those of P_T in topology graph G. Given a vertex v in P_T, its matching vertices in TreeMat are arranged into

– $\mathsf{match}(\cdot)$: the set of vertices $\{u\}$ in G that map to v in some solutions to P_T; and
– $\mathsf{stree}(\cdot)$: the set of vertices $\{u\}$ in G such that 1) the subtree residing at v matches the corresponding subtree at u via *subgraph homomorphism* [10], and 2) there does not exist a solution to P_T that map v to u.

Here, subgraph homomorphism can be obtained by just removing the injectivity constraint. It can be seen that the two sets are *mutually exclusive*, and we use a general designation *candidates* of v i.e., $\mathsf{cand}(v)$ to refer the vertices in either $\mathsf{match}(v)$ or $\mathsf{stree}(v)$. As a consequence, the structure of TreeMat is defined as follows.

– It is a tree-like structure, and for each query vertex v in P_T, there is a *node* containing the candidates of v, which is constituted of two sets $\mathsf{match}(v)$ and $\mathsf{stree}(v)$; and
– there is an edge between $u \in \mathsf{cand}(v)$ and $u' \in \mathsf{cand}(v')$ for adjacent query vertices v and v' in TreeMat, if and only if edge $\langle u, u' \rangle \in G$.

It is noted that $\mathsf{stree}(v_r)$ of the root vertex v_r in P_T is empty, since P_T is also a subtree residing at v_r.

Example 1 Figure 4b shows the TreeMat for P_T (Figure 4a) and initial data graph G_0. Given a vertex v in T, the orange square in $\mathsf{cand}(v)$ represents a data vertex $u \in \mathsf{stree}(v)$; and the black square in $\mathsf{cand}(v)$ represents a data vertex $u \in \mathsf{match}(v)$. Furthermore, we can see that the root vertex v_1 of P_T only has the set $\mathsf{match}(\cdot)$.

Remark As pointed out in [10], existing work on continuous subgraph matching caches either a set of partial solutions or a set of candidate vertices for each query vertex. These paradigms incur not only great memory overhead but also large computational cost. In contrast, our model takes a more eager strategy, and proposes to keep complete solutions (in $\mathsf{match}(\cdot)$) as well as solution-likely-to-be's (in $\mathsf{stree}(\cdot)$). In this way, we save TreeMat from filling up the main memory while offering guidance to efficiently derive affected answers.

3.2 Data graph change-oriented rationale of maintenance

In this subsection, we propose a vertex state transition strategy (denoted as VST) to efficiently maintain the intermediate results.

When an edge update operation $\langle u, u' \rangle$ arrives, we try to match it with an edge $\langle v, v' \rangle$ in P_T. Here, the level of v is deemed to be smaller than the level of v'. Then, we use VST to maintain the TreeMat. We set the data vertex $u \in \mathsf{NULL}$ if $u \notin \mathsf{cand}(v)$. Figure 1 shows the state transition diagram, consisting of three states and six transition rules (Transitions 1–6), which demonstrates how one state is transited to another. Here, Transition 1–3 are triggered by edge insertion, and Transition 4–6 are triggered by edge deletion.

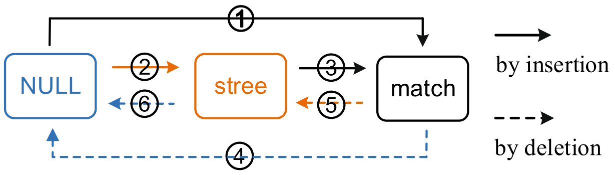


Figure 1 Conceptual Model of Maintenance

3.2.1 Handling edge insertion

Consider an edge $\langle u, u' \rangle$ inserted into G_0, to which $\langle v, v' \rangle$ is matched in P_T. Let v be the parent vertex of v'.

① **From NULL to match.** Suppose that $u \in \mathsf{match}(v)$ and $u' \in \mathsf{NULL}$. If v' is a leaf vertex, then we add u' into $\mathsf{match}(v')$.

Suppose that v is the root vertex in P_T, $u' \in \mathsf{cand}(v')$ and $u \in \mathsf{NULL}$. For each child vertex v_c of v except v', if v_c is a leaf vertex, we check if there is an edge $\langle u, u_c \rangle$ matching $\langle v, v_c \rangle$; else we further check if $u_c \in \mathsf{cand}(v_c)$. If so, we add vertex u into $\mathsf{match}(v)$. In specific, if v_c is a leaf vertex and $u_c \in \mathsf{NULL}$, we should also add vertex u_c into $\mathsf{match}(v_c)$.

② **From NULL to stree.** Suppose that $u \in \mathsf{NULL}$ and $u' \in \mathsf{cand}(v')$. Here, v is not the root vertex in P_T. For each child vertex v_c of v except v', if v_c is a leaf vertex, we check if there is an edge $\langle u, u_c \rangle$ matching $\langle v, v_c \rangle$; else we further check if $u_c \in \mathsf{cand}(v_c)$. If so, we add vertex u into $\mathsf{stree}(v)$. In specific, if v_c is a leaf vertex and $u_c \in \mathsf{NULL}$, we should also add vertex u_c into $\mathsf{stree}(v_c)$.

Suppose that the data vertex u is added into $\mathsf{stree}(v)$. For each $u_p \in \mathsf{NULL}$ that is adjacent to u, if $\langle u, u_p \rangle$ matches $\langle v, v_p \rangle$ where v_p is the parent vertex v'' of v, we further check whether u_p can be added into $\mathsf{stree}(v_p)$ with a similar manner (Fig. 2).

③ **From stree to match.** Suppose that $u' \in \mathsf{stree}(v')$ and $u \in \mathsf{match}(v)$. Then we remove u' from $\mathsf{stree}(v')$ to $\mathsf{match}(v')$.

Suppose that the data vertex u is added into $\mathsf{match}(v)$. For each child vertex v_c of v, if there is a vertex u_c in $\mathsf{stree}(v_c)$ that is adjacent to u in TreeMat, then we remove u_c from $\mathsf{stree}(v_c)$ to $\mathsf{match}(v_c)$.

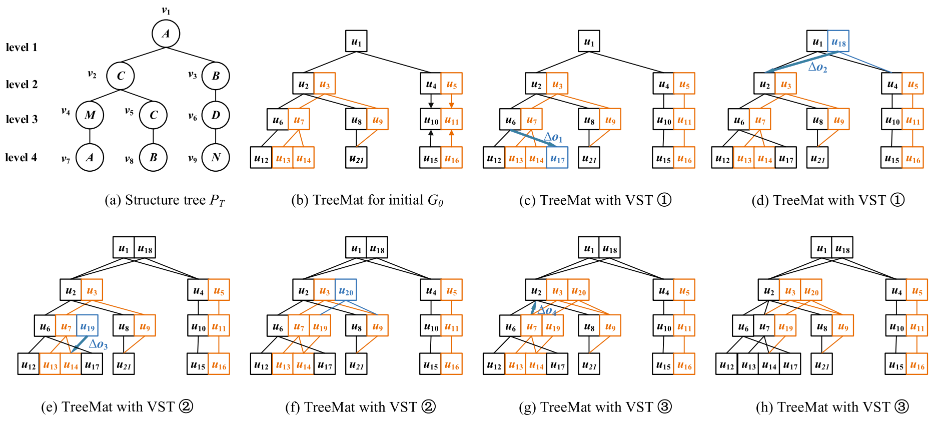


Figure 2 Example of edge insertions of on the data graph

Example 2 Figure 2c–h give the examples of vertex state transition strategy for edge insertion, where Figure 2c–d show the strategy ①, Figure 2e–f show the strategy ②, and Figure 2g–h show the strategy ③. In Figure 2c, the edge insertion Δo_1 matches $\langle v_4, v_7 \rangle$ where $u_6 \in \mathsf{match}(v_4)$. Since v_7 is a leaf vertex in P_T, we add u_{17} to $\mathsf{match}(v_7)$. In Figure 2d, the edge insertion Δo_2 matches $\langle v_1, v_2 \rangle$ where $u_2 \in \mathsf{match}(v_2)$. Since v_1 is the root vertex in P_T and $\langle u_{18}, u_4 \rangle$ matches $\langle v_1, v_3 \rangle$ with $u_4 \in \mathsf{match}(v_3)$, we add u_{18} into $\mathsf{match}(v_1)$. In Figure 2e, the edge insertion Δo_3 matches $\langle v_4, v_7 \rangle$ where $u_{14} \in \mathsf{stree}(v_7)$. Since v_4 has no child vertex exclude v_7, we add u_{19} into $\mathsf{stree}(v_4)$. In Figure 2f, there is a neighbor u_{20} of u_{19} that satisfies $\langle u_{19}, u_{20} \rangle$ matches $\langle v_4, v_2 \rangle$. Since $\langle u_{20}, u_9 \rangle$ matches $\langle v_2, v_5 \rangle$, we further add u_{20} into $\mathsf{stree}(v_2)$. In Figure 2g, the edge insertion Δo_4 matches $\langle v_4, v_2 \rangle$ where $u_2 \in \mathsf{match}(v_2)$ and $u_7 \in \mathsf{stree}(v_4)$. We then remove u_7 from $\mathsf{stree}(v_4)$ to $\mathsf{match}(v_4)$. In Figure 2h, we further check the data vertices in $\mathsf{stree}(v_7)$ where v_7 is the child vertex of v_4. Since u_{13} and u_{14} are the neighbors of u_7 in $\mathsf{stree}(v_7)$, we remove u_{13} and u_{14} from $\mathsf{stree}(v_7)$ to $\mathsf{match}(v_7)$.

3.2.2 Handling edge deletion

Consider an edge $\langle u, u' \rangle$ deleted from G_0, to which $\langle v, v' \rangle$ is matched in P_T. Let v be the parent vertex of v'.

④ **From match to NULL.** Suppose that $u \in \mathsf{match}(v)$ and $u' \in \mathsf{match}(v')$. If there is no data vertex in $\mathsf{match}(v')$ that is adjacent to u except u', we delete u from $\mathsf{match}(v)$. In specific, if v' is a leaf vertex, and there is no other data vertex in $\mathsf{cand}(v)$ that is adjacent to u', we delete u' from $\mathsf{match}(v')$.

Suppose that u is deleted from $\mathsf{match}(v)$. For each neighbor u_p of u in $\mathsf{match}(v_p)$ where v_p is the parent of v, if there is no other data vertex in $\mathsf{match}(v)$ that is adjacent to u_p, then we delete u_p from $\mathsf{match}(v_p)$.

⑤ **From match to stree.** Suppose that $u \in \mathsf{match}(v)$ and $u' \in \mathsf{match}(v')$. If there is no other data vertex in $\mathsf{match}(v)$ that is adjacent to u', then we remove u' from $\mathsf{match}(v')$ to $\mathsf{stree}(v')$. In specific, if v' is a leaf vertex, we need further check if there is a vertex in $\mathsf{stree}(v)$ that is adjacent to u'; if so, remove u' from $\mathsf{match}(v')$ to $\mathsf{stree}(v')$.

⑥ **From stree to NULL.** Suppose that $u \in \mathsf{stree}(v)$ and $u' \in \mathsf{cand}(v')$. If there is no other data vertex in $\mathsf{cand}(v')$ that is adjacent to u, we then delete u from $\mathsf{stree}(v)$. In specific, if v' is a leaf vertex in P_T and $u' \in \mathsf{stree}(v')$, we need further check whether there is a data vertex in $\mathsf{stree}(v)$ that is adjacent to u'. If not, we delete u' from $\mathsf{stree}(v')$.

Suppose that the vertex u is deleted from $\mathsf{stree}(v)$. For each neighbor u_p of u in $\mathsf{stree}(v_p)$ where v_p is the parent of v, if there is no other data vertex in $\mathsf{cand}(v)$ that is adjacent to u_p, then we delete u_p from $\mathsf{stree}(v_p)$.

3.3 Pattern graph change-oriented rationale of maintenance

It can be seen that if inserted (or deleted) edge is a non-tree edge, we do not update TreeMat, since it has no impact on TreeMat. Thus, the following exposition concentrates on tree edges.

Handling edge insertion Consider a tree edge $\langle v, v' \rangle$ inserted into P_T, where v' is the vertex newly introduced. Under this scenario, candidate vertices are only to be excluded from $\mathsf{match}(\cdot)$ or $\mathsf{stree}(\cdot)$, back to NULL state, but not vice versa. To identify affected candidates, we check, for each vertex u in $\mathsf{match}(v)$, whether there is an edge $\langle u, u' \rangle$ with $u' \in$ NULL matching $\langle v, v' \rangle$. If not, we delete u from $\mathsf{match}(v)$; otherwise, we add vertex u' into $\mathsf{match}(v')$ if $u \in \mathsf{match}(v)$. $\mathsf{stree}(v)$ or $\mathsf{stree}(v')$ can be updated in a similar fashion.

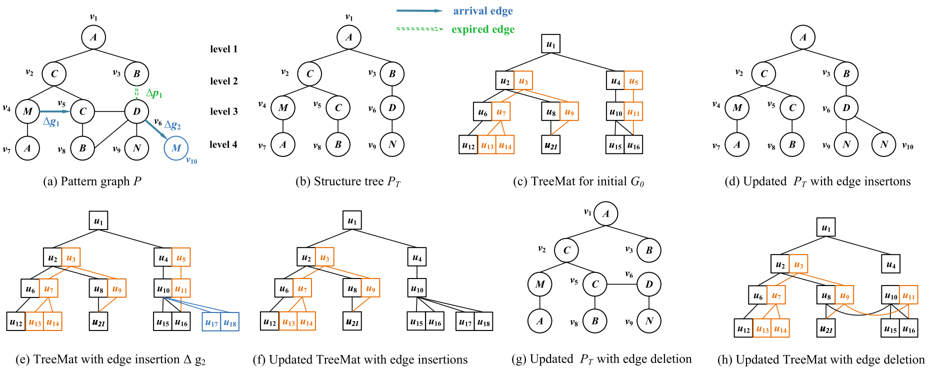

arrival edge
expired edge
level 1
level 2
level 3
level 4
(a) Pattern graph P
(b) Structure tree P_T
(c) TreeMat for initial G_0
(d) Updated P_T with edge insertons
(e) TreeMat with edge insertion Δ g2
(f) Updated TreeMat with edge insertions
(g) Updated P_T with edge deletion
(h) Updated TreeMat with edge deletion

Figure 3 Example of edge updates on the pattern graph

Moreover, when vertex u is excluded from the candidates of v, such update needs to be propagated upwards in TreeMat till the root vertex. Consider the parent vertex v_p of v, if u_p is the neighbor of u in $\mathsf{match}(v_p)$, and there is no vertex in $\mathsf{match}(v)$ that is adjacent to u_p in TreeMat, we exclude u_p from $\mathsf{match}(v_p)$.

Handling edge deletion We discuss edge deletion in two cases based on whether the deletion involves a leaf vertex of P_T.

Case 1 Consider tree edge $\langle v, v'\rangle$ with v' as a leaf vertex. Note that in this case, NULL vertices only are to be included into $\mathsf{match}(\cdot)$ or $\mathsf{stree}(\cdot)$, but not vice versa. Intuitively, a vertex u of G_0 is added into $\mathsf{stree}(v)$, only if for each child vertex v_c of v exclude v', there is a vertex u_c that is candidate to v_c such that $\langle u, u_c\rangle$ matches $\langle v, v_c\rangle$.

Then, update needs to be propagated upwards to the root of TreeMat. Suppose that vertex u is added into $\mathsf{stree}(v)$. For each vertex u_p that is adjacent to u and $\langle u_p, u\rangle$ matches $\langle v_p, v\rangle$, if $u_p \in$ NULL, we check whether u_p can be added into $\mathsf{stree}(v_p)$ in a similar manner; else if $u_p \in \mathsf{match}(v_p)$, we move u from $\mathsf{stree}(v)$ to $\mathsf{match}(v)$. In the other situation when vertex u is added into $\mathsf{match}(v)$, we examine, for each child vertex v_c of v, whether there is vertex u_c in $\mathsf{stree}(v_c)$ that is adjacent to u in TreeMat; if so, remove data vertex u_c to $\mathsf{match}(v_c)$.

Case 2 Consider a tree edge $\langle v, v'\rangle$ not involving any leaf vertex. This type of edge deletion will break the connectivity of P_T but not P[1]. Thus, a non-tree edge that connects v' with an arbitrary vertex will become a tree edge. By intuition, we choose, among all the non-tree edges, that one v'' that connects v' to a vertex closer to the root and has smaller $\mathsf{match}(\cdot)$ set.

Then, for each vertex $u'' \in \mathsf{stree}(v'')$, we check whether there is a candidate u' of v' such that $\langle u'', u'\rangle$ matches $\langle v'', v'\rangle$; if not, we exclude u'' from $\mathsf{stree}(v'')$, and further check the vertices in $\mathsf{stree}(v_p)$, where v_p is the parent of v''. The update is propagated upwards till the root.

Example 3 Figure 3d–h give the examples of updating process for edge insertions and deletion of the pattern graph. In Figure 3d, since $\langle v_4, v_5\rangle$ is a non-tree edge, we only add edge $\langle v_6, v_{10}\rangle$ into P_T. In Figure 3e, since there is no vertex u' that is adjacent to

[1]A pattern graph seldom loses connectivity for threats surveillance.

u_{11} such that $\langle u_{11}, u' \rangle$ matches $\langle v_6, v_{10} \rangle$, we remove u_{11} from $\mathsf{stree}(v_6)$. Accordingly, we remove the parent vertex u_5 of u_{11} from $\mathsf{stree}(v_3)$. What's more, since $u_{10} \in \mathsf{match}(v_6)$, and there are two vertices u_{17} and u_{18} that are adjacent to u_{10} such that edges $\langle u_{10}, u_{17} \rangle$ and $\langle u_{10}, u_{18} \rangle$ match $\langle v_6, v_{10} \rangle$, we add u_{17} and u_{18} into $\mathsf{match}(v_{10})$. Figure 3f gives the updated TreeMat with edge insertion Δg_2. When the edge Δp_1 is deleted from P, there are two non-tree edges $\langle v_5, v_6 \rangle$ and $\langle v_6, v_8 \rangle$ that can be translated into tree edges. Here, we translate $\langle v_5, v_6 \rangle$ into tree edge, since $|\mathsf{match}(v_5)| = |\mathsf{match}(v_8)|$ and v_5 is closer to the root vertex v_1. The updated P_T and TreeMat are given in Figures 3g and h, respectively.

Algorithm 2 constructTreeMat.

```
Input  : $P_T$ is the spanning tree of a pattern graph $P$;  $G_0$ is the initial data graph.
Output: TreeMat is the auxiliary data structure.
1  Mark $\{v_l\}$ as visited; Set $V(u) \leftarrow 0$ for all $u$ in $G_0$;
2  foreach level $lev$ from $maxlevel$ to 1 in bottom-up fashion do
3      foreach unvisited vertex $v$ at level $lev$ do
4          $N(v) \leftarrow 0$;
5          foreach visited vertex $v'$ that is adjacent to $v$ do
6              if $v'$ is a non-leaf vertex then
7                  foreach vertex $u' \in \mathsf{cand}(v')$ do
8                      foreach vertex $u$ that is adjacent to $u'$ do
9                          if $V(u) = N(v)$ and $\langle u, u' \rangle$ matches $\langle v, v' \rangle$ then
10                             $V(u) \leftarrow V(u) + 1$;
11             else
12                 Same as Lines 9–10;
13             $N(v) \leftarrow N(v) + 1$;
14         foreach vertex $u$ in $G_0$ with $V(u) = N(v)$ do
15             $\mathsf{cand}(v) \leftarrow \mathsf{cand}(v) \cup \{u\}$;
16         if $v_c$ is a child of $v$ and $v_c$ is a leaf vertex then
17             foreach vertex $u \in \mathsf{cand}(v)$ do
18                 if there exists an edge $\langle u, u' \rangle$ matching $\langle v, v' \rangle$ then
                       $\mathsf{cand}(v') \leftarrow \mathsf{cand}(v') \cup \{u'\}$ ;
19         constructAdj($v$, $v_p$);
20         Mark $v$ as visited; set $V(u) = 0$ s.t. $V(u) > 0$;
21 $\mathsf{match}(v_r) \leftarrow \mathsf{match}(v_r) \cup \{\mathsf{cand}(v_r)\}$; $\mathsf{stree}(v_r) \leftarrow \emptyset$;
22 foreach level $lev$ from 2 to $maxlevel$ do
23     foreach vertex $v$ at level $lev$ do
24         foreach vertex $u \in \mathsf{cand}(v)$ do
25             if there is a vertex in $\mathsf{match}(v_p)$ that is adjacent to $u$ then
                   $\mathsf{match}(v) \leftarrow \mathsf{match}(v) \cup \{u\}$ ;
26             else $\mathsf{stree}(v) \leftarrow \mathsf{stree}(v) \cup \{u\}$ ;
27 return TreeMat;
```

4 CEPDG algorithms

In this section, we present detailed algorithms for CEPDG. We develop efficient techniques for constructing TreeMat. While we update the TreeMat, we need only apply necessary transition rules. This motivated us to develop an enhanced version of the maintenance algorithm for the TreeMat. Then we conceive an effective cost model for estimating the step-wise cost of query pattern matching.

4.1 TreeMat construction

To construct TreeMat, constructTreeMat (Line 3 of Algorithm 1) (1) first generates $\mathsf{cand}(v)$ (candidates of v) for each query vertex v in P_T; (2) then constructs the adjacent lists corresponding to query vertices and their parent vertices; and (3) finally divides the $\mathsf{cand}(v)$ into $\mathsf{stree}(v)$ and $\mathsf{match}(v)$.

In the forward processing, we mark all the leaf vertices of P_T as visited and then process the query vertices level-by-level in a bottom-up fashion (Lines 1–20). In processing an unvisited vertex v, let $N(v)$ denotes the set of visited neighbors of v in P_T (Line 13). Intuitively, a data vertex u is in $\mathsf{cand}(v)$ only if for each $v' \in N(v)$, there is a data vertex $u' \in \mathsf{cand}(v')$ such that $\langle u, u' \rangle$ matches $\langle v, v' \rangle$. In specific, in above process, if v' is a leaf vertex, we need only verify whether there is a data vertex u' such that $\langle u, u' \rangle$ matches $\langle v, v' \rangle$. To achieve this, we maintain a counter $V(u)$ for each data vertex in G_0 to count the number of visited query neighbors of v that have a candidate u' adjacent to u such that $\langle u, u' \rangle$ matches $\langle v, v' \rangle$. $V(u)$ is updated at Lines 8–10. The candidate $\mathsf{cand}(v)$ is the set of vertices satisfying $N(v) = V(u)$ (Lines 14–15). After generating $\mathsf{cand}(v)$, we will further generate $\mathsf{cand}(v')$ if v' is a leaf vertex. That is, u' is added to v' if there is a data vertex $u \in \mathsf{match}(v)$ such that $\langle u, u' \rangle$ matches $\langle v, v' \rangle$ (Lines 16–18).

At the same time, we construct the adjacency lists corresponding to vertex v and its parent vertex v_p in P_T (Line 19). The adjacency lists corresponding to an edge $\langle v_p, v \rangle$ is constructed. That is, for each data vertex $u \in \mathsf{cand}(v_p)$, an adjacency list $N_v^{v_p}(u)$ is constructed, which is the set of data vertices $\{u'\}$ in $\mathsf{cand}(v)$ such that $\langle u', u \rangle$ matches $\langle v_p, v \rangle$. Then, we mark v as visited, reset $V(u)$ to be 0 for every vertex u that has a positive count (Line 18).

In the backward processing, we reprocess the query vertices of P_T in a top–down manner to divide $\mathsf{cand}(v)$ into $\mathsf{match}(v)$ and $\mathsf{stree}(v)$ for each query vertex v. Firstly, we set $\mathsf{match}(v_r) = \mathsf{cand}(v_r)$ for the root vertex v_r, since T_{EQ} is also a subtree residing at v_r. Then, we process vertices downwards according to their levels. In processing a query vertex v, let v_p denote the parent vertex of v. For each data vertex u in $\mathsf{cand}(v)$, we check if there is a data vertex u_p in $\mathsf{match}(v_p)$ that is adjacent to u. If so, we move u to $\mathsf{match}(v)$; otherwise we move u to $\mathsf{stree}(v)$ (Lines 24–26).

Lemma 1 *The worst storage complexity of TreeMat is* $O(|E_{G_0}| \times |V_{P_T}|)$.

Proof The TreeMat stores at most $|E_{G_0}|$ edges for each pattern vertex in P_T and thus, its worst storage complexity is $O(|E_{G_0}| \times |V_{P_T}|)$. □

Lemma 2 *The worst time complexity of constructTreeMat is* $O(|E_{G_0}| \times |E_{P_T}|)$.

Proof In the worst case, constructTreeMat is called for every query vertex v and every data vertex u. We show that in the forward process for a special v take time $O(|E_{G_0}| \times |N(v)|)$.

In particular, for each data vertex $u' \in \mathsf{cand}(v')$, it takes $O(deg(u'))$ time to check whether $\langle u, u'\rangle$ matches $\langle v, v'\rangle$ where $deg(u')$ is the degree of u'; thus, for all vertices in $\mathsf{cand}(v')$, the checking processes take $O(\sum_{u' \in \mathsf{cand}(v')} deg(u')) = O(|E_{G_0}|)$. Similarly, in the backward process for a special v takes time $O(|E_{G_0}|)$ time. Thus, the total time for a special v is $O(|E_{G_0}| \times (|N(v)| + 1)) = O(|E_{G_0}| \times deg(v))$ where $deg(v)$ is the degree of v in P_T, and the total running time of constructTreeMat is $O(\sum_{v \in P_T} |E_{G_0}| \times deg(v)) = O(|E_{G_0}| \times |E(P_T)|)$. $\square$

Algorithm 3 G-insertEval.

```
Input  : TreeMat: the auxiliary data structure;
         $\langle u, u'\rangle$: incoming edge to be inserted.
1 foreach edge $\langle v, v'\rangle \in P_T$ that matches $\langle u, u'\rangle$ do
2     if $v$ is the parent of $v'$ and $u' \in \mathsf{cand}(v')$ then
3         $E \leftarrow E \cup \{\langle v, v'\rangle\}$
4 foreach $\langle v, v'\rangle \in E$ do
5     chooseVST($\langle u, u'\rangle$);
6     if TreeMat.getTransition($\langle u, u'\rangle$)=true then
7         if $u \in \mathsf{match}(v)$ then updateTreeMat($u', v'$) ;
8         else updateTreeMat($u, v$) ;
```

Algorithm 4 updateTreeMat.

```
Input  : $v$ is a vertex in $P_T$; $u$ is a data vertex to which $v$ maps in TreeMat.
1 if vertex $u$ is moved to match($v$) in last iteration then
2     foreach child vertex $v_c$ of $v$ do
3         foreach neighbor $u_c$ of $u$ such that $\langle u, u_c\rangle$ matches $\langle v, v_c\rangle$ do
4             chooseVST($\langle u, u_c\rangle$);
5             if TreeMat.getTransition($\langle u, u_c\rangle$)=true then
6                 updateTreeMat($u_c, v_c$);
7 if vertex $u$ is moved to stree($v$) in last iteration then
8     foreach neighbor $u_p$ of $u$ such that $\langle u, u_p\rangle$ matches $\langle v, v_p\rangle$ where $v_p$ is the
       parent vertex of $v$ do
9         chooseVST($\langle u, u_p\rangle$);
10        if TreeMat.getTransition($\langle u, u_p\rangle$)=true then
11            updateTreeMat($u_p, v_p$);
```

4.2 Edge updates on the data graph

Now, we explain G-insertEval (Algorithm 3), which is invoked for each edge insertion $\langle u, u'\rangle$. The main idea of G-insertEval is explained as follows: we try to match $\langle u, u'\rangle$ with tree edges in P_T and then update the TreeMat through the vertex position transition strategy. Note that there may be more than one query edge in P_T to which $\langle u, u'\rangle$ matches, and not all matching situations can cause the update of TreeMat. For this purpose, we should exclude the invalid matching situations.

In order to exclude invalid matching situations, we first obtain the query edges in P_T with the same edge label as $\langle u, u'\rangle$. Let v be the parent of v'. Then, for each matched query edge $\langle v, v' \rangle$, we check whether $u' \in \mathsf{cand}(v')$; if not, it will not cause the update of TreeMat and will be ignored (Line 1–3). For each valid matching situation, we execute chooseVST to check whether $\langle u, u' \rangle$ can cause the update of TreeMat (Line 5). If so, chooseVST chooses the corresponding transition rule and updates the states of u and u'. What's more, chooseVST will also check whether the update caused by $\langle u, u' \rangle$ needs to be propagated upwards or downwards. If so, we set TreeMat.getTransition($\langle u, u' \rangle$)=**true** and update TreeMat by calling updateTreeMat (Algorithm 4) recursively (Lines 6–8). Here, updateTreeMat decides the update propagation direction (i.e., upwards or downwards) for current iteration and executes corresponding transition rule. Algorithms for edge deletions on the data graph are similar to those for edge insertions except that they use the transitions 4–6, instead of transitions 1–3; Omitted in the interest of space, the algorithm G-deleteEval (Line 9 of Algorithm 1) is not described here.

Algorithm 5: P-deleteEval.

```
Input : TreeMat: the auxiliary data structure;
        ⟨v, v'⟩: an edge to be deleted.
1  Set $V(u) \leftarrow 0$ for all $u$ in $G_0$;
2  if $\langle v, v' \rangle \in$ non-tree edge then
3      break
4  if $v'$ is a leaf vertex then
5      $N(v) \leftarrow 0$;
6      foreach child vertex $v_c$ of $v$ except $v'$ do
7          if $v_c$ is a non-leaf vertex then
8              foreach vertex $u_c \in \mathsf{cand}(v_c)$ do
9                  foreach vertex $u$ that is adjacent to $u_c$ do
10                     if $u \notin \mathsf{cand}(v)$ and $V(u) = N(v)$ and $\langle u, u_c \rangle$ matches $\langle v, v_c \rangle$
                        then
11                         $V(u) \leftarrow V(u) + 1$;
12         else
13             Same as Lines 9–10;
14         $N(v) \leftarrow N(v) + 1$;
15     foreach vertex $u$ with $V(u) = N(v)$ do
16         $\mathsf{stree}(v) \leftarrow \mathsf{stree}(v) \cup \{u\}$;
17         checkLeafVertex($u$);
18         updateTreeMat($u$, $v$);
19 else
20     $v'' \leftarrow$ translateTreeEdge($v'$); updateState($v'$);
21     Same as Lines 5–18;
22     foreach vertex $u''$ in $\mathsf{cand}(v'')$ do
23         if there is no vertex in $\mathsf{cand}(v')$ that is adjacent to $u''$ then
24             $\mathsf{cand}(v'') \leftarrow \mathsf{cand}(v'') \cap \{u''\}$;
25         else
26             updateTreeMat($u''$, $v''$);
27     updateAncestor($v''$);
```

4.3 Edge updates on the pattern graph

In this subsection, we introduce P-deleteEval (Algorithm 5), which is invoked for each edge deletion $\langle v, v' \rangle$.

We first check whether $\langle v, v' \rangle$ is a non-tree edge; if so, it will not cause the update of TreeMat (Line 2–3). In other case, if v' is a leaf vertex, some NULL vertices may be added into stree(v) under this situation. In detail, if a vertex u satisfies: (1) u has the same label as v; (2) $u \notin$ cand(v); and (3) for each child vertex v_c of v except v', there is a data vertex $u_c \in$ cand(v_c) that is adjacent to u, then we add u into stree(v) (Lines 4–16). Note that, if v_c is a leaf vertex, we should further check whether there is an edge $\langle u, u_c \rangle$ matching $\langle v, v_c \rangle$ and $u_c \in$ NULL; if so, add u_c into stree(v_c) (Line 17). After that, we call updateTreeMat (Algorithm 4) recursively to update the TreeMat based on the status of u (Line 18). What's more, if v' is not a leaf vertex, we should translate the non-tree edge with an endpoint of v' to tree edge. We also set the status of all the candidates of v' and the descendants of v' as stree at this condition (Line 20). Next, we update stree(v) in a similar way as Lines 5–18. Adding a non-tree edge into P_T will cause some candidate vertices to be executed. As a result, we should further check for each vertex $u'' \in$ cand(v''), if there is a vertex in $cand(v')$ that is adjacent to v''. If not, remove u'' from cand(v''); else we call updateTreeMat (Algorithm 4) recursively to update the TreeMat based on the status of u'' (Lines 22–26). The update is propagated upwards till the root vertex(Line 27).

Algorithms for edge insertions on the pattern graph are similar to those for edge deletions under the situation that v' is not a leaf vertex. Omitted in the interest of space, the algorithm P-insertEval (Line 11 of Algorithm 1) is not described here.

4.4 Cost-driven pattern matching

Pattern evaluation phase is to harvest complete solutions to pattern graphs by leveraging TreeMat. We are in quest of boosting performance by conducting exploration on TreeMat.

Standard backtracking is viable but inefficient, which neglects the matching order that may greatly affect the performance. A classic models for generic graph patten matching [1, 12] is as follows. Assume the total cost is proportional to the number of comparisons for determining whether a vertex (or an edge) matches. Given an arbitrary order of vertices $(v_1, v_2, \ldots, v_n)$ for P, the number of comparisons performed in a backtracking algorithm is

$$T_{iso} \triangleq T_{|V_P|} = |M_1| + \sum_{i=2}^{|V_P|} \sum_{j=1}^{|M_{i-1}|} |d_i^j| \cdot (r_i + 1), \tag{1}$$

where M_i represents the set of *intermediate results* for the subgraph of P induced by $(v_1, v_2, \ldots, v_i)$, d_i^j is the vertices in match(v_i) joinable with an intermediate result in M_{i-1}, and r_i is the number of non-tree edges between v_i and vertices before v_i in the matching order.

Nonetheless, r_i largely depends on the actual order. The total number of configurations of r_i is exponential in $O(|V_P|!)$, and thus, it is prohibitively expensive to optimize T_{iso} online. In response, we choose to minimize T_{iso} greedily, i.e., every time choose the vertex of the minimum cost on the basis of current intermediate results. Then, to match vertex v_i, the number of comparisons concerning v_i can be expressed by $T'(v_i) = \sum_{j=1}^{|M_{i-1}|} |d_i^j|(r_i + 1)$.

In addition, we unveil that the advantage of harnessing TreeMat also comes from the derivation of d_i^j given M_j, which is inaccessible in pattern matching. Recall that a likelihood estimated over entire topology graph is used to delegate d_i^j [1, 12], which can be inaccurate. Lastly, to select the first vertex, we choose the one with minimum $\frac{|\mathsf{match}(v)|}{deg(v)}$, where $deg(v)$ is the total degree of v.

The estimation above only considers the cost thus far (i.e., current cost), but ignores the cost from the vertices to be accessed (i.e., future cost). It is contended that combining current and future costs may provide rewarding guidance for future steps. However, it is non-trivial to precisely compute the actual intermediate results after mapping u_i. To this end, we heuristically estimate the number of intermediate results as

$$|M_i| = \sum_{j=1}^{|M_{i-1}|} |d_i^j| \cdot \prod_{j=0}^{n-1} p_j^i, \tag{2}$$

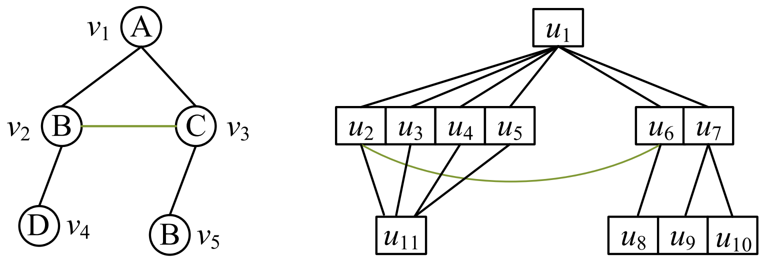
where p_j^i is the likelihood of a vertex in d_i^j has an edge satisfying the restriction of the j-th non-tree edge of v_i connecting to a vertex that has been accessed. Then, we estimate the number of intermediate results for each vertex that has not been accessed. Let v_k be an unvisited vertex, the number of intermediate results predicted for v_k is $|\mathsf{match}(v_k)| \times \prod_{j=0}^{n-1} p_j^k$. Thus, the summation becomes the total number of intermediate results predicted for all the vertices that have not been accessed. Then, the future cost of mapping vertex v_i can be expressed by

$$T''(v_i) = |M_i| \cdot \sum \left[|\mathsf{match}(v_k)| \times \prod_{j=0}^{n-1} p_j^k \right] (r_k + 1), \tag{3}$$

where r_k represents the number of vertices that has been accessed except the parent of v_i that has edges connected to unaccessed vertices.

In overall, the cost of mapping u_i can be estimated by $T'(v_i)+T''(v_i)$. Experiments show that it provides better guidance to the matching processing, in comparison with alternative strategies.

Example 4 Consider the pattern graph and the $\mathsf{match}(\cdot)$ set of TreeMat in Figure 4. v_1 is set as the root vertex since $\frac{\mathsf{match}(v_1)}{2}$ is minimum. Suppose that the vertices v_1 and v_3 have been matched. At this time, the number of intermediate results is 2, and we are going to choose the next vertex. If we choose v_5, the number of comparisons is 1 + 2 = 3; if we choose v_2, the number of comparisons is $8 \times 2 = 16$. According to the greedy selection that only consider the current matching cost, we will choose v_5 as the next vertex, and the current



(a) Pattern graph P (b) match($\cdot$) set of TreeMat

Figure 4 Sample pattern graph and $\mathsf{match}(\cdot)$ set of TreeMat

total number of comparisons is $1 + 2 + 3 + 12 \times 2 + 1 = 31$. However, if we take the future matching cost into account, we will choose v_2 as the next vertex, and the total number of comparisons is $1 + 2 + 8 \times 2 + 1 + 1 = 21$ that is smaller than 31.

Correctness and complexity Based on the discussion, we can implement a procedure for choosing the next vertex for matching. Note that, f we use the new cost model may bring fewer cost. While the details of the procedure is omitted in the interest of space, it can be seen that the procedure runs in $O(|E_{G^*}| \times |V_{P^*}| \times |E_{P^*}|)$, where G^* and P^* are the updated data graph and updated pattern graph, respectively.

Remark In comparison with existing cost models for pattern matching and order selection, the proposed model and algorithm are advantageous in the sense that

– As identified by existing work [1], TurboFlux [10] fails to be applicable to large and complex query patterns; in contrast, CEPDG lends itself to large and complex queries against the more difficult matching criteria of subgraph isomorphism;
– Compared with QuickSI [12], which merely concentrate on a local cost with a greedy strategy, our proposed cost model generates a more effective matching order, which takes both existing and future costs into account, and hence, reduces a large number of unpromising intermediate results;
– In comparison with CFL [1], which implements a path-based cost model, our model chooses an edge-based cost most, and thus, is more flexible and less computationally expensive, while retaining the quality of order selection.

It can be seen that the cost-driven matching algorithm heavily relies on a good estimation of cand$(\cdot)$, and the more accurate estimation, the better guidance for matching ordering. In the sequel, we strive to offer a good estimation of candidates by levering an online saturation strategy with index support.

5 Experiments

In this section, we evaluate the performance of CEPDG against the state-of-the-art continuous subgraph matching methods, TurboFlux [10], and GraphFlow [9] on two real-life datasets. The source code of TurboFlux was obtained from its authors. The source code of GraphFlow was downloaded from github [2]. Then, we report experimental results and analyses.

5.1 Experiment setup

The proposed algorithms were implemented using C++, running on a Linux machine with two Core Intel Xeon CPU 2.2Ghz and 32GB main memory.

Datasets/Queries We used two datasets referred as Yago[3] and Netflow[4]. Yago is a dataset that extracts facts from Wikipedia and integrates them with the WordNet thesaurus. This

[2]https://github.com/graphflow/graphflow
[3]http://www.mpi-inf.mpg.de/yago-naga/yago/
[4]https://data.caida.org/datasets/passive-2014.

dataset consists of an initial graph G_0 and a graph update stream Δg. G_0 contains 12,375,749 triples while Δg consists of insertions of 1,124,302 triples and deletions of 1,027,828 triples. Netflow contains anonymized passive traffic traces monitored from high-speed internet backbone links. In this dataset, G_0 contains 14,378,113 triples and Δg consists of insertions 1,236,412 triples and deletions of 1,107,635 triples.

As the dataset does not come with patterns, we comprehensively generated various patterns as follows. We first make 4 pattern categories ($A1 \sim A4$), and then, extract for each category 20 patterns by randomly traversing the topology graph. The size of patterns in $A1$, $A2$, $A3$ and $A4$ is 15, 20, 25 and 30, respectively. Then, for each graph pattern, to generate the update stream, every time we (1)randomly removed an existing edge while keeping the pattern graph connected; and (2) randomly added an edge between two disconnected vertices with a random edge label conforming uniform distribution. Note that, the size of edge insertions/deletions of each pattern graph did not exceed half of the pattern size ($\leq 50\%$); otherwise, fundamental characteristics of the pattern disappear.

Algorithms Since there is no existing research directly targeting our problem, two state-of-the-art algorithms were adapted and involved for comparison: 1) TurboFlux [10] is an algorithm for pattern matching over dynamic graph; to deal with evolving pattern graph, it has to recompute its auxiliary data structure during update. 2) GraphFlow [9] is an incremental algorithm without maintaining intermediate results. 3) our proposed algorithm CEPDG.

Unless specified otherwise, values in boldface in Table 2 are used as default parameters in the experiments.

5.2 Evaluation of data graph updates

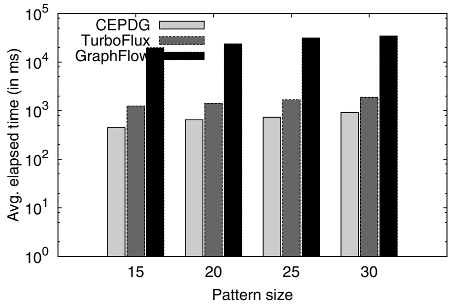
We use two measures, the average elapsed time and the size of intermediate results. Note that, for fair comparison, we exclude the elapsed time for updating the data graph. That is, we set the average elapsed time of CEPDG as the difference between the time for processing the graph update stream with and without continuous query answering, and measure the time of the competitors for query processing only. What's more, we conduct experiments by inserting/deleting edges in batches of 10K ($= 10 \times 10^3$). Inserting/deleting edges in batches means that we need only calculate matching results when all the edges have added into or removed from the data graph. We set a 1-hour timeout for each query.

5.2.1 Varying pattern size

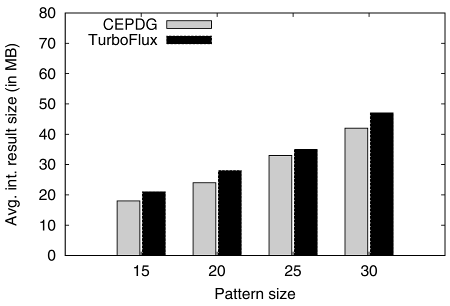
Figure 5 shows the performance results in Yago dataset. Here, we set edge insertions/deletions as 500K ($= 500 \times 10^3$) and vary the query size from 15 to 30. Figure 5(1)

Table 2 Parameters used in the experiments

Parameters	Values Used
Datasets	**Yago**, Netflow
Query size	15, 20, **25**, 30
Dataset size	0.2, 0.4, 0.6, 0.8, **1** (Yago)
Insertion/deletion size (data graph)	250K, **500K**, 750K, 1000K
Insertion/deletion size (pattern graph)	3, **6**, 9, 12



(1) Avg. elapsed time

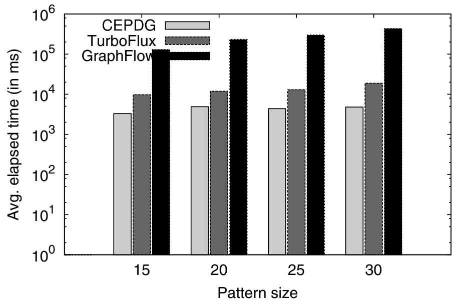


(2) Avg. int. result size

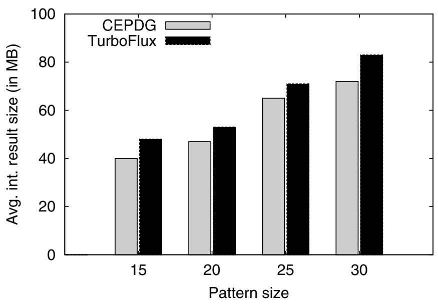
Figure 5 Performance for varying pattern size in Yago

shows the average elapsed time. CEPDG behaves better than its competitors regardless of pattern size. Specially, CEPDG outperforms TurboFlux by 2.28 $\sim$ 3.13 times, and GraphFlow by 36.67 $\sim$ 44.28 times. The reason is that GraphFlow does not maintain any intermediate results and it will generate a much larger number of partial solutions than CEPDG and TurboFlux. CEPDG only needs to update partial intermediate results for an edge update operation. So even $|E(P)|$ is big, CEPDG can also achieve a better performance. Moreover, CEPDG can significant reduce the time cost based on the cost model in the pattern matching process. Figure 5(2) shows the average number of intermediate results. Since GraphFlow does not maintain any intermediate results, we only compare CEPDG with TurboFlux. Specially, the average size of intermediate results of TurboFlux is larger than that of CEPDG by 1.28 $\sim$ 1.54 times. It means that the representation by CEPDG (TreeMat) is more concise than that by TurboFlux.

Figure 6 shows the performance results in Netflow dataset. CEPDG behaves better than its competitors in both of average elapsed time and average size of intermediate results regardless of pattern size. Specially, in Figure 6(1), CEPDG outperforms TurboFlux by up to 2.86 times, and GraphFlow by up to 90.72 times; in Figure 6(2), the average size of intermediate results of TurboFlux is larger than that of CEPDG by up to 1.47 times. This is because Netflow has only eight edge labels and no vertex label. Hence, the size of intermediate results is enormous, and time costs in TurboFlux and GraphFlow are very expensive.



(1) Avg. elapsed time



(2) Avg. int. result size

Figure 6 Performance for varying pattern size on Netflow

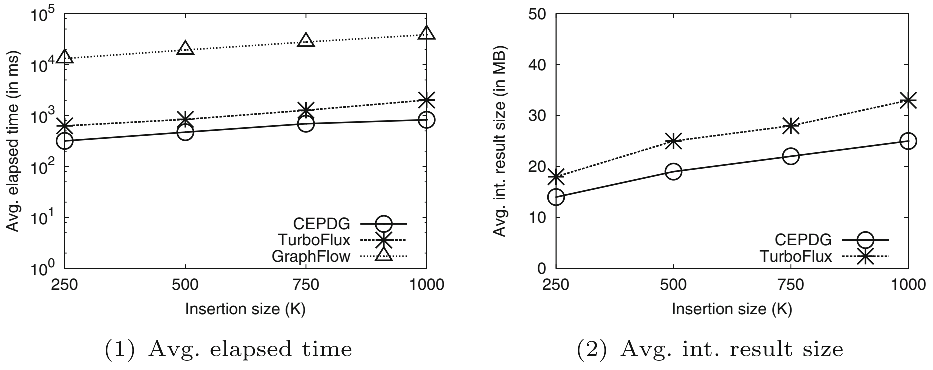


Figure 7 Performance for varying edge insertion size on Yago

5.2.2 Varying edge insertion size

In this subsection, we evaluate the impact of edge insertions of data graph on the performance of CEPDG and its competitors. Here, we fixed patterns in $A3$ and varied the number of newly-inserted edges from 250K (= 250×10^3) to 1000K in 250K increments on Yago. Thus, the number of total update operations also increases accordingly. Figure 7(1) shows the processing time for each algorithm. We see that CEPDG has consistently better performance than it competitors. What's more, the figure reads a non-exponential increase as edge insertion size grows. Specially, CEPDG outperforms TurboFlux by up to 2.44 times, and GraphFlow by up to 46.78 times at edge insertion size 1000K. CEPDG also outperforms its competitors in terms of the size of intermediate results as shown in Figure 7(2). Specially, the size of intermediate results of TurboFlux is larger than that of CEPDG by up to 1.43 times when the insertion size is 1000K.

5.2.3 Varying edge deletion size

In this subsection, we evaluate the impact of edge deletions of data graph on the performance of CEPDG and its competitors. Here, we fixed patterns in $A3$ and varied the number of deleted edges from 250K (= 250×10^3) to 1000K in 250K increments on Yago. Figure 8(1)

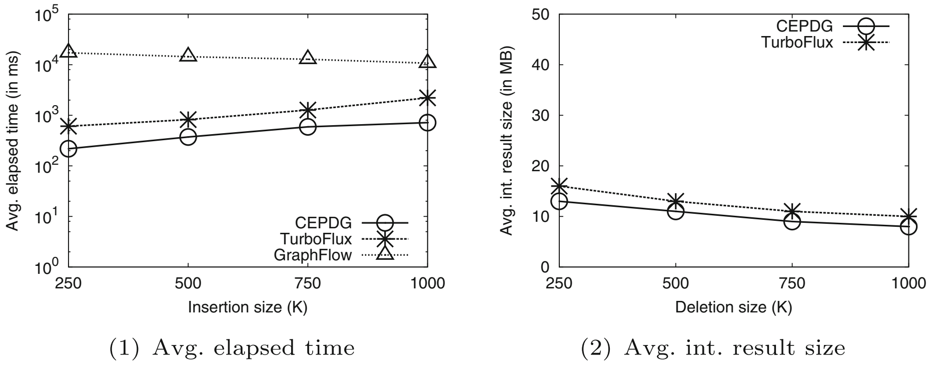


Figure 8 Performance for varying edge deletion size on Yago

shows the processing time for each algorithm. Note that the gap between the performance of CEPDG and TurboFlux is larger than that in Figure 7(1). This is because deletion of an edge (u, u') could affect all subtrees of u' in TurboFlux. However, in CEPDG, we need only main the affected vertices in TreeMat, which relatively small. Note also that the processing GraphFlow slightly decreases when the size of edge insertions decrease. This is because, the edge deletions reduce the input data size of GraphFlow directly. Specially, CEPDG outperforms TurboFlux by up to 3.16 times, and GraphFlow by up to 74.51 times. CEPDG also outperforms its competitors in terms of the size of intermediate results as shown in Figure 8(2). Specially, the size of intermediate results of TurboFlux is larger than that of CEPDG by up to 1.25 times when the insertion size is 1000K.

5.2.4 Varying the data size

In this testing, we evaluate the performance results of CEPDG against existing algorithms regarding the scalability by using Yago for varying dataset size. Here, we fixed patterns in $A3$, set edge insertions/deletions as 500K ($= 500 \times 10^3$), and randomly sampled about 20% to 100% from the Yago dataset so that the data and result distribution remain approximately the same with the whole dataset. Then, we plot the total processing time and the size of intermediate in Figure 9.

It is revealed that CEPDG consistently outperforms its competitors regardless of the dataset size. In generally, CEPDG and TurboFlux show similar performance for all sizes of datasets. This can be attributed to the proposed pruning and validation technique, which dramatically reduces the required sample size and maintains the intermediate results incrementally. The scalability suggest that CEPDG and TurboFlux can handle reasonably large real-life graphs as those existing algorithms for deterministic graphs. Specially, CEPDG outperforms TurboFlux by up to 2.14 times, and GraphFlow by up to 37.57 times. Figure 9(2) shows similar scalability of intermediate result sizes for CEPDG and TurboFlux. The size of intermediate results of TurboFlux is larger than that of CEPDG by up to 1.64 times.

5.2.5 Evaluating the effectiveness of the cost model

In this subsection, we evaluate the effectiveness of our proposed cost model. We compare the time cost in pattern matching with the state-of-the-art algorithm CFL [1] over Yago and Netflow dataset, respectively. Since the size of candidates is also a key factor affecting

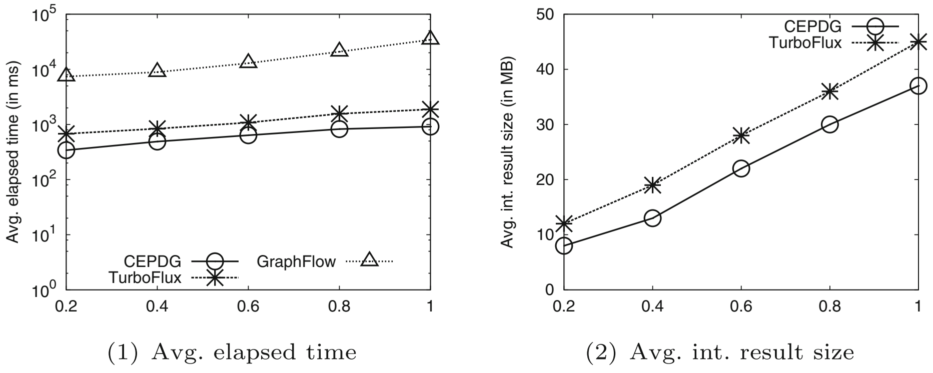


Figure 9 Performance for varying data size on Yago

the running time despite matching order, for a fair comparison, we choose to use the same candidate set for every pattern vertex in both solutions. Here, we use the match(·) set of TreeMat as candidates and plot the running time in Figure 10.

It is revealed that our proposed cost model never perform worse than that in CFL. In specific, it can help lower the time cost by a factor of 10. The reason is that CFL implements a path-based cost model. The path selected each time is that with minimal growth in result size, and after dealing with this path, a new growing path will be selected. Compared with CFL, our cost model does not estimate the cost for each path, but analyses the cost for each edge and considers the cost of next and current steps. Adjusting the cost model is more flexible after joining an edge than joining a path. The result can also prove that our cost model is close to the real cost of the join process. Otherwise, the new join strategy will not work well and may choose an awful edge in some steps, which results in the cost of the join process being high.

5.3 Evaluation of pattern graph updates

In this section, we measure the average elapsed time and the size of intermediate results of CEPDG and TurboFlux.

5.3.1 Comparison of different matching orderings

In this set of experiments, we ran CEPDG using patterns in $A3$, and measured the average elapsed time for unit insertion/deletion, but applied three different ordering strategies—randomly choose an order, greedily choose as per the current cost , and greedily choose as per the estimated overall cost (our method). The results are within expectation that our proposed strategy outperforms the first strategy by 20.12x/23.74x, and the second strategy by 7.26x/8.12x for each edge insertion/deletion.

5.3.2 Varying pattern size

In this set of experiments, we demonstrate the advantage of CEPDG regarding update. We used two measures—average elapsed time for unit insertion/deletion and size of partial solutions. Figure 11 shows the comparison against edge insertion, where we varied pattern size from 15 to 30 ($A1 \sim A4$). Note that the matching cost does not always increase as the

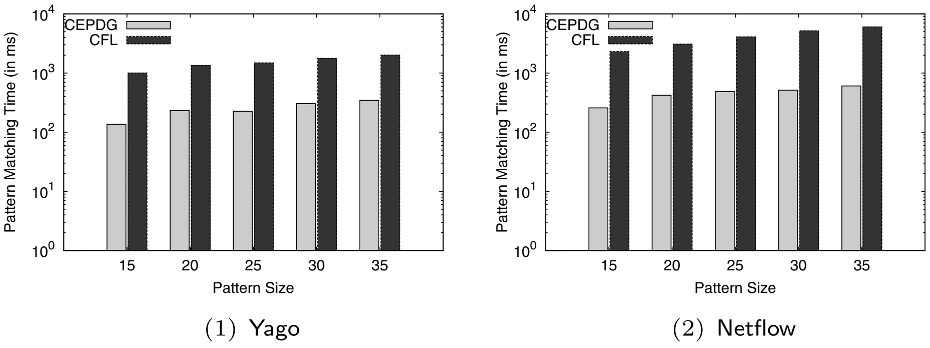


Figure 10 Comparing partial performance

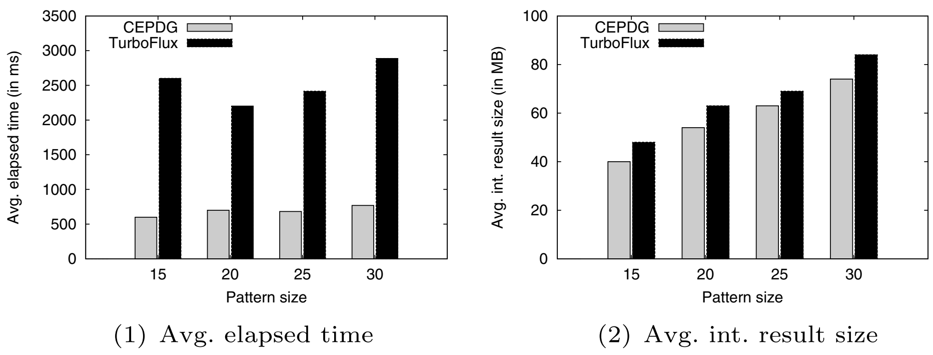


Figure 11 Efficiency against edge insertion

pattern size increases. Specially, in Figure 11(1), CEPDG outperforms TurboFlux by up to 4.36 times. Note that TurboFlux has to recompute the auxiliary data structure, which is not rewarding under this setting. Figure 11(2) shows the size of partial solutions, which reads that the size of partial results of CEPDG is smaller than that of TurboFlux. This is intuitive, the representation by CEPDG (TreeMat) is more concise than that by TurboFlux.

Figure 12 shows the comparison against edge deletion, and similar trends are observed as from Figure 8. Figure 12(1) shows the average elapsed time by the two algorithms. CEPDG significantly outperforms TurboFlux in all cases. Specially, CEPDG outperforms TurboFlux by up to 3.43 times. Further, the average elapsed time for a deletion is much longer than that for an insertion, which suggests deleting an edge from the pattern graph may be more computationally expensive. Figure 12(2) shows the average size of partial solutions. The average size by TreeMat is significantly smaller than that of TurboFlux by up to 1.14 times.

5.3.3 Varying edge update volume

In this set of experiments, we evaluate the impact of the number of edge updates on the performance of CEPDG (and its alternatives). We fixed patterns in $A3$ and varied edge updates from 3 to 12 in 3 increments. Figure 13 shows the average elapsed time for each algorithm. We see that CEPDG has a better performance than others. In Figure 13(1), it witnesses a non-exponential increase as insertions grows, and CEPDG beats TurboFlux by

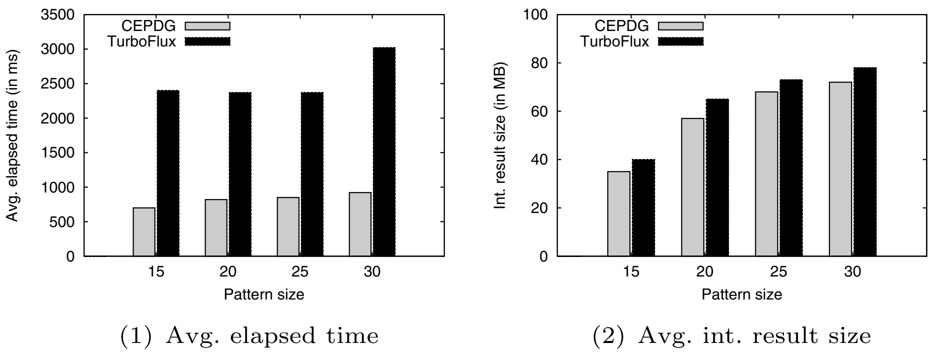
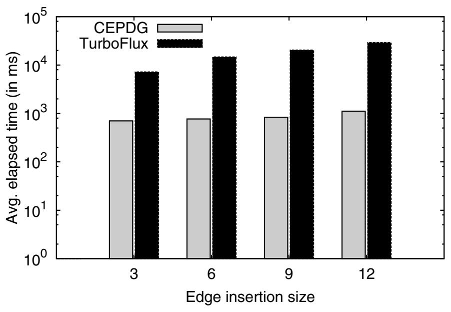
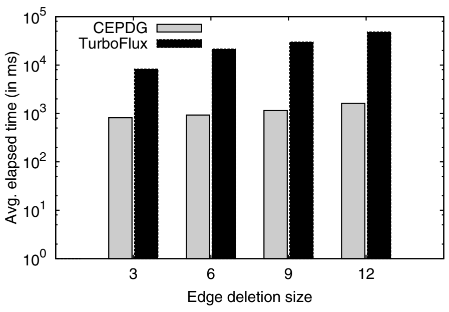


Figure 12 Efficiency against edge deletion



(1) Avg. elapsed time



(2) Avg. elapsed time

Figure 13 Performance for varying edge update volume

up to 26.12 times. In Figure 13(2), for edge deletion, the performance gap is slightly larger than that for edge insertion, and CEPDG is more efficient than TurboFlux by up to 29.82 times.

6 Related work

This section categorizes related work on graph pattern matching into two streams: static and dynamic.

Graph pattern matching Graph pattern matching is typically defined in terms of subgraph isomorphism [4], which has been studied extensively since 1976. A key issue of subgraph isomorphism is to reduce the number of unpromising intermediate results when iteratively mapping vertices one by one from a pattern graph to a data graph. VF2 [4] and QuickSI [12] propose to enforce the connectivity to prune the candidates. TurboISO [8] proposes to merge together the nodes in a pattern graph with the same labels and the same neighborhoods to further reduce unpromising candidates. Another key issue is to generate an effective matching order. QuickSI [12] proposes to generate a matching order based on the infrequent-labels first strategy. SPath [17] proposes to generate a matching order based on the infrequent-paths first strategy, but the efficiency will get lower when the size of a patten graph get larger. Bi et al. [1] develops a new framework that decomposes a pattern graph into a core and a forest for graph pattern matching. They showed that the core-forest-leaf ordering effectively reduces redundant Cartesian products. Han et al. [7] proposes novel techniques for subgraph matching: dynamic programming between a DAG and a graph, adaptive matching order with DAG ordering, and pruning by failing sets. These methods work well on static graphs. However, substantial work is needed to support dynamic graphs.

Dynamic graph pattern matching As graphs are dynamic in nature in real-life applications, pattern matching over a large dynamic graph attracts more attention. INCISOMAT [5] identifies the the nodes of data graph that may produce new matches according to the changes of data graph. But the number of these nodes will get larger when the pattern graph gets larger and the efficiency will decrease dramatically. GraphFlow [9] applies a worst-case optimal join algorithm called *Generic Join* to incrementally evaluate subgraph matching for each update without maintaining intermediate results. For each query edge (v, v') that matches an updated edge (u, u'), Graphflow evaluates subgraph matching starting from a partial solution $\{(v, u), (v', u')\}$. SJ-TREE [2] decomposes the main pattern graph based on the selectivity of vertice attributes, the highly selective sub-pattern is evaluated first, and the remaining sub-patterns are evaluated only when new results are found in sub-patterns evaluated previously. Thus, a lot of unnecessary computational cost is avoided. But the decomposition features are simple, lots of intermediate results will be produced when the pattern graph gets larger. The pattern decomposition approach of the work in [6] is based on identifying optimal sub-DAGs (directed acyclic graph) in the pattern graph. The DAGs' are then traversed to identify source and sink vertices to define message transition rules in the *Giraph* framework. This approach is on distributed implementation and it is not suitable for all types of patterns. TurboFlux [10] is the state-of-the-art algorithm for continuous subgraph matching, which employs a data-centric representation of intermediate results, in the sense that the query pattern P is embedded into the data graph G and its execution model allows fast incremental maintenance. Wang and Chen [13] also deals with continuous subgraph matching for evolving graphs. However, this method produces *approximate* results only, while our approach generates exact results.

Above algorithms only solve the graph pattern matching problem under the scenario of the data graph updating alone. In this paper, we propose to investigate a new problem, *continuous matching of evolving patterns over dynamic graph data*, to report matches for each update operation in the graph update stream continuously.

7 Conclusion

In this paper, we are devoted to a more complicated but very practical graph pattern matching problem, *continuous matching of evolving patterns over dynamic graph data*, and the investigation presents a novel algorithm CEPDG for continuously pattern matching along with changes of both pattern graph and data graph We showed that CEPDG solved the problems of existing methods and efficiently processed continuous subgraph matching for each update operation on the data graph and pattern graph.

We first proposed a concise representation TreeMat based on the spanning tree of the initial pattern graph for storing partial solutions. We then proposed the vertex state transition strategy, which efficiently identifies which update operation on the data graph can affect the current partial solutions and maintain TreeMat accordingly. We next presented an execution model to efficiently and incrementally maintain the representation during edge updates on the pattern graph, which are compatible with the algorithm proposed for data graph very well. Finally, we conceived an effective cost model for estimating step-wise cost of pattern matching.

Extensive experiments showed that CEPDG outperformed existing competitors by up to orders of magnitude. Overall, we believe our continuous subgraph matching solution provides comprehensive insight and a substantial framework for future research.

Acknowledgements This work is supported by the National key research and development program under Grant Nos. 2018YFB1800203 and 2018YFE0207600.

References

1. Bi, F., Chang, L., Lin, X., Qin, L., Zhang, W.: Efficient subgraph matching by postponing cartesian products. In: SIGMOD' 16, San Francisco, CA, USA, June 26 - July 01, 2016, pp. 1199–1214 (2016)
2. Choudhury, S., Holder, L.B. Jr.., Agarwal, G.C.K., Feo, J.: A selectivity based approach to continuous pattern detection in streaming graphs. In: EDBT' 15, Brussels, Belgium, March 23-27, 2015, pp. 157–168 (2015)
3. Choudhury, S., Holder, L.B. Jr., Ray, G.C., Beus, A., Feo, S.J.: Streamworks: A system for dynamic graph search. In: Ross, K.A., Srivastava, D., Papadias, D. (eds.) Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013, pp. 1101–1104. ACM (2013)
4. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell **26**(10), 1367–1372 (2004)
5. Fan, W., Li, J., Luo, J., Tan, Z., Wang, X., Wu, Y.: Incremental graph pattern matching. In: SIGMOD'11, Athens, Greece, June 12-16, 2011, pp. 925–936 (2011)
6. Gao, J., Zhou, C., Yu, J.X.: Toward continuous pattern detection over evolving large graph with snapshot isolation. VLDB J. **25**(2), 269–290 (2016)
7. Han, M., Kim, H., Gu, G., Park, K., Han, W.: Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In: Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019, pp. 1429–1446 (2019)
8. Han, W., Lee, J., Lee, J.: Turbo$_{\mathrm{iso}}$: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In: SIGMOD'13, New York, USA, June 22-27, 2013, pp. 337–348 (2013)
9. Kankanamge, C., Sahu, S., Mhedbhi, A., Chen, J., Salihoglu, S.: Graphflow: An active graph database. In: SIGMOD'17, Chicago, IL, USA, May 14-19, 2017, pp. 1695–1698 (2017)
10. Kim, K., Seo, I., Han, W., Lee, J., Hong, S., Chafi, H., Shin, H., Jeong, G.: Turboflux: A fast continuous subgraph matching system for streaming graph data. In: SIGMOD'18, Houston, TX, USA, June 10-15, 2018, pp. 411–426 (2018)
11. Ouyang, D., Yuan, L., Qin, L., Chang, L., Zhang, Y., Lin, X.: Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees. Proc. VLDB Endow. **13**(5), 602–615 (2020)
12. Shang, H., Zhang, Y., Lin, X., Yu, J.X.: Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. PVLDB **1**(1), 364–375 (2008)
13. Wang, C., Chen, L.: Continuous subgraph pattern search over graph streams. In: ICDE'09, Shanghai, China, March 29 - April 2, 2009, pp. 393–404 (2009)
14. Yuan, L., Qin, L., Lin, X., Chang, L., Zhang, W.: Diversified top-k clique search. VLDB J. **25**(2), 171–196 (2016)
15. Yuan, L., Qin, L., Zhang, W., Chang, L., Yang, J.: Index-based densest clique percolation community search in networks. IEEE Trans. Knowl. Data Eng. **30**(5), 922–935 (2018)
16. Zhang, Q., Guo, D., Zhao, X., Guo, A.: On continuously matching of evolving graph patterns. In: Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM 2019, Beijing, China, November 3-7, 2019, pp. 2237–2240 (2019)
17. Zhao, P., Han, J.: On graph query optimization in large networks. PVLDB **3**(1), 340–351 (2010)