# BICE: Exploring Compact Search Space by Using Bipartite Matching and Cell-Wide Verification

Yunyoung Choi
Alsemy
Seoul, South Korea
yunyoungchoi96@gmail.com

Kunsoo Park
Seoul National University
Seoul, South Korea
kpark@theory.snu.ac.kr

Hyunjoon Kim[*][†]
Hanyang University
Seoul, South Korea
hyunjoonkim@hanyang.ac.kr

## ABSTRACT

Subgraph matching is the problem of searching for all embeddings of a query graph in a data graph, and subgraph query processing (also known as subgraph search) is to find all the data graphs that contain a query graph as subgraphs. Extensive research has been done to develop practical solutions for both problems. However, the existing solutions still show limited query processing time due to a lot of unnecessary computations in search. In this paper, we focus on exploring as compact search space as possible by using three techniques: (1) pruning by bipartite matching, (2) pruning by failing sets with bipartite matching, and (3) cell-wide verification. We propose a new algorithm BICE, which combines these three techniques. We conduct extensive experiments on real-world datasets as well as synthetic datasets to evaluate the effectiveness of the techniques. Experiments show that our approach outperforms the fastest existing subgraph search algorithm by up to two orders of magnitude in terms of elapsed time to process a query. Our approach also outperforms state-of-the-art subgraph matching algorithms by up to two orders of magnitude.

**PVLDB Reference Format:**
Yunyoung Choi, Kunsoo Park, and Hyunjoon Kim. BICE: Exploring Compact Search Space by Using Bipartite Matching and Cell-Wide Verification. PVLDB, 16(9): 2186 - 2198, 2023.
doi:10.14778/3598581.3598591

**PVLDB Artifact Availability:**
The code, data, and other artifacts have been made available at https://github.com/SNUCSE-CTA/BICE.

## 1 INTRODUCTION

In recent decades, researchers have been motivated to develop efficient algorithms to analyze graphs in various domains. *Subgraph matching* and *subgraph query processing (or subgraph search)* are fundamental problems arising in these domains.

Given a query graph $q$ and a data graph $G$, subgraph matching is the problem of finding all distinct *embeddings* of $q$ in $G$. Given a query graph $q$ and a set $\mathcal{D}$ of data graphs, subgraph search is the problem of finding all the data graphs that contain $q$ as subgraphs. Both problems have a variety of real-world applications [37, 39] such as social network analysis [11, 41], RDF query processing [24, 25], protein-protein interaction (PPI) network analysis [7, 34], and chemical compound search [23, 47]. For example, researchers find a given pattern in a large social network, and search biomedical networks for important subgraphs by counting their occurrences. In organic chemistry, patterns of atoms called functional groups are considered to indicate the molecules' properties. The number of occurrences of the substructure has been used to produce molecular fingerprints [31] and compute similarities between molecules [2, 35]. National Institutes of Health (NIH) provides a web user interface for subgraph search of chemical compounds in a chemistry database called PubChem[1]. In knowledge graphs, common substructures are extracted by querying them in the larger target graph [33].

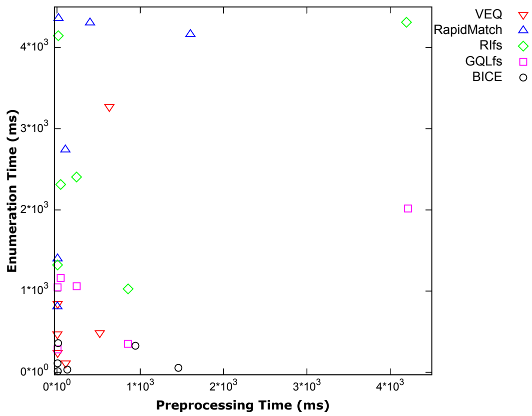


**Figure 1: Preprocessing time vs. enumeration time of state-of-the-art subgraph matching algorithms for six datasets**

These problems are based on *subgraph isomorphism*, which is a well-known NP-hard problem [12]. Therefore, solving these problems is a bottleneck in overall performance of such applications.

Extensive research has been done to develop practical solutions for these problems. The recent study [21, 42] on subgraph search used the *filtering-verification* strategy: (1) given a query graph $q$ and a set $\mathcal{D}$ of data graphs, filter out false answers in $\mathcal{D}$, and (2) a subgraph isomorphism test is performed against every remaining candidate graph in a verification step. The recent study [5, 14, 15, 43] on subgraph matching proposed algorithms based on the *preprocessing-enumeration* framework: (1) an auxiliary data

[*] Corresponding author.
[†] Department of Data Science, Department of Artificial Intelligence.

Proceedings of the VLDB Endowment, Vol. 16, No. 9 ISSN 2150-8097.
doi:10.14778/3598581.3598591

[1] https://pubchem.ncbi.nlm.nih.gov/

structure on a query graph and a data graph is constructed, and (2) all matches of the query graph are found by using the data structure.

These algorithms adopt the backtracking approach (in the enumeration of subgraph matching and in the verification of subgraph search), which recursively extends a partial embedding of a query graph by mapping the next query vertex to a data vertex. Although these efforts achieved some success, these algorithms still show limited response time and scalability as there could be a lot of redundant computations in the search process by the nature of backtracking. Figure 1 compares the mean preprocessing time and the mean enumeration time for the queries finished within a time limit of 10 minutes by state-of-the-art subgraph matching algorithms on six real-world datasets. All algorithms (except BICE) spend much more time in enumeration (i.e., backtracking) than in preprocessing. This phenomenon motivates us to design techniques that dramatically reduce the search space of backtracking.

To tackle this issue, we propose a new subgraph matching and subgraph search framework BICE, which significantly reduces search space of backtracking with three novel techniques. We make the following contributions.

**Pruning by bipartite matching**. At each node of the search tree during backtracking, we can construct a bipartite graph between unmapped query vertices and their candidate vertices of the data graph, and compute a maximum matching of the bipartite graph. If the size of the maximum matching in the bipartite graph is smaller than the number of unmapped query vertices (i.e., there is no way to match the unmapped query vertices to their candidate vertices), it is guaranteed that there is no embedding in the subtree of this node in the search tree. Thus we can prune the subtree of this node and backtrack. This bipartite graph matching technique is very effective in pruning the search tree, but there is an overhead to compute the maximum matching in the bipartite graph. In our technique, however, we inherit the maximum matching of the parent node in the search tree, and compute only additional matchings in the current node. Hence the overhead of the maximum matching computation is marginal.

**Pruning by Failing Sets with Bipartite Matching**. It has been shown that the failing set is an effective technique in pruning the search tree [14, 43]. We combine the failing set with our new bipartite matching technique so that we can compute failing sets in the search tree pruned by bipartite matching. Consequently, taking full advantage of two different pruning techniques (i.e., failing set and bipartite matching) can reduce a lot of search space.

**Cell-Wide Verification**. During backtracking in previous research, each query vertex is mapped to a single data vertex, which is called a mapping. In cell-wide verification, we map a query vertex to a cell (which is a set of candidate vertices with the same neighbors), which we call a hypermapping. Instead of computing mappings, we compute hypermappings in the backtracking, saving a lot of computation in the search space.

**Experiments**. We conduct extensive experiments on real-world datasets as well as synthetic datasets to evaluate the effectiveness of the techniques. Experiments show that our approach outperforms the fastest existing subgraph search algorithm $\mathsf{VEQ_S}$ [21, 22] by up to two orders of magnitude in terms of elapsed time to process a query. Furthermore, our approach also outperforms state-of-the-art subgraph matching algorithms by up to two orders of magnitude.

The rest of the paper is organized as follows. Section 2 gives the problem definition and related work. Section 3 provides a brief overview of our approach. Section 4 introduces pruning by bipartite matching. Section 5 describes computing failing sets by using bipartite matching. Section 6 presents a new backtracking method based on cell-wide verification. Section 7 presents an experimental comparison with previous work, and Section 8 concludes the paper.

## 2 PRELIMINARIES

For simplicity, we focus on connected graphs with labeled vertices. Our techniques can be easily extended to disconnected graphs with labeled edges. A graph $g = (V(g), E(g), L_g)$ consists of a set $V(g)$ of vertices, a set $E(g)$ of edges, and a labeling function $L_g : V(g) \to \Sigma$ that assigns a label to each vertex, where $\Sigma$ is a set of labels. A graph $g$ with no labels on vertices is denoted by $g = (V(g), E(g))$. For a subset $S$ of $V(g)$, an *induced subgraph* $g[S]$ denotes the subgraph of $g$ whose vertex set is $S$ and whose edge set consists of all the edges in $E(g)$ that have both endpoints in $S$.

Given graphs $q = (V(q), E(q), L_q)$ and $G = (V(G), E(G), L_G)$, an *embedding* of $q$ in $G$ is a mapping $M : V(q) \to V(G)$ such that:

(1) $M$ is injective (i.e., $M(u) \neq M(u')$ for $u \neq u'$ in $V(q)$).
(2) $L_q(u) = L_G(M(u))$ for every $u \in V(q)$ .
(3) $(M(u), M(u')) \in E(G)$ for every $(u, u') \in E(q)$.

We call that $q$ is *subgraph isomorphic* to $G$, denoted by $q \subseteq G$, if there exists an embedding of $q$ in $G$. A mapping that satisfies (2) and (3) is called a *homomorphism.* An embedding of an induced subgraph of $q$ in $G$ is called a *partial embedding*. For the sake of traceability, we enumerate the mapping pairs in a partial embedding $M$ in the order in which they are added to $M$ during backtracking.

A *bipartite graph* is a graph whose vertices can be partitioned into two disjoint sets such that no two vertices within a same set are adjacent. We will use the bipartite graph as a tool to reduce search space of backtracking. Given a bipartite graph $g = (V(g), E(g))$, a *bipartite matching* in $g$ is a set of pairwise non-adjacent edges in $E(g)$. A *maximum bipartite matching* in $g$ is a bipartite matching in $g$ that contains the largest number of edges. A bipartite graph will be denoted by $g = (V_1(g), V_2(g), E(g))$ in which $V_1(g)$ and $V_2(g)$ are disjoint sets of $V(g)$.

Table 1 lists the notations frequently used in the paper.

**Table 1: Notations.**

| Symbol | Definition |
|---|---|
| $G$ | Data graph |
| $q$ | Query graph |
| $\mathcal{D}$ | Set of data graphs |
| $M$ | Partial embedding of $q$ in $G$ |
| $d_M$ | Dynamic DAG of $q$ regarding $M$ |
| $C(u)$ | Set of candidate vertices of $u \in V(q)$ |
| $C_M(u)$ | Set of extendable candidates of $u$ regarding $M$ |
| $B_M$ | Candidate bipartite graph of $M$ |
| $\mathcal{M}$ | Partial hypermapping of $q$ in CS |

## 2.1 Problem Statement

**Subgraph Matching.** Given a query graph $q$ and a data graph $G$, the *subgraph matching problem* is to find all embeddings of $q$ in $G$.

**Subgraph Search.** Given a query graph $q$ and a set $\mathcal{D}$ of data graphs, the *subgraph search problem* is to find all data graphs in $\mathcal{D}$ that contains $q$ as subgraphs. That is, subgraph search is to compute the answer set $A_q = \{G \in \mathcal{D} \mid q \subseteq G\}$.

**Example 2.1.** Given query graph $q$ in Figure 2a and a data graph $G$ in Figure 2b, there are five embeddings of $q$ in $G$: $M_1 = \{(u_1, v_1), (u_2, v_3), (u_3, v_7), (u_4, v_{11}), (u_5, v_{10}), (u_6, v_9), (u_7, v_{13})\}$, $M_2 = \{(u_1, v_1), (u_2, v_3), (u_3, v_7), (u_4, v_{11}), (u_5, v_4), (u_6, v_9), (u_7, v_{13})\}$, $M_3 = \{(u_1, v_1), (u_2, v_4), (u_3, v_7), (u_4, v_{11}), (u_5, v_{10}), (u_6, v_9), (u_7, v_{13})\}$, $M_4 = \{(u_1, v_1), (u_2, v_5), (u_3, v_7), (u_4, v_{11}), (u_5, v_{10}), (u_6, v_9), (u_7, v_{13})\}$, and $M_5 = \{(u_1, v_1), (u_2, v_5), (u_3, v_7), (u_4, v_{11}), (u_5, v_4), (u_6, v_9), (u_7, v_{13})\}$.

**Example 2.2.** Given query $q$ in Figure 2a and a set $\mathcal{D} = \{G_1, G_2\}$ of data graphs in Figure 2b and Figure 2c, $q$ is subgraph isomorphic to only $G_1$.

## 2.2 Related Work

**Subgraph Matching.** Many subgraph matching algorithms [4, 5, 8, 9, 14, 15, 17, 28, 40, 43, 49, 50] are based on the backtracking framework.

VF2 [9], VF3 [8] and QuickSI [40] adopt *direct-enumeration* framework, which directly explores a data graph $G$ to enumerate all results. GraphQL [17], Turbo$_{\text{iso}}$ [15], CFL-Match [5], DAF [14], and VEQ [21] are based on the *preprocessing-enumeration* framework, i.e., they build auxiliary data structures with a query graph and a data graph to get a small set of candidate vertices for each query vertex. In fact, they vary significantly in performances, which rely on the size of search space in backtracking. To reduce the search space, state-of-the-art algorithms [14, 21] design pruning strategies which eliminate unnecessary computations of the search process originated from the nature of backtracking. They aim to prune out unpromising partial embeddings that will not lead to embeddings in the future by utilizing the knowledge gained from past exploration in search space, and they attain considerable performance improvement. For example, DAF [14] exploits the structure of a query graph to identify a set (i.e., *failing set*) of query vertices that may potentially be involved in each mapping failure, then it backtracks if the mapping of a query vertex not involved in the failure was just updated. VEQ [21, 22] prunes out unnecessary subtrees of a search tree by exploiting the equivalence of the subtrees.

Unlike backtracking search methods, *join-based* framework [1, 19, 30, 44] models subgraph homomorphism as a relational query and evaluate the query with relational operators such as selections and joins. To find either subgraph isomorphisms or subgraph homomorphisms, RapidMatch [44] utilizes graph structures to optimize relation filtering and join plan generation.

**Subgraph Search.** Most subgraph search algorithms [6, 10, 13, 26, 47, 51, 52] adopt the *indexing-filtering-verification* paradigm, which constructs indexes on a set $\mathcal{D}$ of data graphs, filters out data graphs that cannot lead to answers with the assistance of the indexes, and verify if each of remaining data graphs contains the query graph as a subgraph in a subgraph isomorphism test.

However, the indexing methods have a considerable limitation in scalability [16, 20, 42]. Researchers [21, 42] recently leverage subgraph matching algorithms based on *preprocessing-enumeration* paradigm. Without indexes, this approach such as CFQL [42] and VEQ [21, 22] shows better performance and scalability.

**Maximum Bipartite Matching.** Subgraph matching and subgraph search can be seen as assignment problems that map each vertex in a query graph to a distinct vertex in a data graph. SUMGRA [3] and SGMatch [38] use a bipartite graph between neighbors of a query vertex $u$ and neighbors of its candidate vertex $v$ as a local filter to check if $u$ and $v$ can match. However, applying the maximum bipartite matching problem globally to all query vertices in a partial embedding and their candidate vertices has been underexplored, which is the focus of this work.

**Graph Compression.** For subgraph matching, BoostIso [36, 46] compresses a data graph by merging symmetric vertices in preprocessing before it processes a query. The graph compression works well only for very dense data graphs according to [5].

# 3 OVERVIEW OF OUR APPROACH

We outline our subgraph matching algorithm and its modification for subgraph search.

## 3.1 Subgraph Matching

```
Algorithm 1: SubgraphMatching($q, G$)
  Input: query graph $q$, data graph $G$
  Output: all embeddings of $q$ in $G$
  $CS \leftarrow$ BuildCS($q, G$);
  $M \leftarrow \emptyset$;
  $H \leftarrow$ GetMaximumMatching($B_{\emptyset}, \emptyset$);
  Backtrack$_m$($q, CS, M, H$);
```

Given a query graph $q$ and a data graph $G$, Algorithm 1 performs subgraph matching of $q$ in $G$. Initially, BuildCS is invoked to build an auxiliary data structure *candidates space* (CS) on $q$ and $G$, which consists of the *candidate set* $C(u)$ for each vertex $u \in V(q)$ and edges between the candidates as in *extended DAG-graph DP* [21].

**Definition 3.1. (Extendable Vertex)** An unmapped vertex $u$ of query graph $q$ in a partial embedding $M$ is called *extendable* regarding $M$ if at least one neighbor of $u$ is mapped in $M$.

**Definition 3.2. (Extendable Candidates)** Suppose that we are given a partial embedding $M$. Given an unmapped query vertex $u \in V(q)$, let $n_1, n_2, ..., n_k$ be $u$'s neighbors mapped in $M$. The set $C_M(u)$ of $u$'s *extendable candidates* regarding $M$ is defined as follows

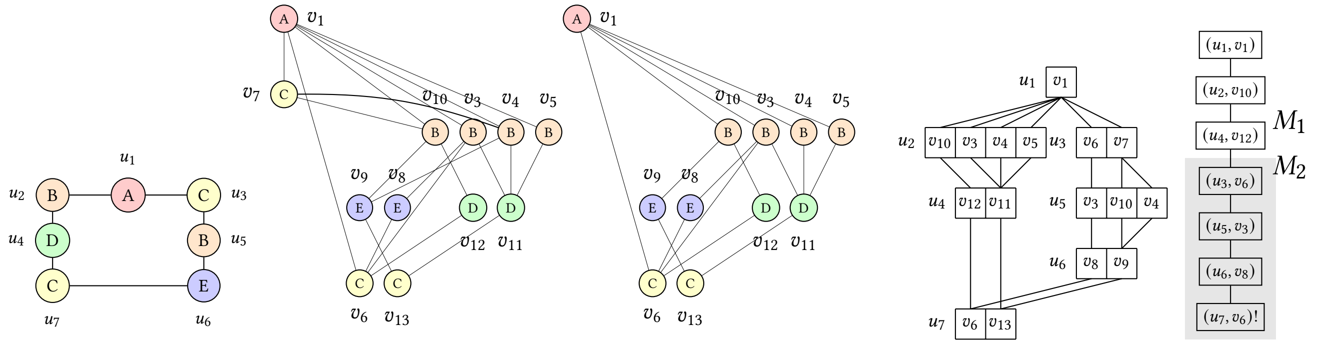
- If there are no mapped neighbors of $u$, $C_M(u) = C(u)$.
- Otherwise, $C_M(u)$ is the set of vertices $v \in C(u)$ adjacent to $M(n_i)$ in CS for every mapped neighbor $n_i$ of $u$.

Next, Backtrack$_m$ is invoked to find all embeddings of $q$ in CS by using three techniques in Sections 4, 5, and 6 to reduce the search space. We recursively extend a partial embedding by mapping an extendable vertex $u$ to each extendable candidate in $C_M(u)$ [21].

## 3.2 Subgraph Search

In Algorithm 2, we first initialize a set $A_q$ of answer graphs as $\emptyset$. For each data graph $G \in \mathcal{D}$, we repeat the following steps on $q$ and $G$. After all iterations over $\mathcal{D}$, we output the updated $A_q$.

(1) For the query graph $q$ and a data graph $G$, BuildCS is invoked to build CS in the same way as in subgraph matching.



**(a) Query graph $q$** **(b) Data graph $G_1$** **(c) Data graph $G_2$** **(d) CS** **(e) Search tree**

**Figure 2: Query graph $q$, Data graph $G_1$, Data graph $G_2$, CS on $q$ and $G_1$, and search tree for $q$ and $CS$ in Figure 2d**

```
Algorithm 2: SubgraphSearch($q, \mathcal{D}$)
  Input: query graph $q$, a set of data graphs $\mathcal{D}$
  Output: A set of answer graphs $A_q$
  $A_q \leftarrow \emptyset$;
  foreach $G \in \mathcal{D}$ do
      $CS \leftarrow$ BuildCS($q, G$);
      $M \leftarrow \emptyset$;
      $H \leftarrow$ GetMaximumMatching($B_\emptyset, \emptyset$);
      if Backtrack$_s$($q, CS, M, H$) returns $G$ then
          $A_q \leftarrow A_q \cup \{G\}$;
```

(2) *Backtrack$_s$* is invoked to find up to one embedding of $q$ in $G$. This step returns $G$ as an answer if it finds an embedding of $q$ in $G$; nothing otherwise. We add $G$ into $A_q$ if $G$ is an answer.

## 4 PRUNING BY BIPARTITE MATCHING

In this section, we propose a new technique to prune out unnecessary search space by using bipartite matching between query vertices and candidate vertices in the search process.

**Example 4.1.** Suppose that we constructed CS in Figure 2d from query graph $q$ in Figure 2a and data graph $G_1$ in Figure 2b. Figure 2e illustrates a part of search tree to find embeddings of $q$ in this CS. We first map $u_1$ to $v_1$, and then map $u_2$ to $v_{10}$, $u_4$ to $v_{12}$, and so on.

A node $(u, v)$ represents the last mapping of a partial embedding $M$. Let $M$ denote a partial embedding as well as a node of a search tree. We say that a node (or a partial embedding) of the search tree is *redundant* if it cannot lead to an embedding of $q$. Let a node $(u, v)!$ in a search tree denote a mapping conflict where $v$ is already mapped so we cannot map $u$ to $v$.

**Example 4.2.** In the search tree in Figure 2e, partial embedding $M_2$ is redundant, since extending $M_2$ will end up with a mapping conflict $(u_7, v_6)!$ between $(u_3, v_6)$ and $(u_7, v_6)$. Figure 3d illustrates the CS of Figure 2d regarding partial embedding $M_2$. Candidate vertices mapped in $M_2$ are colored with gray, and extendable candidates of unmapped query vertices are blue. Note that $u_7$ has only one extendable candidate $v_6$ which has already been mapped to $u_3$.

As shown in the above example, a partial embedding will not lead to an embedding if the extension of that partial embedding will result in only mapping conflicts. Thus, extending such partial embedding $M$ could cause huge redundant search space. To address this issue, we propose a new technique called *pruning by bipartite matching* that makes use of information obtained from CS.

### 4.1 Pruning by Bipartite Matching

**Definition 4.1. (Candidate Bipartite Graph on Partial Embedding)** Given CS on a query graph $q$ and a data graph $G$, and a partial embedding $M$, we define a *candidate bipartite graph* $B_M = (V(q), V(G), E(B_M))$ on $M$ as follows.

- There is an edge $(u, M(u))$ if $u$ is mapped in $M$; there is an edge between $u \in V(q)$ and every $v \in C_M(u)$ otherwise.

**Example 4.3.** Consider the partial embeddings $M_1$ and $M_2$ in the search tree of Figure 2e. Figure 3a and Figure 3c illustrate candidate bipartite graphs $B_{M_1}$ on $M_1$ and $B_{M_2}$ on $M_2$, respectively. A query vertex mapped in a partial embedding is only connected to its mapping in the bipartite graph. In Figure 3a, query vertices $u_1, u_2$, and $u_4$ are mapped in $M_1$, so each of them is connected to its mapping $M_1(u_1) = v_1$, $M_1(u_2) = v_{10}$, and $M_1(u_4) = v_{12}$, respectively. A query vertex not mapped in $M_1$ is connected to its extendable candidates, e.g., $u_3$ is connected to its extendable candidates $v_6, v_7$.

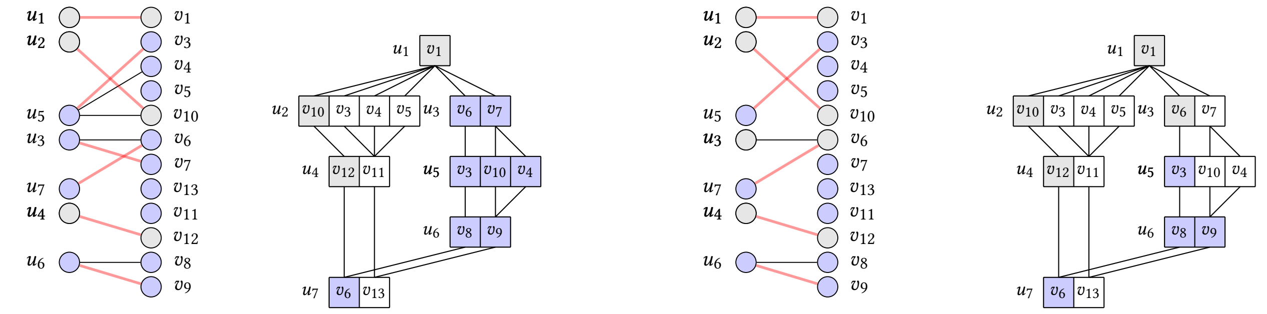
We observe that if the partial embedding $M$ can lead to an embedding, that embedding must be a bipartite matching in $B_M$.

**Lemma 4.1.** Given a candidate bipartite graph $B_M$ of a partial embedding $M$ and a maximum bipartite matching $H$ in $B_M$, partial embedding $M$ is redundant if $|H| < |V(q)|$.

By Lemma 4.1, we prune out the subtree rooted at partial embedding $M$ if $|H| < |V(q)|$ where $H$ is the maximum bipartite matching in $B_M$.

**Example 4.4.** Figure 3a illustrates the maximum matching $H_1 = \{(u_1, v_1), (u_2, v_{10}), (u_5, v_3), (u_3, v_7), (u_7, v_6), (u_4, v_{12}), (u_6, v_9)\}$ in $B_{M_1}$ with red lines. Since $|H_1| = |V(q)|$, we extend $M_1$ to $M_2$ in the search tree of Figure 2e. Figure 3c illustrates the maximum matching $H_2 = \{(u_1, v_1), (u_2, v_{10}), (u_5, v_3), (u_7, v_6), (u_4, v_{12}), (u_6, v_9)\}$ in $B_{M_2}$ (red edges). Since $|H_2| < |V(q)|$, $M_2$ is redundant, so the subtree rooted at $M_2$ (enclosed by the gray box) in Figure 2e is pruned out.

**Search Process**. Algorithm 3 illustrates our backtracking for subgraph matching, which prunes out search space by Lemma 4.1. Given query graph $q$, CS, current partial embedding $M$, and maximum matching $H$ in candidate bipartite graph $B_M$, Backtrack$_m$ searches CS for all embeddings of $q$.



(a) Bipartite graph $B_{M_1}$ (b) CS regarding $M_1$ (c) Bipartite graph $B_{M_2}$ (d) CS regarding $M_2$

**Figure 3: Candidate bipartite graphs and their corresponding CS**

```
Algorithm 3: Backtrack_m(q, CS, M, H)
1  if $|H| < |V(q)|$ then
2      return;
3  else if $|M| = |V(q)|$ then
4      Report $M$;
5  else
6      Select a next extendable vertex $u$;
7      foreach $v \in C_M(u)$ do
8          if $v$ is unvisited then
9              $M' \leftarrow M \cup \{(u, v)\}$;
10             $\widetilde{H} \leftarrow \{(s, t) \in H \mid s \notin \{u\} \cup UN_M(u)\}$;
11             $H' \leftarrow$ GetMaximumMatching($B_{M'}, \widetilde{H}$);
12             Mark $v$ as visited;
13             Backtrack_m($q, CS, M', H'$);
14             Mark $v$ as unvisited;
```

First, if $|H| < |V(q_d)|$, we immediately backtrack (lines 1-2). If $|M| = |V(q_d)|$, $M$ is an embedding of $q$, so we report $M$ (lines 3-4). Otherwise, we first select a next extendable vertex $u$ (line 6). For each unvisited extendable candidate $v \in C_M(u)$ (lines 7-8), we extend $M$ to $M' = M \cup \{(u, v)\}$ (line 9). We then compute a maximum matching $H'$ in candidate bipartite graph $B_{M'}$ on $M$ (lines 10-11). $UN_M(u)$ denotes a set of neighbors unmapped in $M$. Finding a maximum matching in $B_{M'}$ from scratch for every extension $M'$ may incur considerable computational overhead. Hence, we remove matchings incident to $u$ and to every unmapped neighbor of $u$ from $H$, and let $B_{M'}$ inherit from $B_M$ the remaining set $\widetilde{H}$ of matchings that both $B_M$ and $B_{M'}$ have in common (see Example 4.5). Next, we compute $H'$ in $B_{M'}$ from $\widetilde{H}$ (line 11) (see Section 4.2 for details). We recursively invoke Backtrack$_m$ with $M'$ (line 13). Backtrack$_s$ in Algorithm 2 is the same as Backtrack$_m$ of Algorithm 3 except line 4: Backtrack$_s$ returns data graph $G$, and terminates.

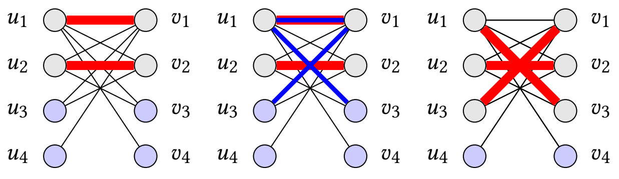
**Example 4.5.** Suppose that we just extended partial embedding $M_1$ to $M_2 = M_1 \cup \{(u_3, v_6)\}$ in Figure 2e, i.e., we selected $u_3$ as the next extendable vertex and $v_6$ as its extendable candidate to extend $M_1$ to $M_2$ in Algorithm 3. Note that candidate bipartite graph $B_{M_2}$ is obtained by Definition 4.1, as a neighbor $u_3$ of $u_5$ is newly mapped in $M_2$, which updates a set of $u_5$'s extendable candidates from $C_{M_1}(u_5) = \{v_3, v_{10}, v_4\}$ to $C_{M_2}(u_5) = \{v_3\}$ by Definition 3.2. Computing a maximum bipartite matching in $M_2$ from scratch is costly. Hence, we reuse a part of the maximum bipartite matching $H_1$ of $B_{M_1}$ in Figure 3a. Specifically, let $B_{M_2}$ inherit the subset $\widetilde{H} = \{(u_1, v_1), (u_2, v_{10}), (u_7, v_6), (u_4, v_{12}), (u_6, v_9)\}$ of $H_1 = \{(u_1, v_1), (u_2, v_{10}), (u_5, v_3), (u_3, v_7), (u_7, v_6), (u_4, v_{12}), (u_6, v_9)\}$. Indeed, $B_{M_1}$ in Figure 3a and $B_{M_2}$ in Figure 3c have edges incident to all vertices except $u_3$ and (its unmapped neighbor) $u_5$ in common. Note that $\widetilde{H}$ does not contain the edges incident to $u_3$ and $u_5$ in $H_1$. In Figure 3c, $\widetilde{H}$ is still a bipartite matching in $B_{M_2}$, so we will start computing the maximum bipartite matching in $B_{M_2}$ from $\widetilde{H}$.

## 4.2 Maximum Bipartite Matching Algorithm

For each partial embedding $M$ in the search process, we find a maximum bipartite matching in a candidate bipartite graph $B_M$. We describe a way to efficiently find the matching in this subsection.

**Definition 4.2. (Augmenting Path)** Given a bipartite graph $g = (V(g), E(g))$ and a bipartite matching $H$ in $g$, a vertex $v \in V(g)$ is called *free* if it is incident to no edges in $H$. A simple path $P$ in a bipartite graph is called an *augmenting path* relative to $H$ if both start and end vertices are free, and its edges are alternatively in $E(g) - H$ and in $H$.

**Definition 4.3. (Symmetric Difference)** Given two sets $A$ and $B$, $A \oplus B$ denotes the *symmetric difference* of $A$ and $B$, i.e., $A \cup B - A \cap B$.



(a) Bipartite matching $H$ (red) (b) Augmenting path $P$ (blue) (c) New matching $H \oplus P$ (red)

**Figure 4: Finding a larger bipartite matching $H \oplus P$ from a given bipartite matching $H$ using an augmenting path $P$**

**Theorem 4.1.** [18] We have the following properties.

(a) Given a matching $H$ and an augmenting path $P$ relative to $H$, $H \oplus P$ is a matching, and $|H \oplus P| = |H| + 1$.

(b) Suppose that we are given two bipartite matchings $H$ and $H'$ in a given bipartite graph. If $|H| > |H'|$, then $H \oplus H'$ contains at least $|H| - |H'|$ vertex-disjoint augmenting paths relative to $H$.

(c) $H$ is a maximum matching if and only if there are no augmenting paths relative to $H$.

Figure 4a shows bipartite matching $H = \{(u_1, v_1), (u_2, v_2)\}$ (red) in bipartite graph $g$, where $u_3, u_4, v_3$, and $v_4$ are free. Figure 4b shows an augmenting path $P = (u_3, v_1), (v_1, u_1), (u_1, v_3)$ (blue). Figure 4c shows a new maximum matching $H \oplus P = \{(u_1, v_3), (u_2, v_2), (u_3, v_1)\}$ (red). Note that $|H \oplus P| = |H| + 1$.

**Algorithm 4:** GetMaxiumMatching($g, H_{init}$)

```
1 $H \leftarrow H_{init}$;
2 $P \leftarrow$ GetAugmentingPath($g, H$);
3 while $P \neq \emptyset$ do
4     $H \leftarrow H \oplus P$;
5     $P \leftarrow$ GetAugmentingPath($g, H$);
6 return $H$;
```

Algorithm 4 finds the maximum bipartite matching $H$ based on Theorem 4.1. First, $H$ is initialized as $H_{init}$ (line 1). Next, GetAugmentingPath performs depth-first search (DFS) to find an augmenting path relative to $H$ in $g$ in the $O(|E(g)|)$ runtime (line 2). This returns an augmenting path if such a path exists; $\emptyset$ otherwise. We keep updating $H$ to $H \oplus P$, and invoking GetAugmentingPath with the updated $H$ as input, as long as an augmenting path $P$ exists (lines 3-5). If we no longer obtain an augmenting path, we return $H$, as $H$ is the maximum bipartite matching.

**Lemma 4.2. (Time Complexity)** Given a candidate bipartite graph $B_M = (V(q), V(G), E(B_M))$, Algorithm 4 takes $O((|V(q)| - |H_{init}|) \times |E(B_M)|)$ runtime in the worst case.

For further analysis on the runtime of Algorithm 4, we differentiate between following two distinct cases of $H_{init}$, which leads to a large difference of the time complexity.

**(1) Case $H_{init} = \emptyset$:** In Algorithm 1, GetMaxiumMatching($B_M, \emptyset$) is called before the first invocation of Algorithm 3. We therefore run up to $|V(q)|$ iterations of the while loop in Algorithm 4, which results in the total running time bounded by $O(|V(q)| \times |E(B_M)|)$.

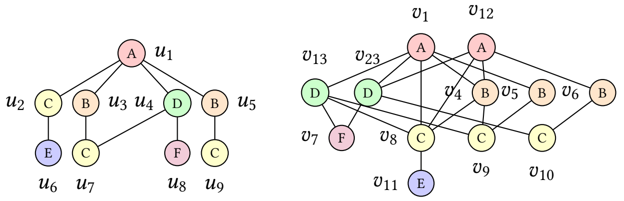
**(2) Case $H_{init} \neq \emptyset$:** Suppose that we are given a partial embedding $M$ and the maximum bipartite matching $H$ in $B_M$ in Algorithm 3. For every $x \in V(q) \setminus (\{u\} \cup UN_M(u))$, the neighbors of $x$ are exactly the same in both $B_M$ and $B_{M'}$ when we just extended $M$ to $M'$. Therefore, the subset $\widetilde{H} = \{(s,t) \in H_p | s \in V(q) \setminus (\{u\} \cup UN_M(u))\}$ of $H$ is a matching in $B_{M'}$. Recall that Algorithm 3 extracts $\widetilde{H}$ from $H$, and then Algorithm 4 is invoked with $H_{init} = \widetilde{H}$.

In Algorithm 4, we repeat finding augmenting paths for up to $|V(q)| - |\widetilde{H}| = |\{u\} \cup UN_M(u)|$ times, which is $O(\deg(u))$ where $\deg(u)$ is the degree of query vertex $u$. As a result, the time complexity of Algorithm 4 is $O(\deg(u) \times |E(B_{M'})|)$.

Most invocations of GetMaxiumMatching belong to the second case where $H_{init} \neq \emptyset$, because the first case applies to only when partial embedding $M$ is $\emptyset$ at the beginning of Algorithm 1. As a result, we efficiently obtain the maximum matching by taking advantage of the second case.

## 5 PRUNING BY FAILING SETS WITH BIPARTITE MATCHING

In this section, we propose a new method to additionally prune out unnecessary search space by employing two pruning methods: pruning by bipartite matching and pruning by *failing sets* [14]. In DAF [14], a failing set is introduced as a concept to avoid redundant search space generated due to either a conflict of mapping or an empty set of extendable candidates. We aim to apply the idea of pruning by bipartite matching in computing failing sets. For this, we introduce a new method to compute failing sets in the search tree pruned by bipartite matching. This approach in fact generalizes the failing set computation method [14]. Consequently, taking full advantage of two different pruning techniques can reduce a lot of unnecessary search space.



**(a) Query graph $q$.**

**(b) Data graph G.**

**Figure 5: A query graph $q$ and a data graph $G$**

**Example 5.1.** Consider query graph $q$, data graph $G$ in Figure 5, and CS constructed over them in Figure 7a as a new running example. Figure 6 is a search tree for this query and the CS. Assume that we just came back to the partial embedding $M$ after visiting partial embeddings $M_1$ and $M_2$ in the exploration of the subtree rooted at $M$. A node "$(u, v) \times$ " such as $M_1$ and $M_2$ in a search tree denotes that $(u, v)$ is pruned out by bipartite matching in Section 4, which subsumes every mapping conflict such as $M_1$. Figure 7b and Figure 7d are candidate bipartite graphs on $M_1$ and $M_2$, respectively, with maximum bipartite matchings (red). Note that $u_4$ has nothing related to the fact that the size of the maximum bipartite matchings is smaller than $|V(q)|$. Thus, updating the mapping of $u_4$ from $v_{13}$ to other extendable candidates (e.g., $v_{23}$) of $u_4$ still does not lead to an embedding. As a result, all the siblings of $M$ will be pruned out, which can reduce the search space.

**Definition 5.1. (Dynamic DAG)** Given a query graph $q$ and a partial embedding $M$, we define a *dynamic DAG* $d_M = (V(d_M), E(d_M))$ of $q$ regarding $M$ as follows.

- $V(d_M)$ is the set of either query vertices mapped in $M$ or neighbors of the mapped query vertices.
- For each undirected edge $(x, y) \in E(q)$, there is a directed edge $(x, y) \in E(d_M)$ if and only if
  (1) both $x$ and $y$ are mapped in $M$, and $x$ is mapped earlier than $y$, or
  (2) $x$ is mapped in $M$, and $y$ is unmapped in $M$.

**Definition 5.2. (Ancestor-Closed Set)** Given a dynamic DAG $d_M$, we say that a set of query vertices $S \subseteq V(d_M)$ is ancestor-closed in $d_M$ if for any $u \in S$, all the ancestors of $u$ in $d_M$ are also in $S$.

For a set of query vertices $S \subseteq V(d_M)$ and a node $M$ in the search tree, let $M[S]$ denote the largest subset of $M$ that is a partial embedding of $q[S]$, i.e., $(u, v) \in M[S]$ if and only if $(u, v) \in M$ and $u \in S$.

**Definition 5.3. (Failing Set)** We define a failing set $F_M \subset V(q)$ of node $M$ as an ancestor-closed set satisfying the following failure property: given a partial embedding $M[F_M]$, there exists no embedding of $q[F_M]$ which is extensions of $M[F_M]$.

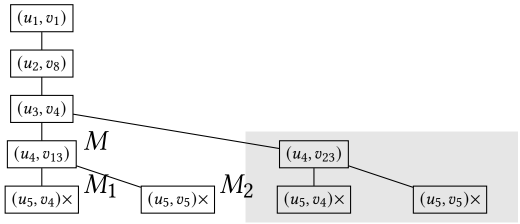


**Figure 6: Search tree for $q$ in Figure 5 and CS on $q$ and $G$ in Figure 5**

**Lemma 5.1.** [14] Let $M$ be a node in the search tree whose last mapping is $(u, v)$ in corresponding partial embedding, and let $F_M$ be a non-empty failing set of node $M$. If $u \notin F_M$, then all siblings of node $M$ cannot lead to an embedding of $q$.

By Lemma 5.1, we prune out a partial embedding $M$ if the failing set $F_M$ does not contain $u$ in the latest mapping in $M$.

**Failing Set Computation.** Since we traverse the search space in the DFS order, a failing set $F_M$ for each node can be well defined in the bottom-up fashion. We compute a failing set for each node $M$ in a search tree according to the following cases:

(A) Get a failing set of every leaf node $M$ as follows.
  (i) A leaf $M$ belongs to *embedding class* if $M$ is a full embedding of $q$. Then $F_M = \emptyset$.
  (ii) A leaf $M$ belongs to *non-injective class* if $|H| < |V(q)|$, where $H$ is the maximum bipartite matching in the candidate bipartite graph $B_M$ on $M$, i.e., $M$ is pruned by bipartite matching. We will describe a way to compute failing sets of this class.

(B) For every non-leaf node $M$, suppose that a node $M$ has $k$ children $M_1, M_2, ..., M_k$, which are all extensions of $M$ to the next vertex $u_n$, i.e, $M_i = M \cup \{(u_n, v_i)\}$. Assume that we have computed failing sets $F_{M_1}, ..., F_{M_k}$ of $M_1, ..., M_k$ respectively. If there exists a child node $M_i$ such that $u_n \notin F_{M_i}$, we set $F_M = F_{M_i}$; otherwise, $F_M = \cup_{i=1}^{k} F_{M_i}$.

A node in a search tree can be classified as one of three categories: a leaf in *embedding class*, a leaf in *non-injective class*, and a non-leaf node. Unlike DAF [14], there is neither *conflict class* nor *empty-set class*, because non-injective class includes these classes.

**Definition 5.4. (Non-injective Connected Component)** Suppose that we have a candidate bipartite graph $B_M$ on a given partial embedding $M$. Let $g' = (V(g'), E(g'))$ be a connected component in $B_M$. A connected component $g'$ is called *non-injective connected component* if $|H'| < |V(g') \cap V(q)|$ where $H'$ is the maximum bipartite matching in $g'$.

A non-injective connected component is the induced subgraph of the vertices in that component. Thus, let $B_M[V(g')]$ denote the non-injective connected component $g' = (V(g'), E(g'))$ in $B_M$.

**Example 5.2.** Figure 7a and Figure 7c demonstrate blue-colored extendable candidates and gray-colored candidates already mapped in partial embeddings $M_1$ and $M_2$, respectively. Figure 7b and Figure 7d show candidate bipartite graphs $B_{M_1}$ and $B_{M_2}$, respectively, generated from the extendable candidates and the mapped candidates. In these bipartite graphs, connected components surrounded by purple boundaries are non-injective connected components, i.e., $B_{M_1}[V_1]$ and $B_{M_2}[V_2]$ are non-injective connected components where $V_1 = \{u_3, u_5, v_4\}$ and $V_2 = \{u_2, u_7, u_9, v_8, v_9\}$.

**Lemma 5.2.** Given a candidate bipartite graph $B_M$ on a partial embedding $M$ of query graph $q$, there exists at least one non-injective connected component in $B_M$ if and only if $|H| < |V(q)|$ where $H$ is the maximum bipartite matching in $B_M$.

By Lemma 5.2, we can obtain a non-injective connected component $g' = (V(g'), E(g'))$ in candidate bipartite graph $B_M$. We set a failing set as follows:

$$F_M = \cup_{x \in V(g') \cap V(d_M)} anc_M(x) \quad (1)$$

where $anc_M(x)$ denotes the set of all ancestors of $x$ in $d_M$ including $x$ itself.

**Lemma 5.3.** For a partial embedding $M$ that belongs to non-injective class, $F_M$ satisfies the failure property.

**Example 5.3.** Partial embeddings $M_1$ and $M_2$ in Figure 6 are pruned by bipartite matching. By Equation (1), failing sets of $M_1$ and $M_2$ are $F_{M_1} = \{u_1, u_3, u_5\}$ and $F_{M_2} = \{u_1, u_2, u_3, u_5, u_7, u_9\}$, respectively. For partial embedding $M$, $u_5 \in F_{M_1}$ and $u_5 \in F_{M_2}$, and thus $F_M = F_{M_1} \cup F_{M_2}$ according to the failing set computation of **Case (B)**, i.e., $F_M = \{u_1, u_2, u_3, u_5, u_7, u_9\}$. Since $u_4 \notin F_M$, all siblings are redundant. Therefore, we prune out the subtrees rooted at $M$'s siblings enclosed by a gray box in Figure 6.

## 6 CELL-WIDE VERIFICATION

In this section, we introduce a new technique that can reduce the search space of backtracking based on the observation that candidate vertices with the same neighbors in CS lead to similar subtrees of a search tree. For some query vertices, candidates that have the same neighbors in CS result in the same extendable candidates, which may generate the similar subtrees when we map those query vertices to the extendable candidates.

**Example 6.1.** Figure 8a shows CS obtained from query graph $q$ in Figure 2a and data graph $G_1$ in Figure 2b. Figure 8b demonstrates a part of search tree of backtracking in Figure 8a. Here, $v_3, v_4, v_5 \in C(u_2)$ have the same neighbors in the CS. Mapping $u_2$ to $v_3, v_4$, or $v_5$ in $C(u_2)$ leads to partial embeddings $M_a$, $M_b$, and $M_c$ respectively. Three subtrees rooted at $M_a$, $M_b$ and $M_c$ are very similar.

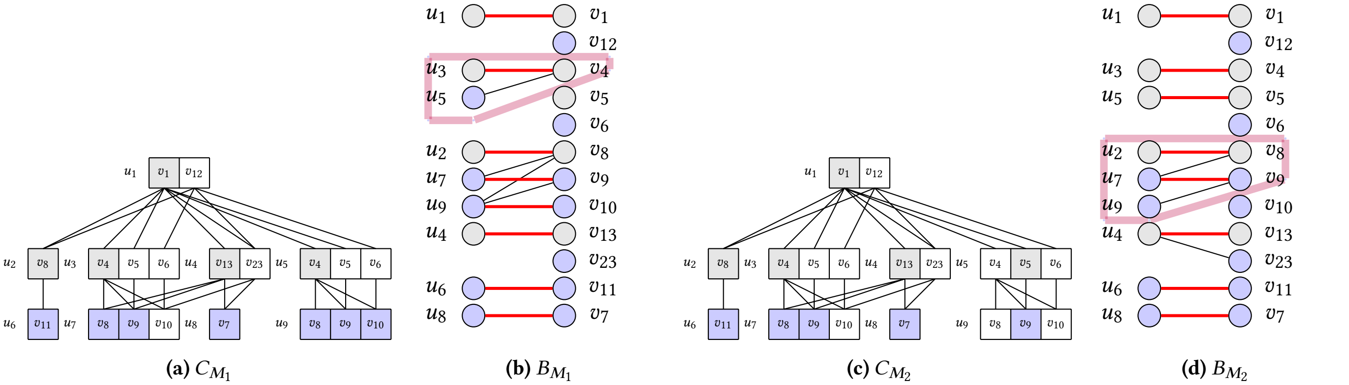
**Definition 6.1. (Cell)** Given a candidate set $C(u)$ of query vertex $u$ and a candidate $v \in C(u)$, *cell* $\gamma(u, v)$ is a subset of $C(u)$ such that:

- $w \in \gamma(u, v)$ if and only if $v$ and $w$ have the same set of neighbors in CS.

**Example 6.2.** The cells with size more than 1 are illustrated as blue and red boxes in Figure 8a. Candidates $v_3, v_4$, and $v_5$ in $C(u_2)$ have $v_1 \in C(u_1)$ and $v_{11} \in C(u_4)$ as neighbors in common (i.e., $\gamma(u_2, v_3)$ = $\gamma(u_2, v_4)$ = $\gamma(u_2, v_5)$). Both $v_{10}$ and $v_4$ in $C(u_5)$ are adjacent to only $v_7 \in C(u_3)$ and $v_9 \in C(u_6)$ (i.e., $\gamma(u_5, v_4) = \gamma(u_5, v_{10})$).

Since the candidates in a cell have the same neighbors in common, the similar subtrees are generated no matter which candidate in the cell is mapped in backtracking. Based on this phenomenon, we map each query vertex to a *cell* when we extend a partial mapping, and then every query vertex is eventually mapped to each candidate in the cell (which has already been mapped to that query vertex) when no extension is needed. This technique may considerably reduce the search space.

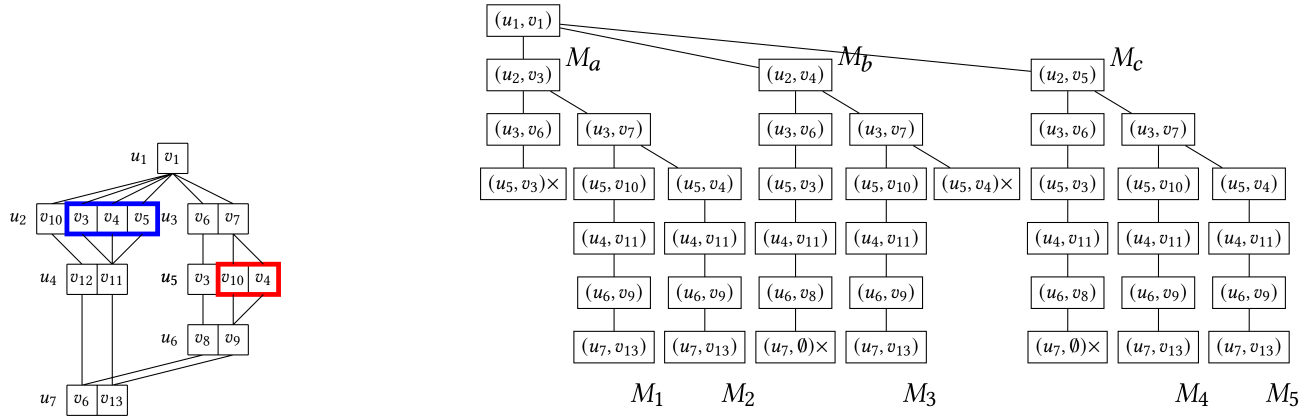
**Definition 6.2. (Hypermapping)** Given a query graph $q$ and a data graph $G$, *(full) hypermapping* $\mathcal{M}$ is a mapping $\mathcal{M} : V(q) \rightarrow 2^{V(G)}$ such that every $u \in V(q)$ is mapped to one of cells in $C(u)$.



$u_1$ $v_1$ $v_{12}$
$u_2$ $v_8$ $u_3$ $v_4$ $v_5$ $v_6$ $u_4$ $v_{13}$ $v_{23}$ $u_5$ $v_4$ $v_5$ $v_6$
$u_6$ $v_{11}$ $u_7$ $v_8$ $v_9$ $v_{10}$ $u_8$ $v_7$ $u_9$ $v_8$ $v_9$ $v_{10}$
$u_1$ $v_1$
$v_{12}$
$u_3$ $v_4$
$u_5$ $v_5$
$v_6$
$u_2$ $v_8$
$u_7$ $v_9$
$u_9$ $v_{10}$
$u_4$ $v_{13}$
$v_{23}$
$u_6$ $v_{11}$
$u_8$ $v_7$

(a) $C_{M_1}$ (b) $B_{M_1}$ (c) $C_{M_2}$ (d) $B_{M_2}$

**Figure 7: Sets of extendable candidates, and candidate bipartite graphs**



$u_1$ $v_1$
$u_2$ $v_{10}$ $v_3$ $v_4$ $v_5$ $u_3$ $v_6$ $v_7$
$u_4$ $v_{12}$ $v_{11}$ $u_5$ $v_3$ $v_{10}$ $v_4$
$u_6$ $v_8$ $v_9$
$u_7$ $v_6$ $v_{13}$
$(u_1, v_1)$
$M_a$ $M_b$ $M_c$
$(u_2, v_3)$ $(u_2, v_4)$ $(u_2, v_5)$
$(u_3, v_6)$ $(u_3, v_7)$ $(u_3, v_6)$ $(u_3, v_7)$ $(u_3, v_6)$ $(u_3, v_7)$
$(u_5, v_3)\times$ $(u_5, v_{10})$ $(u_5, v_4)$ $(u_5, v_3)$ $(u_5, v_{10})$ $(u_5, v_4)\times$ $(u_5, v_3)$ $(u_5, v_{10})$ $(u_5, v_4)$
$(u_4, v_{11})$ $(u_4, v_{11})$ $(u_4, v_{11})$ $(u_4, v_{11})$ $(u_4, v_{11})$ $(u_4, v_{11})$ $(u_4, v_{11})$
$(u_6, v_9)$ $(u_6, v_9)$ $(u_6, v_8)$ $(u_6, v_9)$ $(u_6, v_8)$ $(u_6, v_9)$ $(u_6, v_9)$
$(u_7, v_{13})$ $(u_7, v_{13})$ $(u_7, \emptyset)\times$ $(u_7, v_{13})$ $(u_7, \emptyset)\times$ $(u_7, v_{13})$ $(u_7, v_{13})$
$M_1$ $M_2$ $M_3$ $M_4$ $M_5$

(a) CS with cells (b) Search tree

**Figure 8: CS and search tree**

Given a subset $S \subset V(q)$, a hypermapping of an induced subgraph $q[S]$ is called a *partial hypermapping*.

**Definition 6.3. (Extendable Candidates for Partial Hypermapping)** Suppose that we have a partial hypermapping $\mathcal{M}$ and an unmapped query vertex $u$. Let $p_1, p_2, ..., p_k$ be the parents of $u$ in $q_d$ already mapped in $\mathcal{M}$, then $C_{\mathcal{M}}(u)$ is defined as $\cap_i^k N_u^{p_i}(\mathcal{M}(p_i))$, where $N_{u_c}^u(\gamma(u, v))$ is $N_{u_c}^u(x)$ for any candidate vertex $x \in \gamma(u, v)$.

**Example 6.3.** Search tree in Figure 9a shows extending a partial hypermapping to find hypermappings in the CS of Figure 8a. Figure 9b illustrates CS that corresponds to the partial hypermapping $\mathcal{M}_1$ in the search tree where gray-colored candidate vertices compose every cell mapped in $\mathcal{M}_1$, and extendable candidates for $\mathcal{M}_1$ are blue, e.g., $C_{\mathcal{M}_1}(u_4) = \{v_{11}\}$.

In our backtracking approach, we map a vertex in a query graph $q$ to a candidate cell. Suppose that we are trying to extend a partial hypermapping $\mathcal{M}$. We select an extendable vertex $u$ from unmapped query vertices, and then we extend a partial hypermapping $\mathcal{M}$ to new hypermapping $\mathcal{M}'$ by mapping $u$ to one of cells in $C_{\mathcal{M}}(u)$. We compute also the *candidate bipartite graph* $B_{\mathcal{M}'}$ *on* $\mathcal{M}'$. We backtrack if one of following two conditions (1) and (2) is satisfied.

(1) $\mathcal{M}'$ is a full hypermapping.

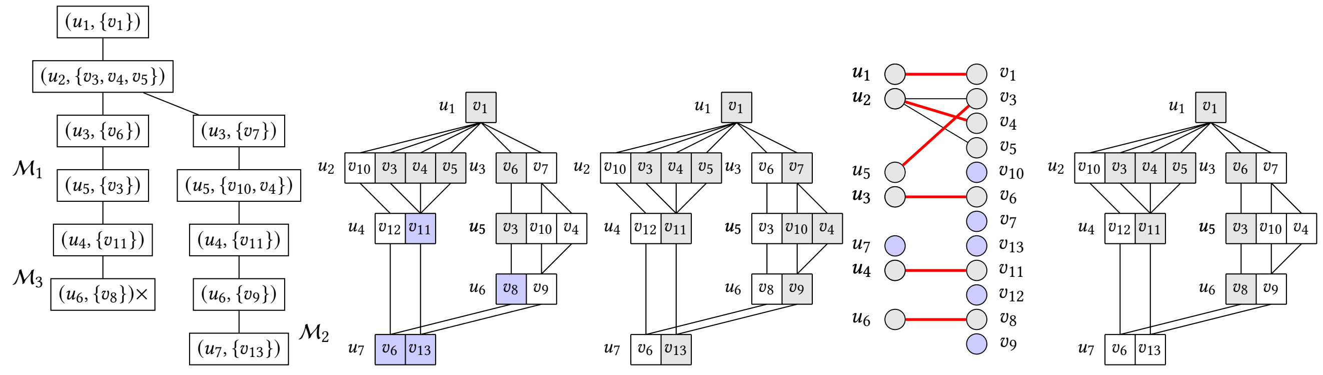
(2) $|H'| < |V(q)|$ ($H'$ is the maximum bipartite matching in $B_{\mathcal{M}'}$).

**Suppose that $\mathcal{M}'$ is a full hypermapping (Condition (1)).** Then we try to map each query vertex $u$ to every possible candidate vertex in the cell mapped to $u$ in $\mathcal{M}'$.

**Lemma 6.1.** For $n$ sets $X_1, X_2, ..., X_n$, let $\Pi_{i \in \{1, \cdots, n\}} X_i$ denote the Cartesian product $X_1 \times X_2 \times \cdots \times X_n$ over these sets. Suppose that a query graph $q$, a data graph $G$, and a full hypermapping $\mathcal{M}$ are given. If $(u_i, u_j) \in E(q)$, then for every $v_x \in \mathcal{M}(u_i)$ and every $v_y \in \mathcal{M}(u_j)$, $(v_x, v_y) \in E(G)$. Therefore, $\Pi_{u \in V(q)} \{(u, v) \mid v \in \mathcal{M}(u)\}$ is the set of homomorphisms of $q$ in $G$.

By Lemma 6.1, we find homomorphisms of $q$ in $G$ by computing the Cartesian product of candidate vertices in each cell of a full hypermapping $\mathcal{M}'$. A query vertex may be mapped to two or more distinct candidate vertices in a homomorphism. All injective mappings among these homomorphisms are embeddings of $q$ in $G$.

**Example 6.4.** Suppose that we obtained a full hypermapping $\mathcal{M}_2$ in Figure 9a. Cells mapped in $\mathcal{M}_2$ are illustrated as gray boxes in CS of Figure 9c. In this CS, all query vertices except $u_2$ and $u_5$ are mapped to cells which contain only one candidate. The combination of each candidate in $C_{\mathcal{M}_2}(u_2) = \{v_3, v_4, v_5\}$ and each candidate in $C_{\mathcal{M}_2}(u_5) = \{v_{10}, v_4\}$ will produce six different homomorphisms, i.e., $\Pi_{u \in V(q)} \{(u, v_i) \mid v_i \in \mathcal{M}_2(u)\}$. Among them, mapping $u_2$ to $v_4$ and mapping $u_5$ to $v_4$ do not lead to an injective mapping, so



**(a) Search tree** **(b) $C_{\mathcal{M}_1}$** **(c) $C_{\mathcal{M}_2}$** **(d) $B_{\mathcal{M}_3}$** **(e) $C_{\mathcal{M}_3}$**

**Figure 9: Extendable candidates, search tree, and extended candidate bipartite graph on partial hypermapping**

the remaining five homomorphisms are embeddings (equivalent to $M_1, M_2, M_3, M_4$, and $M_5$ in the search tree of Figure 8b).

**Suppose that $|H'| < |V(q)|$ where $H'$ is the maximum bipartite matching in $B_{\mathcal{M}'}$ (Condition (2)).** We define a candidate bipartite graph for a partial hypermapping in order to apply pruning by bipartite matching in Section 4.

**Definition 6.4. (Candidate Bipartite Graph on Partial Hypermapping)** Given a query graph $q$, a data graph $G$, a partial hypermapping $\mathcal{M}$, and extendable candidates sets $C_{\mathcal{M}}$. $B_{\mathcal{M}} = (V(q), V(G), E(B_{\mathcal{M}}))$ is defined as follows.

- For $u \in V(q)$ mapped in $\mathcal{M}$, there is an edge $(u, v) \in E(B_{\mathcal{M}})$ if and only if $v \in \mathcal{M}(u)$.
- For $u \in V(q)$ not mapped in $\mathcal{M}$, there is an edge $(u, v) \in E(B_{\mathcal{M}})$ if and only if $v \in C_{\mathcal{M}}(u)$.

**Example 6.5.** Figure 9e shows the gray-colored candidate vertices which compose the cells mapped in $\mathcal{M}_3$. In this CS, $C_{\mathcal{M}_3}(u_7) = \emptyset$. Figure 9d demonstrates the candidate bipartite graph $B_{\mathcal{M}_3}$ on $\mathcal{M}_3$ with the red-colored maximum bipartite matching. In this bipartite graph, $u_7$ has no incident edges as $C_{\mathcal{M}_3}(u_7) = \emptyset$. Since the size of this maximum bipartite matching is 6 which is less than $|V(q)| = 7$, the partial mapping $\mathcal{M}_3$ is pruned out.

**Algorithm 5:** CellWideBacktrack$_m(q, CS, \mathcal{M}, H)$

```
1  if $|H| < |V(q)|$ then
2      return;
3  else if $|\mathcal{M}| = |V(q)|$ then
4      Report all embeddings in $\Pi_{u \in V(q)}\{(u, v) \mid v \in \mathcal{M}(u)\}$;
5  else
6      Select a next extendable vertex $u$;
7      foreach $c \in \{\gamma(u, v) \mid v \in C_{\mathcal{M}}(u)\}$ do
8          $\mathcal{M}' \leftarrow \mathcal{M} \cup \{(u, c)\}$;
9          $\widetilde{H} \leftarrow \{(s, t) \in H \mid s \notin \{u\} \cup UN_M(u)\}$;
10         $H' \leftarrow$ GetMaximumMatching$(B_{\mathcal{M}'}, \widetilde{H})$;
11         CellWideBacktrack$_m(q, CS, \mathcal{M}', H')$;
```

**Search Process**. Given a partial hypermapping $\mathcal{M}$ and the maximum bipartite matching $H$ of the bipartite graph $B_{\mathcal{M}}$, Algorithm 5 applies Conditions (1) and (2) of backtracking. If $|H| < |V(q)|$, it backtracks based on Condition (2), i.e., pruning by bipartite matching) (lines 1-2). If $|\mathcal{M}| = |V(q_d)|$, we report all injective mappings from $\Pi_{u \in V(q_d)}\{(u, v_i) \mid v_i \in \mathcal{M}(u)\}$ (lines 3-4). (For subgraph search, we replace line 4 by returning the data graph and immediately terminating this procedure if there exists at least one injective mapping in the Cartesian product.) Otherwise, we select a next extendable vertex $u$ (line 6). Next, we iterate over every distinct cell $c$ in $\{\gamma(u, v) \mid v \in C_{\mathcal{M}}(u)\}$, which is the set of cells that extendable candidates in $C_{\mathcal{M}}(u)$ belong to. For each cell $c$, we extend $\mathcal{M}$ to $\mathcal{M}' = \mathcal{M} \cup \{(u, c)\}$ (lines 7-8). Then a new maximum matching $H'$ of $B_{\mathcal{M}'}$ is computed, and CellWideBacktrack$_m$ is recursively invoked with arguments $\mathcal{M}'$ and $H'$ (lines 9-11). Finally, we replace Backtrack$_m$ with CellWideBacktrack$_m$ in line 5 of Algorithm 1.

To discriminate between the verification using hypermappings and the existing verification approach that maps a query vertex to a candidate vertex, we refer to them as *cell-wide verification* and *vertex-wide verification*, respectively.

We add failing sets into the implementation of the cell-wide verification. Failing sets in the cell-wide verification are computed exactly the same as those in the vertex-wide verification except that: (1) an embedding is replaced by a hypermapping in the embedding class, and (2) $F_{\mathcal{M}}$ of a hypermapping $M$ in the non-injective class is obtained from $\cup_{x \in V(g') \cap V(q)}\ anc(x)$ where $g' = (V(g'), E(g'))$ is the non-injective connective component in candidate bipartite graph $B_{\mathcal{M}}$ on a partial hypermapping $\mathcal{M}$.

## 7 PERFORMANCE EVALUATION

We evaluate the performance of the competing algorithms for subgraph search and subgraph matching. All the source codes were obtained from the authors of previous papers, and they are implemented in C++. Experiments are conducted on a Linux machine with two Intel Xeon E5-2680 v3 2.5GHz CPUs and 256GB memory.

**Metrics.** We measure the query processing time which is the sum of filtering time and verification time for subgraph search, or the sum of preprocessing time (i.e., time to construct an auxiliary data structure) and search time (i.e., time to enumerate the first $10^5$ embeddings) for subgraph matching. We set a time limit of 10 minutes for each query. If an algorithm does not process a query within the time limit, we regard the processing time of the query as

10 minutes. We say that the query finished within the time limit is *solved.* Each query set consists of 100 query graphs. For each query set, we measure the average of query processing time to process query graphs solved by at least one of the competing algorithms.

## 7.1 Subgraph Search

Since CFQL [42] and VEQ [21, 22] significantly outperformed existing subgraph search algorithms, we compare our subgraph search algorithm $\mathsf{BICE_S}$ with these two algorithms.

**Table 2: Characteristics of real-world datasets for subgraph search where $\Sigma$ is a set of distinct vertex labels**

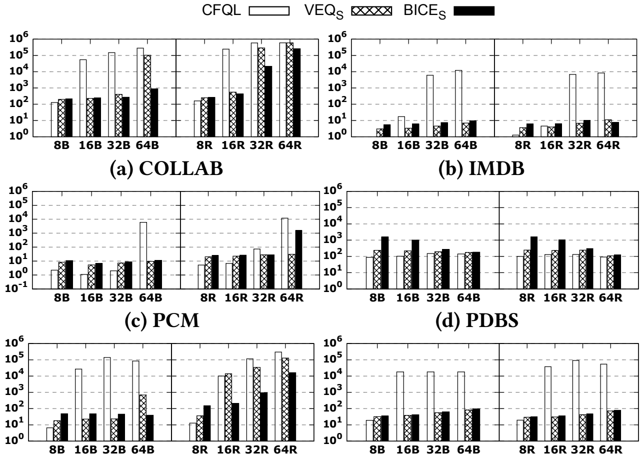
| | Dataset | | Average per graph | | | |
|---|---|---|---|---|---|---|
| | $\lvert\mathcal{D}\rvert$ | $\lvert\Sigma\rvert$ | $\lvert V(G)\rvert$ | $\lvert E(G)\rvert$ | degree | $\lvert\Sigma\rvert$ |
| COLLAB | 5,000 | 10 | 74 | 2,457 | 65.97 | 9.9 |
| IMDB | 1,500 | 10 | 13 | 66 | 10.14 | 6.9 |
| PCM | 200 | 21 | 377 | 4,340 | 23.01 | 18.9 |
| PDBS | 600 | 10 | 2,939 | 3,064 | 2.06 | 6.4 |
| PPI | 20 | 46 | 4,942 | 26,667 | 10.87 | 28.5 |
| REDDIT | 4,999 | 10 | 509 | 595 | 2.34 | 10.0 |

**Real Datasets.** Experiments are conducted on real-world datasets, which are PDBS, PCM, PPI used in [13, 20, 42], IMDB, REDDIT, and COLLAB provided by [48]. PDBS is a set of graphs that represent DNA, RNA, and proteins. PCM is a set of protein contact maps of amino acids. PPI is a database of protein-protein interaction networks. IMDB is a movie collaboration dataset. REDDIT is a dataset of online discussion communities, and COLLAB is a scientific collaboration dataset. As no label information is available for IMDB, REDDIT, and COLLAB, we randomly assigned a label out of 10 distinct labels to each vertex. The characteristics of the datasets are summarized in Table 2.

**Query Sets.** We adopt two query generation methods similar to those in previous studies, which are random walk [20, 42] and breadth first search (BFS) [42, 45]. For each dataset $\mathcal{D}$, we generate eight query sets $Q_{iR}$ (i.e., random walk) and $Q_{iB}$ (i.e., BFS) where $i \in \{8, 16, 32, 64\}$ is the number of edges of a query graph. A query graph is generated by the random walk method as follows: (1) select a vertex uniformly at random from a randomly selected graph $G \in \mathcal{D}$; (2) perform a random walk from the selected vertex until we visit $i$ distinct edges, from which we extract a subgraph with these edges. In the BFS method, we perform a BFS from the selected vertex until we visit $i$ distinct edges.

**Query Processing Time.** Figure 10 shows the mean query processing time of the algorithms. In general, the query processing time increases as the number of vertices increases. $\mathsf{BICE_s}$ outperforms $\mathsf{VEQ_s}$ and CFQL in most cases. $\mathsf{BICE_s}$ is up to two orders of magnitude faster than $\mathsf{VEQ_s}$ for COLLAB ($Q_{64B}$ in Figure 10a), and up to three orders of magnitude faster than CFQL for IMDB ($Q_{64B}$ in Figure 10b). $\mathsf{BICE_s}$ is marginally slower than $\mathsf{VEQ_s}$ and CFQL in PDBS, because embeddings of these queries can be easily found by all the algorithms, so the pruning techniques of $\mathsf{BICE_s}$ may incur an overhead.

**Sensitivity Analysis.** We evaluate the algorithms by varying several characteristics of a set $\mathcal{D}$ of data graphs. We generate each data graph $G \in \mathcal{D}$ by upscaling the smallest data graph of PPI (with 2008 edges) using Evograph [32], and assign labels to vertices based on a power law distribution. We vary the following parameters:

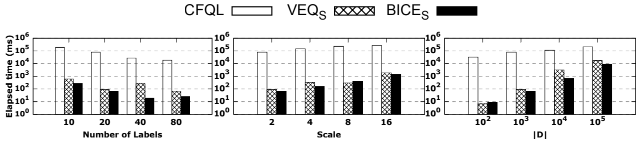


(a) COLLAB (b) IMDB

(c) PCM (d) PDBS

(e) PPI (f) REDDIT

**Figure 10: Query processing time (ms) of the subgraph search algorithms on real datasets**



(a) $\lvert\Sigma\rvert$ (b) Scale (c) $\lvert\mathcal{D}\rvert$

**Figure 11: Query processing time on synthetic datasets.**

- The number of distinct labels in $\Sigma$: 10, 20, 40, 80
- A scaling factor $s$ of a data graph in $\mathcal{D}$: 2, 4, 8, 16
- The number of data graphs in $\mathcal{D}$: $10^2$, $10^3$, $10^4$, $10^5$

where $s$ indicates that $\lvert E(G)\rvert$ is $s$ times larger than that of the input data graph while Evograph keeps the same statistical properties of $G$ by increasing $\lvert V(G)\rvert$ accordingly. Similarly to the existing work [20, 42], we set $\lvert\Sigma\rvert = 20$, $s = 2$, and $\lvert\mathcal{D}\rvert = 10^3$ as default; in fact, we choose $s = 2$ so that the default $\lvert V(G)\rvert$ corresponding to $s = 2$ is larger than that of the existing work for stress testing. If not specified, the parameters are set to their default values. We use query sets $Q_{16}$ which is the union of $Q_{16B}$ and $Q_{16R}$.

The query processing time of the algorithms on the synthetic datasets is shown in Figure 11. In most cases, the order of the algorithms from the fastest to slowest is $\mathsf{BICE_S}$, $\mathsf{VEQ_S}$, and CFQL. In Figure 11a, the query processing time of $\mathsf{BICE_S}$ decreases as the number of distinct labels increases because increasing number of labels make the size of candidates space smaller, which reduces the size of search space. In Figure 11b and Figure 11c, $\mathsf{BICE_S}$ generally outperforms $\mathsf{VEQ_S}$ and CFQL. In Figure 11b, the query processing time rises as the data graphs get larger since the time to verify a false positive data graph can dramatically increase. In Figure 11c, the time also rises as $\lvert\mathcal{D}\rvert$ increases, because more false positive answers may exponentially increase the verification time.

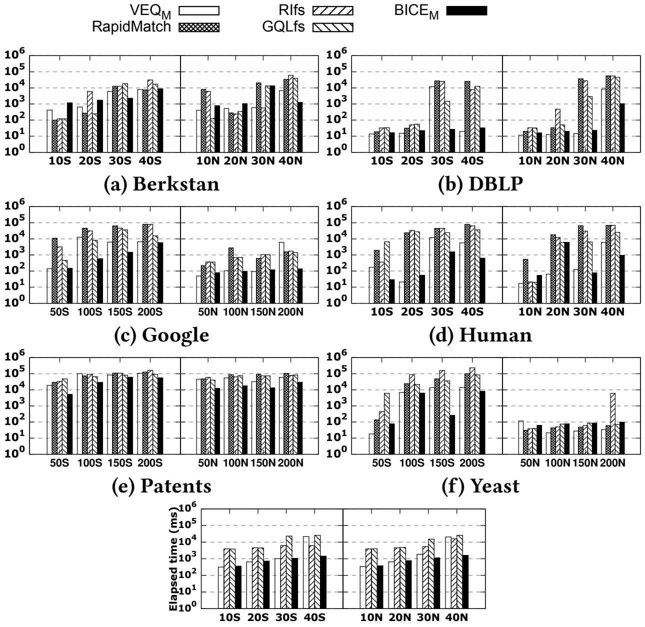
## 7.2 Subgraph Matching

To evaluate the performance of our subgraph matching algorithm $\mathsf{BICE_M}$, we compare it against existing subgraph matching algorithms $\mathsf{VEQ_M}$ [21, 22], RapidMatch [44], RIfs [43], and GQLfs [43].

**Datasets.** We test the algorithms against real-world datasets of Table 3 widely used in previous work [5, 14, 15, 28]. Yeast and

**Table 3: Characteristics of real datasets for subgraph matching where $\Sigma$ is a set of distinct vertex labels in $G$**

| $G$ | $\|V(G)\|$ | $\|E(G)\|$ | Avg degree | $\|\Sigma\|$ |
|---|---|---|---|---|
| Berkstan | 685,230 | 6,649,470 | 19.41 | 20 |
| DBLP | 317,080 | 1,049,866 | 6.62 | 20 |
| Google | 875,713 | 4,332,051 | 9.87 | 20 |
| Human | 4,674 | 86,282 | 36.91 | 44 |
| Patents | 3,774,768 | 16,518,948 | 8.75 | 20 |
| Yeast | 3,112 | 12,519 | 8.04 | 71 |
| Twitter | 41,652,230 | 1,468,364,884 | 70.51 | 1,000 |



(a) Berkstan

(b) DBLP

(c) Google

(d) Human

(e) Patents

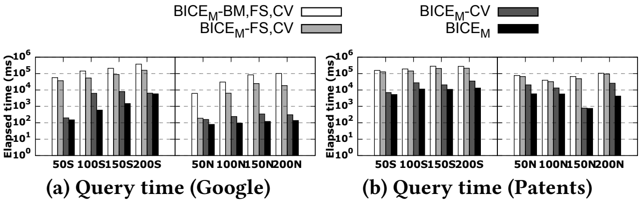
(f) Yeast

(g) Twitter

**Figure 12: Query processing time (ms) of the subgraph matching algorithms on real datasets**

Human are protein-protein interaction networks. Google, Berkstan, Patents and DBLP are obtained from Stanford Large Network Dataset Collection [29]. Google and Berkstan represent hyperlinks connecting webpages. Patents is a dataset of citations made by US patents. DBLP is a co-authorship network. As no label information is available for Google, Berkstan, Patents and DBLP, we randomly assigned a label out of 20 distinct labels to each vertex. We also tested our algorithm on the Twitter graph [27] with billions of edges. We randomly assigned 1000 distinct labels.

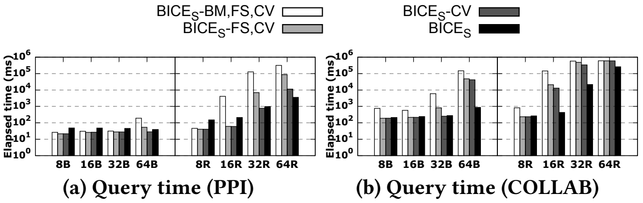
**Query Sets.** We use the same experimental setting as [5] and [14]. We generate sparse query sets $Q_{iS}$ and non-sparse query sets $Q_{iN}$ where $i$ is the number of vertices in a query graph such that $i \in \{50, 100, 150, 200\}$ for Yeast, Google, and Patents, and $i \in \{10, 20, 30, 40\}$ for the remaining datasets. Each query graph in $Q_{iS}$ and $Q_{iN}$ has the average degree $\leq 3$ and $> 3$, respectively. A query graph is generated as follows: (1) select a vertex uniformly at random, (2) perform a random walk on a data graph until we visit $i$ distinct vertices, and (3) extract a subgraph with the visited vertices and some edges between these vertices.



(a) Query time (Google)

(b) Query time (Patents)

**Figure 13: Performance gain of each technique in BICE$_M$**



(a) Query time (PPI)

(b) Query time (COLLAB)

**Figure 14: Performance gain of each technique in BICE$_S$**

**Query Processing Time.** Figure 12 shows the average query processing time of the algorithms. BICE$_M$ generally outperforms VEQ$_M$, which is followed by GQLfs, RIfs, and RapidMatch.

Overall, BICE$_M$ outperforms the others for DBLP, Google, Human, Patents, Yeast, and Twitter. Specifically, BICE$_M$ outperforms RapidMatch and RIfs up to three orders of magnitude for $Q_{30N}$ of DBLP in Figure 12b. In addition, BICE$_M$ outperforms GQLfs up to two orders of magnitude for $Q_{40S}$ of DBLP in Figure 12b, and $Q_{20S}$ of Human in Figure 12d. BICE$_M$ also outperforms VEQ$_M$ up to two orders of magnitude for $Q_{30S}$ of DBLP in Figure 12b. For Twitter, we set a time limit of 15 minutes for each query. We could not include RapidMatch, which spends at least 30 minutes for each query. RIfs and GQLfs cause a memory error for some queries in Twitter, so we exclude them from a query set for all the algorithms. BICE$_M$ is more than 10 times faster than the rest for $Q_{40N}$ in Figure 12g. Only BICE$_M$ solves every query within the time limit.

The query processing time of the other algorithms exponentially grows as the size of query graph increases in DBLP. On the contrary, BICE$_M$ takes nearly constant query processing time for DBLP in all query sets except $Q_{40N}$. We attribute the good performance of BICE$_M$ on DBLP, Google, Patents, and Yeast to the low average degree of these graphs. This may result in a small number of extendable candidates, which can speed up bipartite matching. Exceptionally, BICE$_M$ is marginally slower than the other algorithms in the non-sparse queries in Yeast, because these queries have small search space so our techniques may incur an overhead.

In Berkstan, BICE$_M$ outperforms the other algorithms for $Q_{30S}$ and $Q_{40N}$. Although BICE$_M$ makes fewer recursive calls than the other algorithms due to its effective pruning techniques, it is not the best performer for the remaining query sets. Note that Berkstan has a high average degree and a large number of vertices. These characteristics may lead to a large number of extendable candidates, which results in more time to find the maximum bipartite matching in a candidate bipartite graph.

## 7.3 Effectiveness of Individual Techniques

In this subsection, we run our algorithm and its variants below to measure the performance gain achieved by each technique of Section 4, Section 5, and Section 6.

**Table 4: Average ratio of the number of partial embeddings to the number of partial hypermappings**

| Query | Patents<br>50S | <br>100S | <br>150S | <br>200S | <br>50N | <br>100N | <br>150N | <br>200N |
|---|---|---|---|---|---|---|---|---|
| Ratio | 15.45 | 15.16 | 12.66 | 3.25 | 11.64 | 12.76 | 6.61 | 231.41 |
| Query | COLLAB<br>8B | <br>16B | <br>32B | <br>64B | <br>8R | <br>16R | <br>32R | <br>64R |
| Ratio | 1.00 | 1.00 | 1.49 | 52.26 | 1.07 | 100.33 | 352.69 | – |

- BICE$_M$-BM,FS,CV (or BICE$_S$-BM,FS,CV): a baseline for comparison using vertex-wide verification without pruning by bipartite matching and without failing sets.
- BICE$_M$-BM,CV (or BICE$_S$-BM,CV): using vertex-wide verification with pruning by only bipartite matching.
- BICE$_M$-CV (or BICE$_S$-CV): using vertex-wide verification with pruning by bipartite matching and pruning by failing sets.
- BICE$_M$ (or BICE$_S$): our algorithm using cell-wide verification with pruning by bipartite matching and pruning by failing sets.

Figure 13 shows the query processing time for subgraph matching in Google and Patents, and Figure 14 shows the query processing time for subgraph search in COLLAB and PPI.

**Effectiveness of Pruning by Bipartite Matching.** Overall, BICE$_M$-FS,CV outperforms BICE$_M$-BM,FS,CV with respect to the query processing time, and BICE$_S$-FS,CV outperforms BICE$_S$-BM,FS,CV with respect to the query processing time. BICE$_M$-BM,FS,CV is 33 times faster than BICE$_M$-FS,CV for Google ($Q_{50N}$ in Figure 13a).

**Effectiveness of Pruning by Failing Sets.** The effectiveness of our failing set computation method can be verified by the performance gap between BICE$_M$-FS,CV and BICE$_M$-CV in Figure 13 and the performance gap between BICE$_S$-FS,CV and BICE$_S$-CV in Figure 14. To be specific, BICE$_M$-CV is more than 185 times faster than BICE$_M$-FS,CV for Google ($Q_{50S}$ in Figure 13a).

**Effectiveness of Cell-Wide Verification.** In Figure 13, BICE$_M$ performs slightly better than BICE$_M$-CV with respect to the query processing time. Compared to BICE$_S$-CV, BICE$_S$ is more than one order of magnitude faster for COLLAB ($Q_{16R}$ and $Q_{64B}$ in Figure 14b). BICE$_S$ solves five queries in $Q_{64R}$ of COLLAB, whereas the others solve no queries within the time limit.

**Compression Power of Hypermapping.** To justify the effectiveness of the cell-wide verification, we measure the ratio of the number of nodes in a search tree generated by BICE$_M$-CV (or BICE$_S$-CV) to the number of nodes in a search tree generated by BICE$_M$ (or BICE$_S$) in Table 4. The ratio indicates the average number of partial embeddings covered by a partial hypermapping, so a larger ratio is better. The ratio increases as the size of a query graph grows. On average, each node in the search tree of cell-by-cell backtracking contains 231.41 and 352.69 partial embeddings for $Q_{200N}$ of Patents and for $Q_{32R}$ of COLLAB, respectively.

## 7.4 Complexity Analysis

**Table 5: Time complexities of the compared algorithms.**

| Subgraph matching for query graph $q$ in data graph $G$ | | |
|---|---|---|
| Algorithm | Preprocessing | Enumeration |
| RapidMatch | $O(\lvert E(q)\rvert \lvert E(G)\rvert)$ | Exponential to $\lvert V(q)\rvert$<br>(vertex-by-vertex mapping) |
| RIfs | - | |
| GQLfs | $O(\lvert V(q)\rvert \lvert E(G)\rvert)$ | |
| VEQ$_M$ | $O(\lvert E(q)\rvert \lvert E(G)\rvert)$ | |
| BICE$_M$ | $O(\lvert E(q)\rvert \lvert E(G)\rvert)$ | (cell-by-cell mapping) |

| Subgraph search for query $q$ in each $G \in \mathcal{D}$ | | |
|---|---|---|
| Algorithm | Filtering | Verification |
| CFQL | $O(\lvert E(q)\rvert \lvert E(G)\rvert)$ | Exponential to $\lvert V(q)\rvert$<br>(vertex-by-vertex mapping) |
| VEQ$_S$ | $O(\lvert E(q)\rvert \lvert E(G)\rvert)$ | |
| BICE$_S$ | $O(\lvert E(q)\rvert \lvert E(G)\rvert)$ | (cell-by-cell mapping) |

Table 5 describes the time complexity of each step for the algorithms. While RIfs has no preprocessing step to compute candidate sets, the others take polynomial time with respect to the sizes of $q$ and $G$ in preprocessing and cell computation. Both BICE and VEQ take $O(|E(q)||E(G)|)$ time for preprocessing, as BICE adopts the CS construction method of VEQ. After preprocessing, like VEQ, BICE spends $O(\Sigma_{u \in V(q)} \deg(u) \cdot d_{max}(G) \cdot |C(u)|)$ time in the worst case for the cell computation in CS. Specifically, for each query vertex $u$, we compute the cells via a divide-and-conquer approach over every candidate $v \in C(u)$: for each pivot candidate $v_n \in C(u_n)$ for a neighbor $u_n$ of $u$, they partition elements of $C(u)$ into two subsets according to whether each element $v \in C(u)$ is adjacent to the pivot in CS (refer to Lemma 4 in [22] for details). For each neighbor $u_n$ of $u$, there are at most $d_{max}(G)$ neighbors of $v \in C(u)$ in $C(u_n)$, where $d_{max}(G)$ is the maximum of degrees of all $v \in V(G)$. Therefore, the number of possible pivots is $O(\deg(u) \cdot d_{max}(G))$ for a fixed $u \in V(q)$ and a fixed $v \in C(u)$. Consequently, computing cells for all $u \in V(q)$ and all $v \in C(u)$ takes $O(\Sigma_{u \in V(q)} \deg(u) \cdot d_{max}(G) \cdot |C(u)|)$ time. Since subgraph matching and subgraph search are NP-hard, backtracking (in the enumeration step of subgraph matching or in the verification step of subgraph search) of all the algorithms takes the time exponential to $|V(q)|$ in the worst case. Nevertheless, BICE matches cell by cell unlike the rest that matches vertex by vertex, which empirically results in the performance gap between two different backtracking frameworks.

## 8 CONCLUSION

To speed up subgraph search and subgraph matching, we propose three methods that result in compact search space. We compare our approach with existing state-of-the-art algorithms in extensive experiments for various datasets. The experiments show that our algorithm significantly outperforms the competitors in the query processing time.

## ACKNOWLEDGMENTS

Park was 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). Kim was supported by the research fund of Hanyang University (HY-202100000003161). Kim was supported by Institute of Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01373, Artificial Intelligence Graduate School Program(Hanyang University)).

# REFERENCES

[1] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)* 42, 4 (2017), 1–44.

[2] Noga Alon, Phuong Dao, Iman Hajirasouliha, Fereydoun Hormozdiari, and S Cenk Sahinalp. 2008. Biomolecular network motif counting and discovery by color coding. *Bioinformatics* 24, 13 (2008), i241–i249.

[3] Anonyme Anonyme. 2014. *Subgraph Matching for Single Large Multigraphs Subgraph Matching for Single Large Multigraphs.* Ph.D. Dissertation. LIRMM.

[4] 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.

[5] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of ACM SIGMOD.* 1199–1214.

[6] Vincenzo Bonnici, Alfredo Ferro, Rosalba Giugno, Alfredo Pulvirenti, and Dennis Shasha. 2010. Enhancing graph database indexing by suffix tree structure. In *IAPR International Conference on Pattern Recognition in Bioinformatics.* Springer, 195–203.

[7] Mario Cannataro and Pietro H Guzzi. 2012. *Data management of protein interaction networks.* Vol. 17. John Wiley & Sons.

[8] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. 2017. Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with VF3. *IEEE transactions on pattern analysis and machine intelligence* 40, 4 (2017), 804–818.

[9] 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.

[10] Raffaele Di Natale, Alfredo Ferro, Rosalba Giugno, Misael Mongiovì, Alfredo Pulvirenti, and Dennis Shasha. 2010. Sing: Subgraph search in non-homogeneous graphs. *BMC bioinformatics* 11, 1 (2010), 96.

[11] Wenfei Fan. 2012. Graph pattern matching revised for social network analysis. In *Proceedings of ICDT.* 8–21.

[12] Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co.

[13] Rosalba Giugno, Vincenzo Bonnici, Nicola Bombieri, Alfredo Pulvirenti, Alfredo Ferro, and Dennis Shasha. 2013. Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. *PloS one* 8, 10 (2013), e76911.

[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 ACM SIGMOD.* 1429–1446.

[15] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turbo iso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *Proceedings of ACM SIGMOD.* 337–348.

[16] Wook-Shin Han, Jinsoo Lee, Minh-Duc Pham, and Jeffrey Xu Yu. 2010. iGraph: a framework for comparisons of disk-based graph indexing techniques. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 449–459.

[17] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. In *Proceedings of ACM SIGMOD.* 405–418.

[18] John E Hopcroft and Richard M Karp. 1973. An nˆ5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing* 2, 4 (1973), 225–231.

[19] 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.

[20] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. 2015. Performance and scalability of indexed subgraph query processing methods. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1566–1577.

[21] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile Equivalences: Speeding up Subgraph Query Processing and Subgraph Matching. In *Proceedings of ACM SIGMOD.* 925–937.

[22] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2022. Fast subgraph query processing and subgraph matching via static and dynamic equivalences. *The VLDB Journal* (2022), 1–26.

[23] Hyunjoon Kim, Seunghwan Min, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2020. IDAR: Fast supergraph search using DAG integration. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1456–1468.

[24] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. 2015. Taming Subgraph Isomorphism for RDF Query Processing. *Proceedings of the VLDB Endowment* 8, 11 (2015).

[25] 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 ACM SIGMOD.* 411–426.

[26] Karsten Klein, Nils Kriege, and Petra Mutzel. 2011. CT-index: Fingerprint-based graph indexing combining cycles and trees. In *Proceedings of IEEE ICDE.* 1115–1126.

[27] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web.* 591–600.

[28] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *Proceedings of the VLDB Endowment* 6, 2 (2012), 133–144.

[29] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

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

[31] Noel M O'Boyle and Roger A Sayle. 2016. Comparing structural fingerprints using a literature-based similarity benchmark. *Journal of cheminformatics* 8, 1 (2016), 1–14.

[32] Himchan Park and Min-Soo Kim. 2018. EvoGraph: an effective and efficient graph upscaling method for preserving graph properties. In *Proceedings of ACM SIGKDD.* 2051–2059.

[33] Eric Plotnick. 1997. *Concept mapping: A graphical system for understanding the relationship between concepts.* ERIC Clearinghouse on Information and Technology Syracuse, NY.

[34] N Pržulj, Derek G Corneil, and Igor Jurisica. 2006. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics* 22, 8 (2006), 974–980.

[35] Syed Asad Rahman, Matthew Bashton, Gemma L Holliday, Rainer Schrader, and Janet M Thornton. 2009. Small molecule subgraph detector (SMSD) toolkit. *Journal of cheminformatics* 1, 1 (2009), 1–13.

[36] 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.

[37] Pedro Ribeiro, Pedro Paredes, Miguel EP Silva, David Aparicio, and Fernando Silva. 2021. A survey on subgraph counting: concepts, algorithms, and applications to network motifs and graphlets. *ACM Computing Surveys (CSUR)* 54, 2 (2021), 1–36.

[38] 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.

[39] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment* 11, 4 (2017), 420–431.

[40] 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.

[41] Tom AB Snijders, Philippa E Pattison, Garry L Robins, and Mark S Handcock. 2006. New specifications for exponential random graph models. *Sociological methodology* 36, 1 (2006), 99–153.

[42] Shixuan Sun and Qiong Luo. 2019. Scaling Up Subgraph Query Processing with Efficient Subgraph Matching. In *Proceedings of IEEE ICDE.* 220–231.

[43] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *Proceedings of ACM SIGMOD.* 1083–1098.

[44] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: a holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment* 14, 2 (2020), 176–188.

[45] Jing Wang, Nikos Ntarmos, and Peter Triantafillou. 2017. GraphCache: a caching system for graph queries. (2017), 13–24.

[46] Junhu Wang, Xuguang Ren, Shikha Anirban, and Xin-Wen Wu. 2019. Correct filtering for subgraph isomorphism search in compressed vertex-labeled graphs. *Information Sciences* 482 (2019), 363–373.

[47] Xifeng Yan, Philip S Yu, and Jiawei Han. 2004. Graph indexing: a frequent structure-based approach. In *Proceedings of ACM SIGMOD.* 335–346.

[48] Pinar Yanardag and SVN Vishwanathan. 2015. Deep graph kernels. In *Proceedings of ACM SIGKDD.* 1365–1374.

[49] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: Distance Index Based Subgraph Matching in Biological Networks. In *Proceedings of ACM EDBT.* 192–203.

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

[51] Peixiang Zhao, Jeffrey Xu Yu, and S Yu Philip. 2007. Graph indexing: Tree+ Delta>= Graph.. In *Proceedings of VLDB.* 938–949.

[52] Lei Zou, Lei Chen, Jeffrey Xu Yu, and Yansheng Lu. 2008. A novel spectral coding in a large graph database. In *Proceedings of EDBT.* 181–192.