Versatile Equivalences: Speeding up Subgraph Query Processing and Subgraph Matching

Hyunjoon Kim
Seoul National University
SAP Labs Korea
hjkim@theory.snu.ac.kr

Yunyoung Choi
Seoul National University
yychoi@theory.snu.ac.kr

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

Xuemin Lin
University of New South Wales
lxue@cse.unsw.edu.au

Seok-Hee Hong
University of Sydney
seokhee.hong@sydney.edu.au

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

ABSTRACT

Subgraph query processing (also known as subgraph search) and subgraph matching are fundamental graph problems in many application domains. A lot of efforts have been made to develop practical solutions for these problems. Despite the efforts, existing algorithms showed limited running time and scalability in dealing with large and/or many graphs. In this paper, we propose a new subgraph search algorithm using equivalences of vertices in order to reduce search space: (1) static equivalence of vertices in a query graph that leads to an efficient matching order of the vertices, and (2) dynamic equivalence of candidate vertices in a data graph, which enables us to capture and remove redundancies in search space. These techniques for subgraph search also lead to an improved algorithm for subgraph matching. Experiments show that our approach outperforms state-of-the-art subgraph search and subgraph matching algorithms by up to several orders of magnitude with respect to query processing time.

CCS CONCEPTS

• **Information systems → Information retrieval query processing**;

KEYWORDS

subgraph query processing; subgraph search; subgraph matching; vertex equivalence; neighbor-safety

ACM Reference Format:
Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile Equivalences: Speeding up Subgraph Query Processing and Subgraph Matching. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457265

*contact author

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457265

1 INTRODUCTION

Over the last several decades, a great deal of efforts have been made to develop practical solutions for NP-hard graph problems due to diverse graph data publicly available [32]. On the one hand, researchers have been motivated to develop scalable and efficient algorithms to analyze large graphs such as social networks and Resource Description Framework (RDF) data. One of the most famous problems for a large graph is *subgraph matching*. Given a data graph G and a query graph q, the subgraph matching problem is to find all matches of q in G. On the other hand, smaller graph data including protein-protein interaction (PPI) networks and chemical compounds have encouraged researchers to derive fast and scalable algorithms to deal with a large number of graphs. *Subgraph query processing* (also known as *subgraph search*) is a well-known problem for a collection of these graphs. Given a set D of data graphs and a query graph, subgraph search is to retrieve all the data graphs in D that contain q as subgraphs.

Both subgraph matching and subgraph search have a variety of real-world applications: social network analysis [9, 34], RDF query processing [17, 18], PPI network analysis [5, 28], and chemical compound search [40]. However, these problems are NP-hard because they include finding subgraph isomorphism which is an NP-hard problem. That is, solving these problems is the bottleneck of the applications.

Even though the two problems are closely related to each other, the research on each problem had been separately conducted until recently. Existing work on subgraph search [3, 8, 11, 19, 40, 44, 45] mainly adopted the *indexing-filtering-verification* strategy: (1) given a set D of data graphs, data structures are constructed from substructures (i.e., features) of data graphs in an indexing phase, (2) given a query graph q, the data graphs with a feature that does not contain q as a subgraph are filtered out for every feature in a filtering phase, and (3) a subgraph isomorphism test is performed against every remaining candidate graph in a verification phase. Meanwhile, the recent study on subgraph matching [2, 12, 13, 36] proposed algorithms based on a *preprocessing-enumeration* framework: an auxiliary data structure on a query graph and a data graph is constructed, and all matches of the query graph are found by using the data structure. These algorithms substantially improved

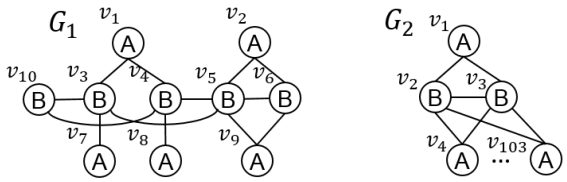


Figure 1: A set D of data graphs

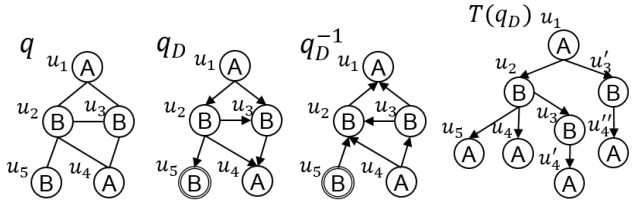


Figure 2: A query graph q, query DAGs q_D and q_D^{-1} built from q, and a path tree $T(q_D)$ of q_D

query processing performance. Researchers recently utilized existing subgraph matching algorithms to efficiently solve the subgraph search problem [35]. However, it showed limited response time and scalability in dealing with large query graphs or many data graphs.

In this paper, we introduce a new subgraph search algorithm $\mathsf{VEQ_S}$ employing static equivalence and dynamic equivalence in order to address the limitations. First, we apply neighbor equivalence of query vertices to the matching order of backtracking, which leads to a smaller search space in the verification phase. Second, we capture run-time equivalence of subtrees of the search space based on neighbor equivalence of candidate data vertices, and prune out redundancies (i.e., the equivalent subtrees) of the search space. Additionally, we propose an efficient filtering method called neighbor-safety that enables us to build a compact auxiliary data structure on a query graph and a data graph to obtain as few candidates as possible. We conduct extensive experiments on several well-known real datasets as well as synthetic datasets to compare our approach with existing algorithms. Moreover, our techniques for subgraph search in turn lead to an improved algorithm $\mathsf{VEQ_M}$ for subgraph matching. Experiments show that our approach outperforms existing subgraph search and subgraph matching algorithms by up to several orders of magnitude in terms of query processing time.

The rest of the paper is organized as follows. Section 2 provides definitions, problem statements and related work. Section 3 gives an overview of our approach. Section 4 introduces our filtering technique. Section 5 describes our query vertex matching order based on static equivalence, and Section 6 presents a new technique to detect and remove a part of search space by using dynamic equivalence. Section 7 presents an extensive experimental comparison with previous work, and Section 8 concludes the paper.

2 PRELIMINARIES

In this paper we focus on undirected and connected graphs with labeled vertices. Our techniques can be easily extended to directed or 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 $L_g : V(g) \rightarrow \Sigma$ that assigns a label to each vertex where Σ is a set of labels. For a subset S of $V(g)$, the *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 a graph $q = (V(q), E(q), L_q)$ and a graph $G = (V(G), E(G), L_G)$, an *embedding* of q in G is a mapping $M : V(q) \rightarrow V(G)$ 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 $u \in V(q)$, and (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*, i.e., it may not be injective. 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.

We will use the directed acyclic graph (DAG) as a tool to build an auxiliary data structure for q and G. Given a DAG g, a vertex is a root if it has no incoming edges, and a vertex is a leaf if it has no outgoing edges. A DAG g is a *rooted DAG* if there is only one root. Let Child(u) denote a set of vertices in $V(g)$ that have incoming edges from u. A *sub-DAG of g rooted at u*, denoted by g_u, is the induced subgraph of g whose vertices are u and all the descendants of u.

Table 1 lists the notations frequently used in the paper.

Table 1: Notations frequently used in this paper

Symbol	Definition
G	Data graph
q and q_D	Query graph and query DAG
A_q	Set of answer graphs for q
M	Partial embedding of q in G
$C(u)$	Set of candidate vertices of $u \in V(q)$
$C_M(u)$	Set of extendable candidates of u regarding M
$\pi(u, v)$	Cell of $v \in C(u)$ (defined in Section 6)
$\pi_M(u, v)$	Equivalence set of $v \in C(u)$ regarding M
$\mathcal{T}_M(u, v)$	Set of embeddings extended from $M \cup \{(u, v)\}$

2.1 Problem Statement

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

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.

The above problems are closely related to each other [35]. Given a query graph q and a set D of data graphs, we can address the subgraph search problem through a little modification of a subgraph matching algorithm, i.e., for every data graph $G \in D$ it reports G and terminates as soon as it finds the first embedding of q in G. Since subgraph isomorphism (i.e., "Does G contain a subgraph isomorphic to q?") is NP-complete [10], the two problems are NP-hard.

2.2 Related Work

Subgraph Search. Plenty of early algorithms for subgraph search adopted an indexing-filtering-verification strategy. These algorithms can be classified into two groups as below, depending on their methods to extract features [16, 35].

First, in feature mining approaches, common features frequently appeared in data graphs are extracted. gIndex [40] extracts frequent subgraphs from data graphs, and build a prefix tree from these features. Tree+Δ [44] mines frequent trees up to predetermined size, and store them as a hash table. These approaches are known to be costly in index construction [14, 16].

Second, all features up to a user-defined size are enumerated and indexed in feature enumeration approaches. GCode [45] enumerates all paths, and produces vertex *signatures* in data graphs by using the paths. CT-index [19] enumerates tree and cycle features, whereas SING [8], GraphGrepSX [3], and Grapes [11] list all paths of bounded length. Since all features of data graphs are enumerated, the index construction in these approaches requires a large amount of memory, resulting in a large size of indices.

The above two approaches aim to filter out as many false answers as possible by using their indices in order to avoid exploring the whole search space for false graphs with no embeddings found in verification; however, index construction of these approaches generally takes a great deal of time and space.

Researchers recently used a filtering-verification strategy without index construction. CFQL [35] leverages existing subgraph matching algorithms to speed up subgraph search. Specifically, the preprocessing technique of CFL-Match and the search method of GraphQL are used in filtering and verification, respectively. Without index construction, CFQL outperforms indexing-filtering-verification algorithms, benefiting from the filtering power and efficient verification technique of the existing subgraph matching algorithms.

Subgraph Matching. A lot of subgraph matching algorithms [2, 7, 12, 13, 15, 20, 33, 36, 42, 43] have been suggested based on Ullmann's backtracking [37]. This approach generally works as follows: (1) for each query vertex u, a candidate set $C(u)$ is obtained through a filtering process, where $C(u)$ is a set of candidate data vertices that u can be mapped to, and (2) a matching order of query vertices is determined, and each query vertex is iteratively mapped to a candidate vertex by following the matching order. Although these algorithms were designed based on this general framework, they vary significantly in performances, which rely on a filtering method, a matching order, and a technique to prune out the search space during backtracking.

Early subgraph matching algorithms such as Ullmann [37], VF2 [7], QuickSI [33], and SPath [43] obtain a candidate set by using local filters that consider the neighborhood of vertices; however, recent algorithms such as GraphQL [15], $\text{Turbo}_{\text{iso}}$ [13], CFL-Match [2] and DAF [12] build auxiliary data structures on the query graph and the data graph in order to get small candidate sets and produce effective matching orders by estimating search cost as precisely as possible. Some algorithms eliminate redundant computations originated from the nature of backtracking (e.g., *failing sets* in [12]).

In-depth studies [23, 24, 36] comprehensively covered and investigated various subgraph matching algorithms recently developed from several different communities. In artificial intelligence, Glasgow subgraph solver (Glasgow) [25] is optimized specifically for subgraph isomorphism, but it also offers subgraph matching as well. Unlike other existing approaches, Glasgow formulates subgraph isomorphism as a *constraint programming* problem. In bioinformatics community, RI [4] proposes a backtracking method based on a global matching order.

In particular, Sun and Luo's in-depth study [36] is a seminal work that not only suggests design guidelines for efficient subgraph matching algorithms but also develops the fastest subgraph matching algorithms (that combine different techniques of existing algorithms). Consequently, we regard these algorithms (GQLfs and RIfs) as the state-of-the-art methods to be compared with our approach in Section 7.

Induced Subgraph Matching. Subgraph isomorphism has two definitions in artificial intelligence or bioinformatics communities: *non-induced subgraph isomorphism* and *induced subgraph isomorphism*. Non-induced subgraph isomorphism means an embedding defined in Section 2. Induced subgraph isomorphism additionally requires the non-adjacency condition (which means that there should exist an edge of q that corresponds to each edge between matched vertices of G). Between the two definitions, VF3 [6], Glasgow and RI mainly deal with induced subgraph isomorphism (or induced subgraph matching). In contrast, our study focuses on non-induced subgraph matching.

Summarization and Compression. Graph summarization is transforming graphs into more compact representations while preserving their structural property or the output of queries. For subgraph matching, SGMatch [31] decomposes a query graph and a data graph into the sets of *graphlets*, and matches graphplets of the query graph to the corresponding graphlets of the data graph along its graphlet matching order.

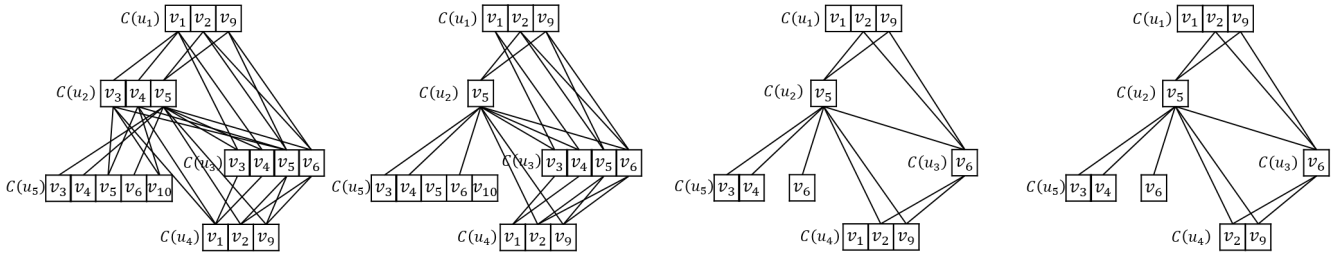
Several compression paradigms such as input compression and output compression aim to alleviate the heavy computation in subgraph matching. As an input compression technique, BoostIso [30, 39] compresses the data graph by merging symmetric vertices in preprocessing (before a query graph is given as input). For output compression, vertex-cover-based compression (VCBC) [29] encodes an output embedding into a compressed code with size smaller than that of the embedding, and crystal-based computation framework (CBF) [29] materializes not embeddings but their codes in order to reduce the overall cost of subgraph matching.

Different from the methods above, our approach employs neighbor equivalence of query vertices to decide a matching order, and finds equivalent subtrees based on neighbor equivalence within an auxiliary data structure.

Other Work. Some approaches focus on comprehensive techniques that can be applied to subgraph matching or its variations. In [26], a new matching order of query vertices (or edges) is designed specifically for a DBMS by using a set of database operators. Given a query workload, WaSQ [22] caches the embeddings of every query of the workload in advance. Next, given a new query q, it reuses the query workload and the cached embeddings to efficiently find the embeddings of q (workload-aware subgraph matching). EmptyHeaded [1] proposes a graph processing engine based on the *worst-case optimal join*.

3 OVERVIEW OF OUR APPROACH

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



(a) Initial CS **(b) After refinement of $C(u_1)$ and $C(u_2)$** **(c) After refinement of $C(u_3)$ and $C(u_5)$** **(d) After refinement of $C(u_4)$**

Figure 3: Extended DAG-graph DP over CS on q in Figure 2 and G_1 in Figure 1 using neighbor-safety

Subgraph Matching. Given a query graph q and a data graph G, our subgraph matching algorithm consists of the following three steps.

1. **Building a query DAG**. We build a *query DAG* q_D, which is a DAG that is built from q by assigning directions to the edges in q (e.g., q_D and its reverse q_D^{-1} in Figure 2 are query DAGs). The vertex with an infrequent label and a large degree is selected as the root r of q_D, and the BFS traversal is performed from r in order to build q_D [12]. We also find neighbor equivalence class (NEC) among all degree-one vertices in q, and merge the vertices in the same NEC into a single vertex in q_D, where NEC is a set of query vertices that have the same label and the same neighbors [13]. In query DAG q_D of Figure 2, the neighbor equivalence class of vertex u_5 in q_D (i.e., NEC(u_5)) corresponds to a singleton set $\{u_5\}$ in q.

2. **Building Candidate Space.** We build an auxiliary data structure *candidate space* (CS) on q and G. A CS on q and G consists of the candidate set $C(u)$ for each vertex $u \in V(q)$, and edges between the candidates as follows:

(a) For each $u \in V(q)$, there is a candidate set $C(u)$, which is a set of vertices in G that u can be mapped to. (The exact condition of mapping is described in Section 4.)
(b) There is an edge between $v \in C(u)$ and $v' \in C(u')$ if and only if $(u, u') \in E(q)$ and $(v, v') \in E(G)$.

Figure 3a shows a CS on q in Figure 2 and G_1 in Figure 1. Three candidates v_1, v_2, v_9 are in $C(u_1)$, and there is an edge between $v_2 \in C(u_1)$ and $v_5 \in C(u_2)$. CS is an auxiliary data structure used in [12], but our CS construction is different from that of [12]. We build a more compact CS by using *extended DAG-graph DP (dynamic programming)* with an additional filtering function that utilizes a concept called *neighbor-safety* (Section 4).

If there is any $u \in V(q)$ such that $C(u) = \emptyset$, we return no results (because there cannot be an embedding of q in G if there is any empty candidate set); proceed to the next step otherwise.

3. **Matching**. We match query vertices $u \in V(q)$ to candidate vertices in $C(u)$ of CS by our new matching order which is based on the number of unmapped *extendable candidates* of u and the size of NEC(u) (Section 5). Furthermore, we propose a new technique to prune out repetitive subtrees of the search space by utilizing dynamic equivalence of the subtrees (Section 6). We also apply failing sets of [12] in our algorithm.

Subgraph Search. In a general framework for subgraph search, an index I is built from a given set D of data graphs. Given a query graph q, a set D of data graphs, and the index I, we can execute the following steps, and output a set A_q of answer graphs.

1. **Filtering using an index**. For every feature in I that does not contain q, the data graphs with the feature are filtered out. The set of remaining data graphs in D is denoted by B_q.

Next, we proceed to the following steps for q and each data graph $G \in B_q$.

2. **Building a query DAG**. A query DAG q_D is built from q in the same way as in subgraph matching.
3. **Building Candidate Space**. For the query DAG q_D and data graph G, we build CS in the same way as in subgraph matching.
4. **Searching**. Unlike **Matching** above, we 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.

Based on our empirical study, building an existing index and filtering using the index incur considerable overhead without gaining higher filtering power for most queries, which is already confirmed by [35]; indeed, the state-of-the-art subgraph search algorithm CFQL [35] has shown that existing indexing methods followed by recent preprocessing and enumeration techniques are inefficient in query processing on widely-used datasets such as PDBS, PCM, and PPI. Therefore, we do not use an index, and regard B_q as a set D of data graphs. Nevertheless, one might take advantage of an index as occasion arises (e.g., I/O intensive applications).

4 FILTERING BY NEIGHBOR-SAFETY

In this section we describe a dynamic programming approach combined with a filtering technique in order to obtain a compact CS.

Let a *path tree* $T(q)$ of a DAG q be the tree such that each root-to-leaf path corresponds to a distinct root-to-leaf path in q, and $T(q)$ shares common prefixes of root-to-leaf paths of q (see Figure 2). A *weak embedding* M of a rooted DAG q with root u at $v \in V(G)$ is defined as a homomorphism of $T(q)$ such that $M(u) = v$.

Given a CS, we define a dynamic programming (DP) table $D[u, v]$ for $u \in V(q)$ and $v \in V(G)$: $D[u, v] = 1$ if $v \in C(u)$ and the following necessary conditions for an embedding that maps u to v hold; $D[u, v] = 0$ otherwise.

(1) There is a *weak embedding* M of a sub-DAG q_u at v (i.e., a homomorphism of $T(q_u)$ such that $M(u) = v$) in the CS.
(2) Any necessary condition $h(u, v)$ (other than Condition (1)) for an embedding that maps u to v is true in the CS. (Below we suggest a new necessary condition $h(u, v)$.)

$D[u, v]$ can be computed using the following recurrence in a bottom up order from leaf vertices to the root vertex, i.e., u is processed after all its children in q are processed:

$$D[u, v] = \begin{cases} 1 & \text{if } \bigwedge_{u_c \in \text{Child}(u)} f(D[u_c, \bullet], v) \wedge h(u, v) \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where a *main function* $f(D[u_c, \bullet], v)$ is 1 if there is v_c adjacent to v in the CS such that $D[u_c, v_c] = 1$; 0 otherwise. Applying h along with f is more effective in filtering than using only f in dynamic programming and applying h separately.

After dynamic programming, the new candidate set is computed as follows: v is in the new $C(u)$ if and only if $D[u, v] = 1$. (Note that candidate sets $C(u)$ serve as a compact representation of D.) This optimization technique will be called *extended DAG-graph DP*. Let the optimization such that $h(u, v)$ is omitted from Recurrence (1) be *simple DAG-graph DP*.

Now we define a necessary condition h for an embedding.

Definition 4.1. For each vertex $u \in V(q)$ and a label $l \in \Sigma$, a *neighbor set* $Nbr_q(u, l)$ is the set of neighbors of u labeled with l. For each vertex $v \in C(u)$ and a label $l \in$, a *neighbor set* $Nbr_{\text{CS}}(u, v, l)$ is defined as $\cup_{u_n \in Nbrq(u,l)} \{v_n \in C(u_n) \mid v_n \text{ is adjacent to } v \in C(u) \text{ in CS}\}$.

Definition 4.2. Given a query graph q and a CS on q and G, we say that $v \in C(u)$ is *neighbor-safe regarding* u if for every label $l \in \Sigma$, $|Nbr_q(u, l)| \leq |Nbr_{\text{CS}}(u, v, l)|$.

Example 4.1. In a query graph q of Figure 2, $Nbr_q(u_2, A) = \{u_1, u_4\}$, and $Nbr_q(u_2, B) = \{u_3, u_5\}$. In CS of Figure 3a, $Nbr_{\text{CS}}(u_2, v_3, A) = \{v_1\}$, and $Nbr_{\text{CS}}(u_2, v_5, B) = \{v_3, v_4, v_6\}$. According to Definition 4.2, v_3 is not neighbor-safe regarding u_2 since $|Nbr_q(u_2, A)| > |Nbr_{\text{CS}}(u_2, v_3, A)|$, whereas v_5 is neighbor-safe regarding u_2.

Lemma 4.1. Suppose that we are given a CS on q and G. For each vertex $u \in V(q)$, mapping u to a candidate vertex $v \in C(u)$ cannot lead to an embedding of q if v is not neighbor-safe regarding u.

By Lemma 4.1 we define $h(u, v)$ such that $h(u, v) = 1$ if v is neighbor-safe regarding u; $h(u, v) = 0$ otherwise.

Lemma 4.2. Given a CS on q and G, the time complexity of extended DAG-graph DP on the CS is $O(|E(q)| \times |E(G)|)$.

The time complexity above includes the computation of neighbor-safety, but it remains the same as the complexity of DP in [12].

Construction of a Compact CS. By using the above optimization technique multiple times with different query DAGs, we can filter as many candidate vertices as possible, and thus compute a compact CS.

At the beginning an initial CS is constructed. For each $u \in V(q)$, $C(u)$ is initialized as the set of vertices $v \in V(G)$ such that $L_G(v) = L_q(u)$. In addition, the neighborhood label frequency (NLF) filter [13] can remove $v \in C(u)$ such that there is a label $l \in \Sigma$ that satisfies $|Nbr_q(u, l)| > |Nbr_G(v, l)|$. We implement NLF as a bit array with $4|\Sigma||V(g)|$ bits to represent $|NLF_g(v, l)|$ up to 4 for each $v \in V(g)$ and $l \in \Sigma$. Therefore it can filter $v \in C(u)$ with $|Nbr_G(v, l)| < 4$ such that $|Nbr_q(u, l)| > |Nbr_G(v, l)|$. Figure 3a illustrates an initial CS on a query graph q in Figure 2 and a data graph G_1 in Figure 1.

Since DP is executed based on a query DAG, we use the DAG q_D and its reverse q_D^{-1} (in Figure 2) to refine candidate sets. In the first step of refinement we run simple DAG-graph DP using q_D^{-1} to the initial CS. In the second step we further refine the CS using q_D via DAG-graph DP for subgraph search or extended DAG-graph DP for subgraph matching. In the third step, we perform extended DAG-graph DP using q_D^{-1}.

Example 4.2. Given q_D^{-1} in Figure 2 and CS in Figure 3a, we refine $C(u_1)$ first, and then refine $C(u_2)$, and so on. After the refinement of $C(u_1)$ and $C(u_2)$ in Figure 3b, v_3 and v_4 are removed from $C(u_2)$ since they are not neighbor-safe regarding u_2. After the refinement of $C(u_5)$ in Figure 3c, therefore, v_5 is removed from $C(u_5)$ since there is no $v_c \in C(u_2)$ adjacent to v_5. In the same figure, $v_3, v_4 \in C(u_3)$ are not neighbor-safe regarding u_3, and $v_5 \in C(u_3)$ has no neighbors in $C(u_2)$, so they are removed from $C(u_3)$.

Finally, we terminate if there exists an empty candidate set, i.e., $C(u) = \emptyset$. Otherwise, after multiple execution of optimization, we get the final CS. We can repeat extended DAG-graph DP by alternating q_D and q_D^{-1} until no changes occur in candidate sets, but three steps of DP are enough from our empirical study.

5 MATCHING ORDER BASED ON STATIC EQUIVALENCE

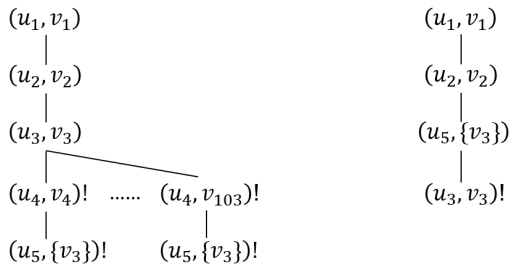
In this section we propose an improved adaptive matching order of query vertices by using static equivalence of the vertices.

Suppose that we are trying to extend a partial embedding M in the search process.

An unmapped vertex u of a query graph q in M is called *extendable* regarding M if at least one neighbor of u is matched in M, and the set $C_M(u)$ of *extendable candidates* of u regarding M is defined as the set of vertices $v \in C(u)$ adjacent to $M(u_n)$ in CS for every mapped neighbor u_n of u. We select an extendable vertex u as the next vertex and match u to each extendable candidate of u.

However, our adaptive matching order is different from those of existing algorithms. State-of-the-art subgraph matching algorithms [2, 12] adopt leaf decomposition strategy in which the vertices in the query graph are decomposed into the set of degree-one vertices and the rest, and the degree-one vertices are matched after the non-degree-one vertices are matched. This method generally helps postponing redundant Cartesian product [2]; nevertheless, it sometimes spends unnecessary search space especially when there are small number of candidates of degree-one vertices. We take all query vertices into consideration in our adaptive matching order to reduce the search space.

Example 5.1. Consider a query DAG q_D of q in Figure 2 and a data graph G_2 in Figure 1. Note that there is no embedding of q in G_2. The search trees in Figure 4 illustrate the search process. A node (u, v) represents the last mapping pair of a partial embedding M, and let M denote a node as well as a partial embedding. A node $(u, v)!$ means a mapping conflict, i.e., v is already matched therefore u cannot be mapped to v. Let $(u, \{v_1, ..., v_n\})$ represent that the n vertices in G are matched to a vertex u in q_D where $n = |\text{NEC}(u)|$. Based on the leaf decomposition as shown in Figure 4a, leaf vertex u_5 is matched after the non-degree-one vertices are matched. Specifically, given a partial embedding $M = \{(u_1, v_1), (u_2, v_2)\}$, we select u_3 as the next extendable vertex to match, and then match u_4 and u_5; however, none of partial embeddings lead to embeddings. Therefore, matching u_4 and u_5 to all their extendable candidates causes huge



(a) Search tree of the existing algorithms with leaf decomposition **(b) Search tree of the matching order based on static equivalence**

Figure 4: Search trees of two different adaptive matching orders where $(u, v)!$ means a mapping conflict (i.e, v is already matched therefore u cannot be mapped to v)

redundant search space by postponing a mapping conflict of u_3 and u_5 at v_3.

New Matching Order. In our adaptive matching order to select the next extendable vertex, we can save much search space by allowing the flexibility in the matching order of degree-one vertices. Suppose that we are trying to extend a partial embedding M.

- If there is a degree-one extendable vertex u such that $|\text{NEC}(u)| \geq |U_M(u)|$ where $U_M(u)$ denotes the set of unmapped extendable candidates of u in $C_M(u)$,
  - If $|\text{NEC}(u)| > |U_M(u)|$, backtrack.
  - Otherwise (i.e., if $|\text{NEC}(u)| = |U_M(u)|$), select u as the next vertex.
- Otherwise,
  - If there are only degree-one extendable vertices, select one of them as the next vertex.
  - Otherwise, select an extendable vertex u such that $|C_M(u)|$ is the minimum among non-degree-one vertices.

Example 5.2. Consider the search tree of the new matching order in Figure 4b. Recall that neighbor equivalence class $\text{NEC}(u_5)$ of vertex u_5 in q_D corresponds to a singleton set $\{u_5\}$ in q. Given a partial embedding $M = \{(u_1, v_1), (u_2, v_2)\}$ and $U_M(u_5) = \{v_3\}$, we choose u_5 as the next vertex to match since $|\text{NEC}(u_5)| = |U_M(u_5)|$. After we extend M to $M \cup \{(u_5, \{v_3\})\}$, there are no degree-one extendable vertices, so we choose u_3 as the next vertex to match. Hence, we can detect a mapping conflict of u_3 and u_5 at v_3 as early as possible without matching u_4.

6 RUN-TIME PRUNING BY DYNAMIC EQUIVALENCE

In this section we develop a new technique to dynamically remove equivalent subtrees of the search tree based on neighbor equivalence of candidate vertices in CS. Once we visit a new node (i.e., a new partial embedding) M, we explore the subtree rooted at M and come back to node M. By utilizing neighbor equivalence of candidates and the knowledge gained from the exploration of the subtree rooted at M, we can prune out some partial embeddings among the siblings of node M.

Definition 6.1. Suppose that we are given a CS on q and G. For a vertex $u \in V(q)$ and two candidate vertices v_i and v_j in $C(u)$, we say that v_i and v_j *share neighbors* if for every neighbor u_n of u in q, v_i and v_j have common neighbors in $C(u_n)$. Then a *cell* $\pi(u, v)$ is defined as a set of vertices $v' \in C(u)$ that share neighbors with v in CS.

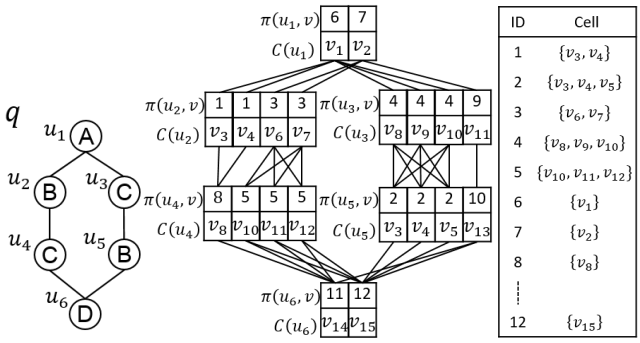


Figure 5: A query graph q and CS. Every cell $\pi(u, v)$ is represented as a unique ID according to a table above

As a new running example, we use a query graph q and the CS in Figure 5. For example, v_3, v_4 and v_5 in $C(u_5)$ share neighbors in CS, thus $\pi(u_5, v_3) = \pi(u_5, v_4) = \pi(u_5, v_5) = \{v_3, v_4, v_5\}$, while only v_3 and v_4 in $C(u_2)$ share neighbors in CS, i.e., $\pi(u_2, v_3) = \pi(u_2, v_4) = \{v_3, v_4\}$.

Assume in the rest of this section that we are given a partial embedding M, an extendable vertex $u \in V(q)$, and $v_i \in C_M(u)$ after the exploration of the subtree rooted at $M \cup \{(u, v_i)\}$. Let $\mathcal{T}_M(u, v_i)$ denote the set of embeddings found in the subtree rooted at $M \cup \{(u, v_i)\}$. For some unmapped extendable candidates $v_j \in C_M(u)$, we aim to avoid exploring the subtree rooted at $M \cup (u, v_j)$ if possible (i.e., if the subtree rooted at $M \cup (u, v_i)$ and the subtree rooted at $M \cup (u, v_j)$ are *equivalent*, which is defined below).

Definition 6.2. Given an (partial) embedding M^* in the subtree rooted at $M \cup \{(u, v_i)\}$, (partial) embedding M_s^* *symmetric* to M^* at v_j is $M^* - \{(u, v_i)\} \cup \{(u, v_j)\}$ if v_j is not mapped in M^*; $M^* - \{(u, v_i), (u', v_j)\} \cup \{(u, v_j), (u', v_i)\}$ if v_j is mapped to u' in M^*.

Definition 6.3. The subtree rooted at $M \cup \{(u, v_i)\}$ and the subtree rooted at $M \cup \{(u, v_j)\}$ are *equivalent* if

- when the subtree rooted at $M \cup \{(u, v_i)\}$ has no embeddings (i.e., $\mathcal{T}_M(u, v_i) = \emptyset$), the subtree rooted at $M \cup \{(u, v_j)\}$ also has no embeddings, and
- when $\mathcal{T}_M(u, v_i) \neq \emptyset$, for every embedding $M^* \in \mathcal{T}_M(u, v_i)$ there exists an embedding $M_s^* \in \mathcal{T}_M(u, v_j)$ symmetric to M^* at v_j, and vice versa.

Now we will find the condition that guarantees the equivalence.

Definition 6.4. Let $I_M(u, v_i)$ be the set of all mappings (u', v_i) that conflict with (u, v_i) at v_i in the subtree rooted at $M \cup (u, v_i)$, and $O_M(u, v_i)$ be the set of all mappings (u', v') visited in the subtree such that $v_i \notin \pi(u', v')$. A *negative cell* $\pi_M^-(u, v_i)$ *regarding* M is $\pi(u, v_i) \cap \{\cap_{(u', v_i) \in I_M(u, v_i)} \pi(u', v_i)\}$ if there was at least one mapping conflict at v_i in the subtree; $\pi(u, v_i)$ otherwise. A *positive cell* $\pi_M^+(u, v_i)$ *regarding* M is $\pi_M^-(u, v_i) - \delta_M(u, v_i)$ where $\delta_M(u, v_i) = \cup_{(u', v') \in O_M(u, v_i)} \pi(u', v')$. The *equivalence set* $\pi_M(u, v_i)$ *regarding* M is defined as $\pi_M^-(u, v_i)$ if $\mathcal{T}_M(u, v_i) = \emptyset$; $\pi_M^+(u, v_i)$ otherwise.

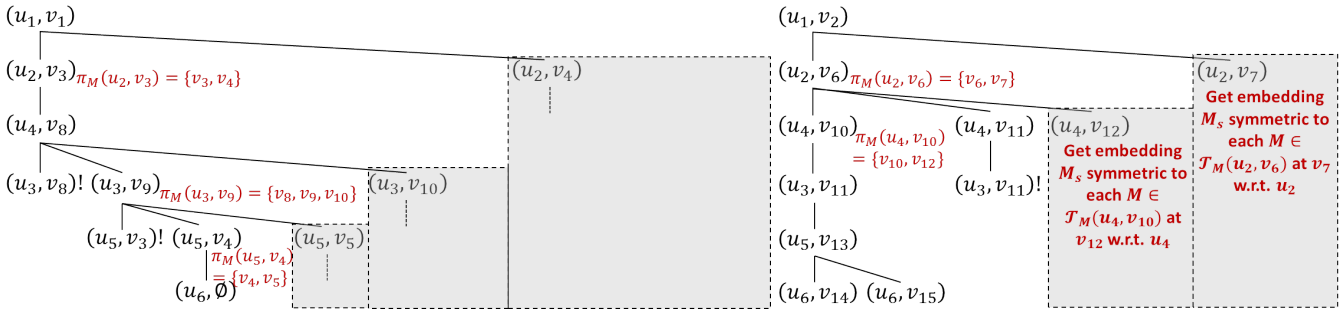


Figure 6: Pruned search tree. Nodes enclosed by dashed boxes are pruned by dynamic equivalence

Example 6.1. Figure 6 is a search tree for a query graph q and a CS in Figure 5. Suppose that we just came back to node $M \cup \{(u_3, v_9)\}$ where $M = \{(u_1, v_1), (u_2, v_3), (u_4, v_8)\}$ after the exploration of the subtree rooted at $M \cup (u_3, v_9)$. The equivalence set $\pi_M(u_3, v_9)$ regarding M is $\pi^-_M(u_3, v_9) = \pi(u_3, v_9) = \{v_8, v_9, v_{10}\}$ since there was no mapping conflict at v_9. For $M \cup \{(u_2, v_3)\}$ where $M = \{(u_1, v_1)\}$, there was a mapping conflict at v_3 after the exploration of the subtree rooted at $M \cup (u_2, v_3)$ with no embeddings found, thus $\pi_M(u_2, v_3)$ is $\pi^-_M(u_2, v_3) = \pi(u_2, v_3) \cap \pi(u_5, v_3) = \{v_3, v_4\}$. Suppose that we just came back to node $M \cup \{(u_4, v_{10})\}$ where $M = \{(u_1, v_2), (u_2, v_6)\}$ after the exploration of the subtree rooted at $M \cup (u_4, v_{10})$. Since there were no mapping conflicts during the exploration, $\pi^-_M(u_4, v_{10})$ is $\pi(u_4, v_{10}) = \{v_{10}, v_{11}, v_{12}\}$. At the same time, we visited a mapping (u_3, v_{11}) such that $v_{10} \notin \pi(u_3, v_{11})$, and thus $\pi^+_M(u_4, v_{10}) = \pi^-_M(u_4, v_{10}) - \delta_M(u_4, v_{10}) = \{v_{10}, v_{12}\}$ where $\delta_M(u_4, v_{10}) = \pi(u_3, v_{11}) = \{v_{11}\}$. Since we found embeddings during the exploration, $\pi_M(u_4, v_{10})$ is $\pi^+_M(u_4, v_{10})$.

Lemma 6.1. For every $v_j \in \pi_M(u, v_i)$, the subtree rooted at $M \cup \{(u, v_i)\}$ and the subtree rooted at $M \cup \{(u, v_j)\}$ are equivalent (i.e., π_M is the condition that guarantees the equivalence).

Example 6.2. Consider the search tree in Figure 6 again. When we come back to the node $M \cup \{(u_3, v_9)\}$ where $M = \{(u_1, v_1), (u_2, v_3), (u_4, v_8)\}$ after the exploration of the subtree rooted at $M \cup \{(u_3, v_9)\}$, we could not find any embeddings of q and there were no mapping conflicts at v_9 during the exploration. Therefore, no matter which vertex in $\pi_M(u_3, v_9)$ is matched to u_3, it will not lead to an embedding of q because all possible extensions will end up with failures in the same way. Hence, we need not extend the siblings $M \cup \{(u_3, v_j)\}$ of node $M \cup \{(u_3, v_9)\}$ for each $v_j \in \pi_M(u_3, v_9)$. Similarly, suppose that we came back to the node $M \cup \{(u_2, v_3)\}$ where $M = \{(u_1, v_1)\}$ after the exploration of the subtree rooted at $M \cup \{(u_2, v_3)\}$. We could not find any embeddings of q, and there was a mapping conflict at v_3 during the exploration, so $\pi_M(u_2, v_3) = \{v_3, v_4\}$. This implies that the subtree rooted at $M \cup \{(u_2, v_4)\}$ will be the same as that rooted at $M \cup \{(u_2, v_3)\}$ except that a mapping conflict occurs at (u_5, v_4) instead of (u_5, v_3). Hence, $M \cup \{(u_2, v_4)\}$ will not lead to an embedding, so we need not extend the sibling $M \cup \{(u_2, v_4)\}$ of node $M \cup \{(u_2, v_3)\}$.

Now, suppose that we explored the subtree rooted at $M \cup \{(u_4, v_{10})\}$ where $M = \{(u_1, v_2), (u_2, v_6)\}$, and came back to the node $M \cup \{(u_4, v_{10})\}$. We found two embeddings in $\mathcal{T}_M(u_4, v_{10})$, and obtain $\pi_M(u_4, v_{10}) = \{v_{10}, v_{12}\}$, which implies that $M \cup \{(u_4, v_{12})\}$ will lead to the embedding symmetric to each $M^* \in \mathcal{T}_M(u_4, v_{10})$ at v_{12}, i.e., the same embedding as $M^* \in \mathcal{T}_M(u_4, v_{10})$ except that u_4 is mapped to v_{12}. Note that $v_{11} \in C_M(u_4)$ is not in $\pi_M(u_4, v_{10})$ since we may not find any embeddings extended from $M \cup \{(u_4, v_{11})\}$ due to a mapping conflict of u_4 and u_3 at v_{11}.

Algorithm 1: Matching(q_D, CS, M)

```
1  if |M| = |V(q_D)| then Report M;
2  else
3      Select a next extendable vertex u;
4      Set v ← inequivalent for each v ∈ C_M(u);
5      foreach v ∈ C_M(u) do
6          if v is unvisited then
7              if v is equivalent then
8                  Report embedding M*_s symmetric to each
                     M* ∈ T_M(u, eq_M(u, v)) at v;
9                  continue;
10             M' ← M ∪ {(u, v)};
11             Mark v as visited;
12             π⁻_M(u, v) ← π(u, v); δ_M(u, v) ← ∅;
13             foreach ancestor (u_a, v_a) of (u, v) where
                 v_a ∉ π(u, v) and π(u_a,v_a)∩π(u, v) ≠ ∅ do
14                 δ_M(u_a, v_a) ← δ_M(u_a, v_a) ∪ π(u, v);
15             Matching(q_D, CS, M');
16             Mark v as unvisited;
17             if T_M(u, v) = ∅ then π_M(u, v) ← π⁻_M(u, v);
18             else π_M(u, v) ← π⁻_M(u, v) − δ_M(u, v);
19             foreach v' ∈ π_M(u, v) do
20                 eq_M(u, v') ← v;
21                 Set v' ← equivalent for v' ∈ C_M(u);
22         else
23             Let M_p be parent node of (M^{-1}(v), v);
24             π⁻_{M_p}(M^{-1}(v), v) ← π⁻_{M_p}(M^{-1}(v), v) ∩ π(u, v);
```

Search Process. Matching in Algorithm 1 is our search process to find all embeddings of q in the CS. We report M as an embedding of q if $|M| = |V(q_D)|$ (line 1); otherwise, we choose an extendable vertex in line 3 (the root vertex of q_D is first selected when $|M| = 0$), and for each unvisited $v \in C_M(u)$, extend the current partial embedding M to $M' = M \cup \{(u, v)\}$, and recursively execute Matching with

M' (lines 5-24). However, our backtracking process differs from existing algorithms as follows.

On the one hand, we select the next extendable vertex u among multiple extendable vertices based on our new adaptive matching order in Section 5 (line 3).

On the other hand, the pruning technique of Lemma 6.1 is added. For every $v \in C_M(u)$, v is initialized as 'inequivalent' (line 4). Let $eq_M(u,v)$ be the vertex in $\pi_M(u,v)$ that has been already matched with u in the extension of M. For each unvisited candidate $v \in C_M(u)$, we report an embedding M_s^* symmetric to M^* at v for every embedding M^* extended from $M \cup \{(u, eq_M(u,v))\}$, and go to line 5 if $v \in C_M(u)$ is equivalent (lines 7-9); otherwise, initialize $\pi^-_M(u,v)$ and $\delta_M(u,v)$, and update $\delta_M(u',v')$ for every ancestor (u',v') of (u,v) in the search tree such that $v_a \notin \pi(u,v)$ and $\pi(u_a,v_a) \cap \pi(u,v) \neq \emptyset$ before the recursive call of Matching (lines 12-14). After the recursive invocation of Matching, $\pi_M(u,v)$ represents $\pi^-_M(u,v)$ if there has been no embedding in the subtree rooted at $M \cup \{(u,v)\}$; $\pi^+_M(u,v) = \pi^-_M(u,v) - \delta_M(u,v)$ otherwise (lines 17-18). Next, we let $eq_M(u,v')$ be v, and set 'equivalent' to every v' in $\pi_M(u,v)$ (lines 19-21). If v is already visited (line 22), a mapping conflict of $M^{-1}(v)$ and u at v occurs, so we update a negative cell $\pi^-_{M_p}(M^{-1}(v),v)$ (lines 23-24).

For subgraph search, we modify Algorithm 1 such that it finds up to one embedding of q in each $G \in D$. First, we terminate and return true as soon as we find the first embedding M. Next, for an extendable vertex u and an unvisited extendable candidate $v \in C_M(u)$ such that v is equivalent, we go to line 5 (line 8 is removed). Finally, the computation of $\delta_M(u,v)$ is no longer needed (lines 13-14 are removed).

Lemma 6.2. Given a vertex $u \in V(q)$ and the set $C_M(u)$ of extendable candidates, the time complexity to compute cells $\pi(u,v)$ for all $v \in C_M(u)$ is $O(\deg(u)|V(G)||C_M(u)|)$.

In the implementation, to compute $\pi(u,v)$ for every $u \in V(q)$ and $v \in C(u)$ is not needed because one may visit only some $v \in C(u)$ and terminate as soon as an embedding in subgraph search (or some embeddings in subgraph matching) is found. Hence, cells are computed not for all candidates in CS right after the CS construction, but for $v \in C_M(u)$ at the first time to compute $C_M(u)$; indeed, computing cells $\pi(u,v)$ for all $v \in C_M(u)$ takes reasonable time since $|C_M(u)| \ll |C(u)|$.

7 PERFORMANCE EVALUATION

In this section 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 machine running CentOS with two Intel Xeon E5-2680 v3 2.5GHz CPUs and 256GB memory.

Since these problems are NP-hard, an algorithm cannot process some queries within a reasonable time; thus, 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 metrics below which are commonly used in previous work [12, 16, 35]:

- False positive ratio $FP_q = \frac{|C_q|-|A_q|}{|C_q|}$ for query graph q: we evaluate the filtering power of the subgraph search algorithms where C_q is the set of remaining data graphs for q after filtering, and A_q is the set of answer graphs for q.
- Size of auxiliary data structure: we measure the sum of sizes of candidate sets, i.e., $\Sigma_{u \in V(q)}|C(u)|$, to evaluate the effectiveness of the subgraph matching algorithms.
- Query processing time: we measure 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. For the sake of reasonable comparison, we compute the average of the time to process query graphs solved by at least one of the competing algorithms.
- Ratio of filtering time to verification time: this shows that how much filtering or verification time accounts for in query processing time.

Although the indexing-filtering-verification approach for subgraph search such as Grapes apparently spends a large amount of time and space in indexing datasets, indexing time and index size will not be considered as metrics for the evaluation since all the other subgraph search algorithms process queries without indexing.

7.1 Subgraph Search

Since CFQL [35] significantly outperformed existing subgraph search algorithms, and Grapes [11] was generally the fastest in query processing among existing indexing-filtering-verification algorithms [16, 35], we select these two algorithms to be compared with our subgraph search algorithm $\mathsf{VEQ_S}$. Furthermore, we modify the state-of-the-art subgraph matching algorithm DAF [12] to solve subgraph search, and include it (which will be called $\mathsf{DAF_S}$ in this section) in our comparisons.

Table 2: Characteristics of real-world datasets for subgraph search where Σ is a set of distinct vertex labels

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

Real Datasets. Experiments are conducted on real-world datasets, which are PDBS, PCM, PPI used in [11, 16, 35], and IMDB, REDDIT, COLLAB provided by [41]. 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. In order to examine the algorithms, we adopt two query generation methods similar to those in previous studies,

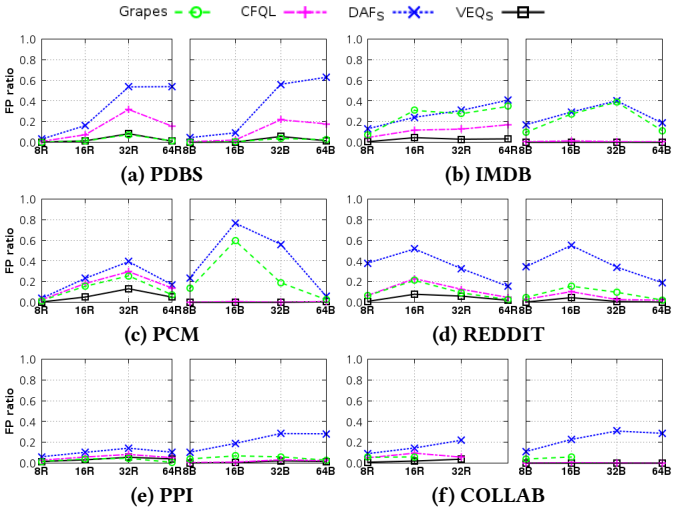


Figure 7: False positive ratio of subgraph search algorithms on real datasets

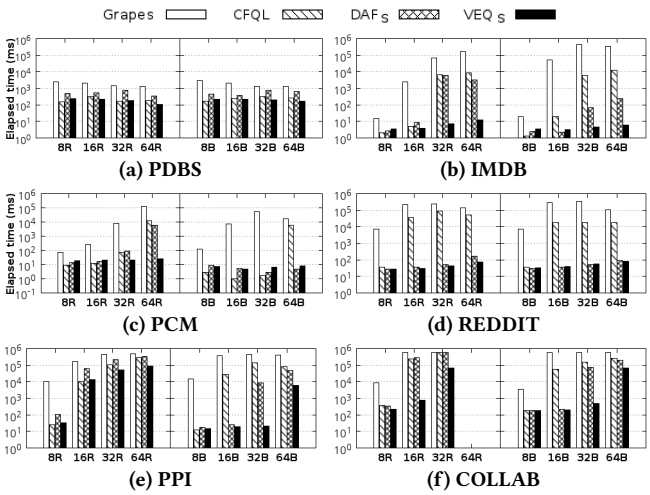


Figure 8: Query processing time of subgraph search algorithms on real datasets

which are random walk [16, 35] and breadth first search (BFS) [35, 38]. For each dataset 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 $G \in D$; (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.

False Positive Ratio. Figure 7 shows the false positive ratio of the subgraph search algorithms on the real datasets. While DAF_S is the worst in filtering false answers, VEQ_S is the best with average false positive ratio less than 0.1 in the most query sets. The big improvement of the false positive ratio originates from extended DAG-graph DP that utilizes neighbor-safety.

Query Processing Time. Figure 8 shows the average query processing time of the algorithms (no algorithm finishes a query in Q_{64R} of COLLAB within the time limit). VEQ_S is generally the fastest (except for some query sets of small sizes) due to not only fewer false positive answers obtained by extended DAG-graph DP but also the smaller search tree of the static-equivalence-based matching order shrunk by dynamic equivalence. VEQ_S is up to two orders of magnitude faster than DAF_S, and up to three orders of magnitude faster than CFQL. VEQ_S outperforms Grapes up to five orders of magnitude in Q_{32B} and Q_{64B} of IMDB. However, the query processing time of VEQ_S is slightly larger than that of CFQL in Q_{8R} and Q_{8B} of PDBS, PCM and PPI because an embedding of a small query graph can be easily found by all the algorithms, therefore exploiting extended DAG-graph DP or the pruning technique of VEQ_S may incur an overhead.

The query processing time of each algorithm varies a lot depending on the size of a query graph and the characteristic of a dataset. In general, the performance gap between VEQ_S and the others increases as the size of a query graph grows. While Grapes shows the stable performance on PDBS which is extremely sparse, its query processing time grows exponentially as the size of a query graph increases on the rest in Figure 8. Spikes in the query processing time of large queries are also observed in the results of CFQL for all the datasets other than PDBS. However, DAF_S takes nearly constant query processing time in the sparse data graphs REDDIT and PDBS, and shows more stable performance than CFQL and Grapes in the rest. The query processing time of VEQ_S remains steady as the size of a query graph increases in all the datasets except PPI (the largest data graphs) and COLLAB (a large number of the densest data graphs).

Table 3: Average ratio of filtering time to verification time (%).

	Grapes	CFQL	DAF_S	VEQ_S
PDBS	2.2 : 97.8	89.9 : 10.1	96.8 : 3.2	92.0 : 8.0
PCM	1.3 : 98.7	45.0 : 55.0	72.9 : 27.1	85.7 : 14.3
PPI	0.01 : 99.99	18.7 : 81.3	28.0 : 72.0	43.9 : 56.1
IMDB	2.6 : 97.4	21.1 : 78.9	36.9 : 63.1	67.6 : 32.4
REDDIT	0.2 : 99.8	21.9 : 78.1	94.0 : 6.0	95.3 : 4.7
COLLAB	0.4 : 99.6	19.2 : 80.8	34.1 : 65.9	49.9 : 50.1

These results originate from a ratio of filtering time to verification time as shown in Table 3. For all the competing algorithms, verification takes exponential time in the worst case whereas filtering takes polynomial time. Indeed, in Table 3, Grapes spends most of query processing time in verifying candidate graphs, which degrades the overall performance. The verification time of CFQL takes up most of its query processing time in all but PDBS. Although the verification time of DAF_S makes up over 60% of its query processing time on PPI, IMDB and COLLAB, the ratio of verification time is consistently smaller than that of CFQL. Unlike the other algorithms, VEQ_S spends the verification time less than or comparable with the filtering time, which confirms its steadier performance than the others.

Sensitivity Analysis. We evaluate the algorithms by varying several characteristics of a set D of data graphs. We generate each data graph $G \in D$ by upscaling the smallest data graph of PPI (with 2008

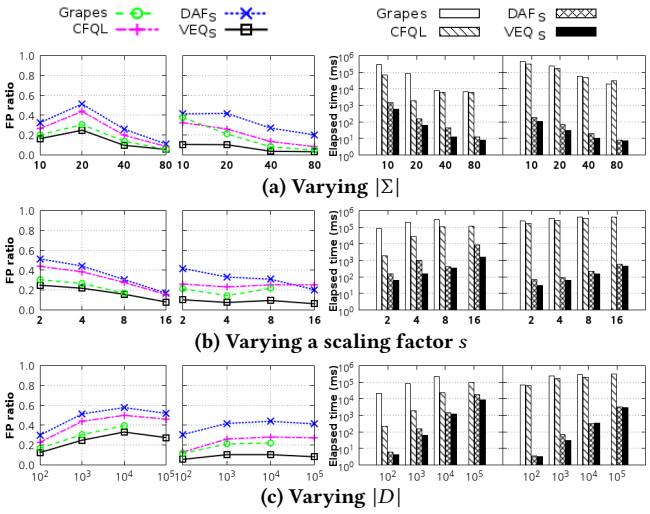


Figure 9: False positive ratio and query processing time on synthetic datasets. The results of Q_{16R} and Q_{16B} are shown in the left and right, respectively, of each figure

edges) using Evograph [27], and assign labels to vertices based on a power law distribution. We vary following parameters:

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

where s indicates that $|E(G)|$ is s times larger than that of the input data graph while Evograph keeps the same statistical properties of G by increasing $|V(G)|$ accordingly. Similarly to the existing work [16, 35], we set $|\Sigma| = 20$, $s = 2$, and $|D| = 10^3$ as default; in fact, we choose $s = 2$ so that the default $|V(G)|$ 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_{16R} and Q_{16B} on each dataset D.

False Positive Ratio. The false positive ratios of the algorithms on the synthetic datasets are displayed in the left column of Figure 9. Grapes is unable to finish indexing data graphs with $s = 16$ or those with $|D| = 10^5$ due to excessive memory usage. VEQ$_S$ consistently outperforms the others regarding false positive ratio. Overall, the false positive ratio decreases as the number of distinct labels grows, because more distinct labels on the vertices enable the algorithms to extract diverse features or to obtain fewer candidates, which results in filtering more false answers. The false positive ratio also generally decreases especially for the random walk query sets as the size of data graphs (i.e., a scaling factor s) gets larger.

Query Processing Time. The query processing time of the algorithms on the synthetic datasets is shown in the right column of Figure 9. The query processing time decreases as the number of distinct labels increases, because we can filter more data graphs by taking advantage of more labels, and verify fewer candidate graphs. The query processing time rises as a data graph gets larger since the time to verify a false positive data graph can dramatically increase. The time also rises as the number of data graphs grows, because more false positive answers may exponentially increase the verification time.

To summarize, VEQ$_S$ is better than other algorithms in filtering out false answers, and takes a smaller portion of query processing time in verification. We observe in the experiments that verification generally takes more time in a false positive answer than an answer, because an algorithm has to explore the whole search space to verify that there are no embeddings in the false positive graph while terminating as soon as it finds an embedding in the answer graph. Therefore, a smaller number of false positive answers results in fewer attempts to explore the whole search space of false positive graphs. Nevertheless, finding an embedding in an answer graph can sometimes cost a lot in the verification phase. Hence a more advanced verification technique can quickly find an embedding of a query graph in an answer graph by avoiding frequent backtracking. Consequently, lowering false positive answers (by extended DAG-graph DP with neighbor-safety) and reducing search space (by matching based on static equivalence and run-time pruning by dynamic equivalence) lead to shorter verification time, resulting in the significant improvement of overall performances.

7.2 Subgraph Matching

To evaluate the performance of our subgraph matching algorithm VEQ$_M$, we compare VEQ$_M$ with recent subgraph matching algorithms CFL-Match [2], DAF [12], RIfs [36] and GQLfs [36] from data management community, and Glasgow [25] from AI community.

Table 4: Characteristics of real datasets for subgraph matching where Σ is a set of distinct vertex labels in G

G	$\|V(G)\|$	$\|E(G)\|$	Avg degree	$\|\Sigma\|$
Yeast	3,112	12,519	8.04	71
HPRD	9,460	37,081	7.83	307
Human	4,674	86,282	36.91	44
Email	36,692	183,831	10.02	20
DBLP	317,080	1,049,866	6.62	20
YAGO	4,295,825	11,413,472	5.31	49,676

Datasets. We test the algorithms against real-world datasets in Table 4, which were widely used in previous work [2, 12, 13, 20]. Yeast, HPRD and Human are protein-protein interaction networks. The Email communication network and the DBLP collaboration network are obtained from Stanford Large Network Dataset Collection [21]. YAGO is an RDF dataset.

Query Sets. We use the same experimental setting as [2] and [12]. 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 and HPRD, and $i \in \{10, 20, 30, 40\}$ for the remaining datasets. Each query graph in Q_{iS} and Q_{iN} has the average degree ≤ 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.

Size of Auxiliary Data Structure. To evaluate how close our CS is to the optimal, we compared the size of our CS and that of Steady in [36]. Steady repeats refinements with different query vertices until no candidates are filtered out during a refinement, and it was used as an optimal CS in [36].

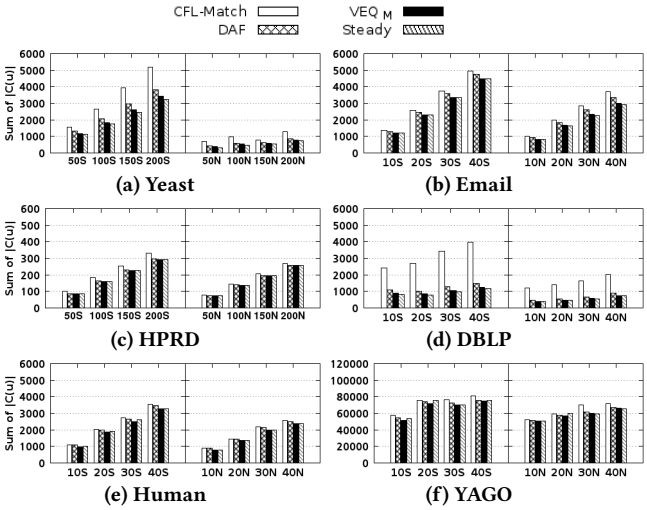


Figure 10: Sizes of auxiliary data structures of subgraph matching algorithms

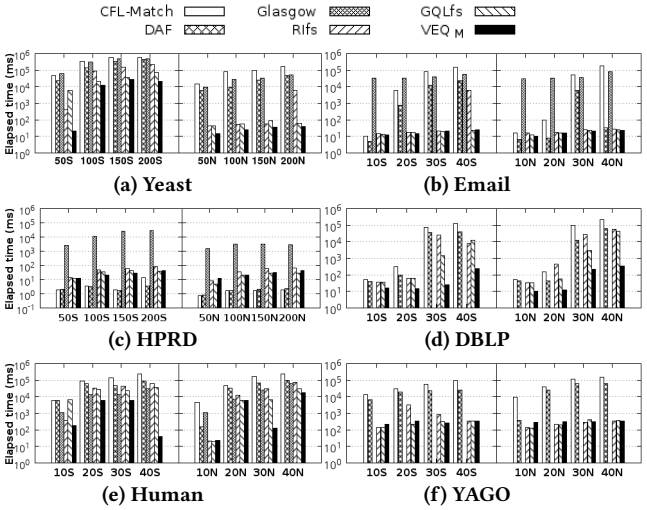


Figure 11: Query processing time of subgraph matching algorithms on real datasets

Figure 10 shows the average size of the auxiliary data structure for each algorithm and Steady. The smaller the size is, the smaller is the search space of an algorithm. The size of the auxiliary data structure grows as a query graph gets larger. VEQ_M consistently has a smaller number of candidates than DAF and CFL-Match due to extended DAG-graph DP with neighbor-safety.

The number of candidates remaining after our extended DAG-graph DP is slightly larger than that of Steady in most query sets, but sometimes less than Steady because the combination of the weak embedding and neighbor-safety (i.e., our filtering condition) is slightly stronger than the filtering condition of Steady.

Query Processing Time. Figure 11 shows the average query processing time of the algorithms. Glasgow runs out of memory on DBLP and YAGO. Due to the three main techniques described in the previous sections, VEQ_M generally outperforms GQLfs and RIfs, which is followed by DAF, CFL-Match, and Glasgow. In particular, VEQ_M outperforms RIfs by up to three orders of magnitude in Q_{40S} of Human, and GQLfs by up to two orders of magnitude in Q_{50S} of Yeast, Q_{40S} of Human, Q_{40N} of DBLP. VEQ_M is more than three orders of magnitude faster than CFL-Match and DAF in many query sets of Yeast, Email, DBLP, and Human. Different from VEQ_S, VEQ_M searches a data graph for multiple embeddings, therefore it can output numerous symmetric embeddings at once by using equivalence sets. However, the query processing time of VEQ_M is slightly more than that of the others in some query sets of HPRD and Email due to the overhead of extended DAG-graph DP and the computation of equivalence sets. For example, HPRD has a small size and many distinct labels, therefore most queries of HPRD finishes within 10ms, which means that they are easy instances for all the algorithms.

Since we find a number of embeddings in the data graph for the subgraph matching problem, the search time takes far more than the preprocessing time in all the datasets except for HPRD. Among preprocessing-search methods, VEQ_M and GQLfs spend 68% of query processing time in the search stage on average, whereas RIfs, DAF, and CFL-Match have 72%, 93%, and 96%, respectively. As a result, our strategy to obtain compact candidate sets and to reduce search space gives rise to an efficient subgraph matching algorithm.

7.3 Effectiveness of Individual Techniques

In this subsection we evaluate the effectiveness of our individual techniques in reducing the overall query processing time. We run DAF and the variants of our algorithms below to measure the performance gain achieved by each technique:

- DAF: a baseline for comparison.
- VEQ_M-SEQ-DEQ: using extended DAG-graph DP with neighbor-safety in filtering and the adaptive matching order of DAF in backtracking.
- VEQ_M-DEQ: using extended DAG-graph DP and the matching order based on static equivalence.
- VEQ_M: using extended DAG-graph DP, the matching order based on static equivalence, and run-time pruning by dynamic equivalence.

Figure 12 shows the query processing time of these algorithms for subgraph matching.

Effectiveness of Neighbor-Safety. VEQ_M-SEQ-DEQ improves DAF_M by up to two orders of magnitude for Q_{50S} of Yeast, Q_{40S} Q_{30N} of Email, and Q_{100N} of Yeast. We compute the maximum $m_q = \max_{u \in V(q), l \in \Sigma} |Nbr_q(u, l)|$ for each query graph q. The average m_q of the top 10 query graphs with the largest performance gains is much larger than that of all queries. That is, the performance gain is larger as a vertex that has more neighbors with the same label exists in a query graph, since the neighbor-safety condition can make use of this query vertex to filter out unqualified candidates of this vertex.

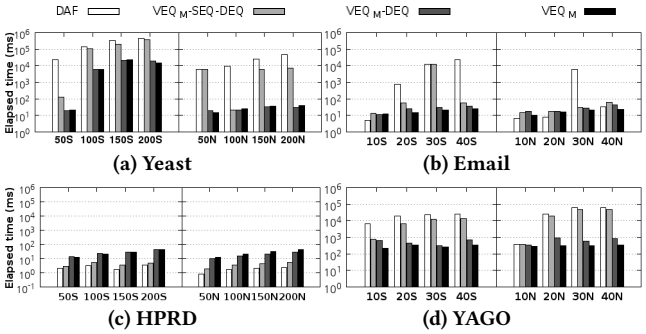
Effectiveness of Matching Order based on Static Equivalence. VEQ_M-DEQ takes into account all query vertices in our matching order, whereas VEQ_M-SEQ-DEQ considers only non-degree-one query vertices in the matching order of DAF. VEQ_M-DEQ outperforms VEQ_M-SEQ-DEQ by up to two orders of magnitude for Q_{50N}, Q_{150N}, Q_{200N} of Yeast and Q_{30S} of Email. The performance gap

Table 5: Ratio (unit: %) of the number of search tree nodes reduced by the new matching order

	Yeast								PPI							
Query set	50S	100S	150S	200S	50N	100N	150N	200N	8R	16R	32R	64R	8B	16B	32B	64B
Reduced ratio	99.7	49.5	99.998	88.7	10.1	8.1	1.2	99.5	40.2	52.7	53.6	97.3	2.6	73.0	71.1	8.4

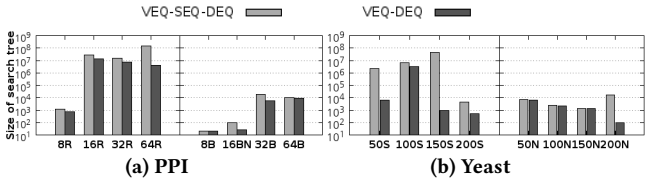
Table 6: Ratio (unit: %) of the number of search tree nodes pruned by dynamic equivalence

	DBLP								COLLAB							
Query set	10S	20S	30S	40S	10N	20N	30N	40N	8R	16R	32R	64R	8B	16B	32B	64B
Pruned ratio	89.9	97.4	99.8	99.97	95.4	99.6	99.8	99.9	53.9	98.8	98.4		0.0	34.3	98.2	96.9



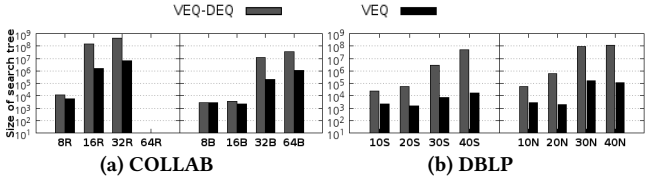
(a) Yeast (b) Email (c) HPRD (d) YAGO

Figure 12: Query processing time of DAF and our variants for subgraph matching on real datasets



(a) PPI (b) Yeast

Figure 13: Number of nodes in search trees of VEQ-SEQ-DEQ and VEQ-DEQ



(a) COLLAB (b) DBLP

Figure 14: Number of nodes in search trees of VEQ-DEQ and VEQ

between these two methods is likely to increase especially when there exist degree-one query vertices that have the same label as non-degree-one vertices.

Effectiveness of Run-Time Pruning by Dynamic Equivalence. Pruning equivalent subtrees of a search tree brings about large performance gains especially in YAGO of Figure 12. There are generally more candidates with neighbor equivalence in CS on these data graphs, resulting in many or large cells which are the potential source of the pruning power. The effectiveness of the pruning technique is obtained at the cost of extra overhead to compute cells and equivalence sets, and thus the time to process some query graphs increases a little.

Size of Search Space. To justify the effectiveness of our techniques, we measure the number of nodes in the search tree, which indicates the size of search space (Figures 13 and 14).

We compare the number of search tree nodes of the matching order of DAF and those of the new matching order in Figure 13 (run-time pruning is turned off on both methods in order to evaluate only the performance of the matching orders). In this figure, VEQ-SEQ-DEQ and VEQ-DEQ represent the the matching order of DAF and VEQ, respectively. Table 5 presents the reduced ratios by the new matching order (i.e., (size(VEQ-SEQ-DEQ) - size(VEQ-DEQ))/size(VEQ-SEQ-DEQ)), which are particularly high for sparse queries of Yeast and for random-walk queries (sparser than BFS queries) of PPI. That is, the new matching order takes more advantage of sparse query graphs that are likely to have more degree-one vertices.

We also compare the difference of search space sizes between our algorithm without run-time pruning and that with run-time pruning in Figure 14. With dynamic equivalence turned on, the size of search space becomes consistently smaller. Table 6 shows the pruned ratios by dynamic equivalence (i.e., (size(VEQ-DEQ) - size(VEQ))/size(VEQ-DEQ)), which are very high in most cases. In general, the pruned ratio increases as the size of a query graph grows.

8 CONCLUSION

To speed up subgraph search and subgraph matching, we introduce versatile equivalences: (i) equivalence of query vertices; and (ii) equivalence of candidate data vertices. In the former, we apply static equivalence of query vertices to the matching order of backtracking. In the latter, we use neighbor equivalence of candidate vertices to obtain dynamic equivalence between subtrees of a search tree so that we can prune out such redundant subtrees during backtracking. We also suggest a filtering technique of neighbor-safety through extended DAG-graph DP. These three techniques lead to improved algorithms for subgraph search and subgraph matching. Extensive experiments show that our algorithms outperform state-of-the-art algorithms for subgraph search or subgraph matching by up to orders of magnitude in query processing time.

9 ACKNOWLEDGMENTS

Hyunjoon Kim, Yunyoung Choi, and Kunsoo Park were supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No. 2018-0-00551, Framework of Practical Algorithms for NP-hard Graph Problems). Wook-Shin Han was supported by Institute of Information communications Technology Planning Evaluation(IITP) grant funded by the Korea government(MSIT) (No. 2018-0-01398, Development of a Conversational, Self-tuning DBMS).

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] 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. https://doi.org/10.1145/2882903.2915236

[3] 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.

[4] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics* 14, 7 (2013), 1–13.

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

[6] 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.

[7] 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. https://doi.org/10.1109/TPAMI.2004.75

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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. https://doi.org/10.1145/2463676.2465300

[14] 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.

[15] 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. https://doi.org/10.1145/1376616.1376660

[16] 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.

[17] 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).

[18] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the 2018 International Conference on Management of Data*. 411–426.

[19] 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.

[20] 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. https://doi.org/10.14778/2535568.2448946

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

[22] Yongjiang Liang and Peixiang Zhao. 2019. Workload-Aware Subgraph Query Caching and Processing in Large Graphs. In *Proceedings of IEEE ICDE*. 1754–1757.

[23] Ciaran McCreesh, Patrick Prosser, Christine Solnon, and James Trimble. 2018. When subgraph isomorphism is really hard, and why this matters for graph databases. *Journal of Artificial Intelligence Research* 61 (2018), 723–759.

[24] Ciaran McCreesh, Patrick Prosser, and James Trimble. 2016. Heuristics and Really Hard Instances for Subgraph Isomorphism Problems.. In *IJCAI*. 631–638.

[25] Ciaran McCreesh, Patrick Prosser, and James Trimble. 2020. The Glasgow subgraph solver: using constraint programming to tackle hard subgraph isomorphism problem variants. In *International Conference on Graph Transformation*. Springer, 316–324.

[26] 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. https://doi.org/10.14778/3342263.3342643

[27] 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.

[28] 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.

[29] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph matching: on compression and computation. *Proceedings of the VLDB Endowment* 11, 2 (2017), 176–188.

[30] Xuguang Ren and Junhu Wang. 2015. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the VLDB Endowment* 8, 5 (2015), 617–628.

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

[32] 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.

[33] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proceedings of the VLDB Endowment* 1, 1 (2008), 364–375. https://doi.org/10.14778/1453856.1453899

[34] 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.

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

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

[37] Julian R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (1976), 31–42. https://doi.org/10.1145/321921.321925

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

[39] 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.

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

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

[42] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: Distance Index Based Subgraph Matching in Biological Networks. In *Proceedings of ACM EDBT*. 192–203. https://doi.org/10.1145/1516360.1516384

[43] Peixiang Zhao and Jiawei Han. 2010. On Graph Query Optimization in Large Networks. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 340–351. https://doi.org/10.14778/1920841.1920887

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

[45] 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.