Noname manuscript No.
(will be inserted by the editor)

Fast Subgraph Query Processing and Subgraph Matching via Static and Dynamic Equivalences

Hyunjoon Kim · Yunyoung Choi · Kunsoo Park* · Xuemin Lin · Seok-Hee Hong · Wook-Shin Han*

the date of receipt and acceptance should be inserted later

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[1], 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.

Keywords Subgraph query processing · Subgraph search · Subgraph matching · Vertex equivalence · Neighbor-safety

* corresponding author

Hyunjoon Kim
Department of Data Science, Hanyang University
Department of Artificial Intelligence, Hanyang University
E-mail: hyunjoonkim@hanyang.ac.kr

Yunyoung Choi
Kyungwontech
E-mail: yychoi@theory.snu.ac.kr

Kunsoo Park
Seoul National University
E-mail: kpark@theory.snu.ac.kr

Xuemin Lin
University of New South Wales
E-mail: lxue@cse.unsw.edu.au

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

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

[1] A preliminary version [19] of this paper was presented at Proceedings of the 2021 International Conference on Management of Data (SIGMOD 2021).

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 [36]. 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 [10,38], RDF query processing [20,21], PPI network analysis [31,6], and chemical compound search [46]. 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 [46,50,51,9,4,12,22] 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 [14,3,13,40] 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 query processing performance. Researchers recently utilized existing subgraph matching algorithms to efficiently solve the subgraph search problem [39]. 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 executable files of our algorithms, datasets, and query sets are publicly available[2].

[2] https://github.com/SNUCSE-CTA/VEQ

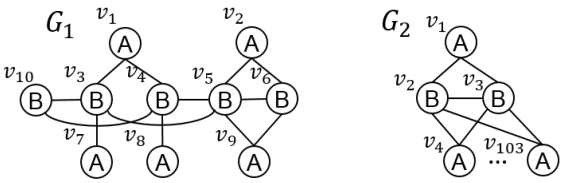


Fig. 1: A set D of data graphs

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 $L_g : V(g) \to \Sigma$ 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 $M : V(q) \to V(G)$ 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 $u \in V(q)$, 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

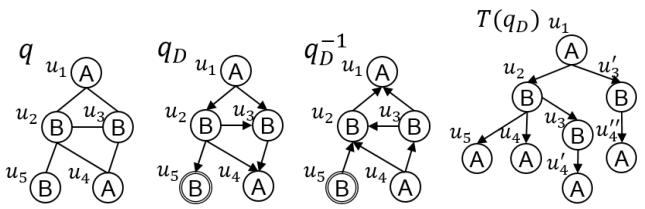


Fig. 2: A query graph q, query DAGs q_D and q_D^{-1} built from q, and a path tree $T(q_D)$ of q_D

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	Query graph
q_D	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 [39]. 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 [11], 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 [18,39].

First, in feature mining approaches, common features frequently appeared in data graphs are extracted. gIndex [46] extracts frequent subgraphs from data graphs, and build a prefix tree from these features. Tree+Δ [50] mines frequent trees up to predetermined size, and store them as a hash table. These approaches are known to be costly in index construction [15,18].

Second, all features up to a user-defined size are enumerated and indexed in feature enumeration approaches. GCode [51] enumerates all paths, and produces vertex *signatures* in data graphs by using the paths. CT-index [22] enumerates tree and cycle features, whereas SING [9], GraphGrepSX [4], and Grapes [12] 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 [39] 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 [8,37,49,16,48,14,3,13,23,40] have been suggested based on Ullmann's backtracking [43]. 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 fil-

tering method, a matching order, and a technique to prune out the search space during backtracking.

Early subgraph matching algorithms such as Ullmann [43], VF2 [8], QuickSI [37], and SPath [49] obtain a candidate set by using local filters that consider the neighborhood of vertices; however, recent algorithms such as GraphQL [16], $\text{Turbo}_{\text{iso}}$ [14], CFL-Match [3], CECI [2] and DAF [13] build auxiliary data structures on the query graph and the data graph in order to get small candidate sets. Furthermore, $\text{Turbo}_{\text{iso}}$, CFL-Match and DAF produce effective matching orders by taking advantage of the auxiliary data structures to estimate search cost as precisely as possible. Some algorithms eliminate redundant computations originated from the nature of backtracking (e.g., *failing sets* in [13]).

In-depth studies [40, 26, 27] comprehensively covered and investigated various subgraph matching algorithms recently developed from several different communities. In artificial intelligence, Glasgow subgraph solver (Glasgow) [28] 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 [5] proposes a backtracking method based on a global matching order.

In particular, Sun and Luo's in-depth study [40] 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.

Some approaches focus on comprehensive techniques for subgraph matching by employing batch query processing. $\text{MQO}_{\text{subiso}}$ [34] computes a query execution order so that cached intermediate results can be exploited. Given a query workload, WaSQ [25] 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).

Induced Subgraph Matching. Subgraph isomorphism has two different definitions in other communities such as artificial intelligence and bioinformatics: *non-induced subgraph isomorphism* and *induced subgraph isomorphism*. Non-induced subgraph isomorphism means an embedding defined in Section 2, and this is the notion of subgraph isomorphism that is commonly used in the data management community. 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 [7], Glasgow and RI mainly deal with induced subgraph isomorphism (or induced subgraph matching). In contrast, our study focuses on non-induced subgraph matching.

Multi-Way Joins. Since a multi-way join can be represented by a graph, many join-based algorithms enumerate all homomorphisms (defined in Section 2) of a query graph in a data graph. While an embedding must be injective, a homomorphism does not need to be injective, i.e., a homomorphism allows that duplicate data vertices are contained in a result, whereas an embedding does not.

Several recent graph homomorphism algorithms are designed based on *worst-case optimal join* (WCOJ), i.e., a collection of join algorithms whose running time is bounded by the number of outputs of a query graph. EmptyHeaded [1] and GraphFlow [17] employ worst-case optimal join to generate join plans specifically for small queries. RapidMatch [42] recently proposed a join-based graph homomorphism engine that can evaluate both small and large queries. Researchers also developed ways to optimize worst-case optimal plans by picking a good order of query vertices. Specifically, [29] proposed a dynamic programming optimizer that produces plans (by either performing a binary join of two smaller sub-queries or extending a sub-query by one query vertex with an intersection) and an adaptive technique that picks the query vertex order of a worst-case optimal sub-plan.

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 [35] 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 [33, 45] 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) [32] encodes an output embedding into a compressed code with size smaller than that of the embedding, and crystal-based computation framework (CBF) [32] 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.

3 Overview of Our Approach

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

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

(i) **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 [13]. 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 [14]. 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.

(ii) **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 [13], but our CS construction is different from that of [13]. 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.

(iii) **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 [13] 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.

(i) **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 $G \in B_q$.

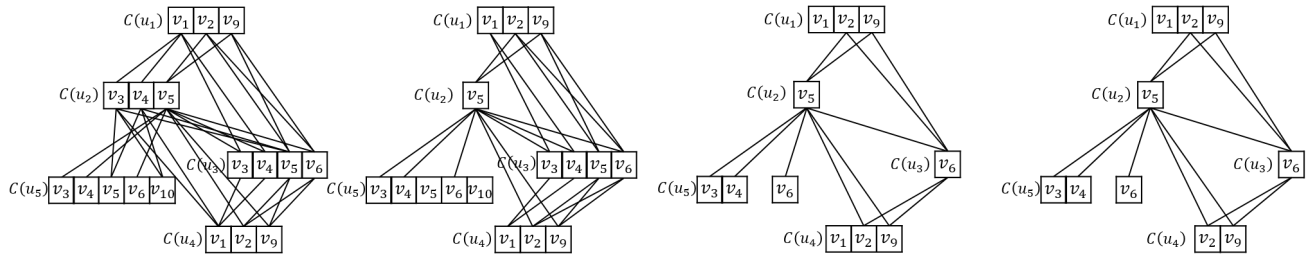
(ii) **Building a query DAG**. A query DAG q_D is built from q in the same way as in subgraph matching.

(iii) **Building Candidate Space**. For the query DAG q_D and data graph G, we build CS in the same way as in subgraph matching.

(iv) **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.

We describe above our subgraph search algorithm as a more general subgraph search framework, because our techniques can be used as the filtering and search stages of the indexing-filtering-verification framework, in which any index can be applied to the indexing and filtering stages.

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 [39]; indeed, the state-of-the-art subgraph search algorithm CFQL [39] 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 (CFQL thus runs subgraph search on multiple data graphs one by one). 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).



(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 $C(u_4)$

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

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 $M(u) = v$. These concepts were first proposed by [13].

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 (2) be *simple DAG-graph DP*. Note that DAF [13] uses simple DAG-graph DP which takes advantage of only Condition (1) above.

We propose a necessary condition that checks if the neighbors of $v \in C(u)$ with a label l in CS are enough to be mapped to u's neighbors with label l when a query vertex u is mapped to its candidate $v \in C(u)$. Given query graph q of Figure 2 and CS of Figure 3a, mapping u_2 to v_3 cannot lead to an embedding because the neighbors (i.e., v_1) of v_3 with label A are not enough to afford u_2's neighbors (i.e., u_1 and u_4) with label A.

Now we define a necessary condition h for an embedding.

Definition 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 $l \in \Sigma$, a *neighbor set* $Nbr_{\mathsf{CS}}(u, v, l)$ 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 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 $l \in \Sigma$, $|Nbr_q(u, l)| \leq |Nbr_{\mathsf{CS}}(u, v, l)|$.

Example 1 In 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, $Nbr_{\mathsf{CS}}(u_2, v_3, A) = \{v_1\}$, and $Nbr_{\mathsf{CS}}(u_2, v_5, B) = \{v_3, v_4, v_6\}$. According to Definition 2, v_3 is not neighbor-safe regarding u_2 since $|Nbr_q(u_2, A)| > |Nbr_{\mathsf{CS}}(u_2, v_3, A)|$, whereas v_5 is neighbor-safe regarding u_2.

Lemma 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.*

Proof We prove the lemma by contradiction. Assume that there exist $v \in C(u)$ which is not neighbor-safe regarding u when there is an embedding M of q that maps u to v. Since v is not neighbor-safe regarding u (i.e., there exists $l \in \Sigma$ such that $|Nbr_q(u,l)| > |Nbr_{\mathsf{CS}}(u,v,l)|$), at least two different vertices u_i and u_j in $Nbr_q(u,l)$ are mapped to the same $v_n \in Nbr_{\mathsf{CS}}(u,v,l)$ in M (i.e., $M(u_i) = M(u_j) = v_n$). This contradicts the condition that M is injective (i.e., $M(u_i) \neq M(u_j)$ for $u_i \neq u_j$) in the definition of an embedding. □

By Lemma 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 2 *Given a CS on q and G, the time complexity of extended DAG-graph DP on the CS is $O(|E(q)| \times |E(G)|)$.*

Proof Before extended DAG-graph DP, a neighbor set $Nbr_q(u,l)$ is computed for every $u \in V(q)$ and $l \in \Sigma$. Now, for each $u \in V(q)$, neighbor sets $Nbr_{\mathsf{CS}}(u,v,l)$ have to be computed for every $v \in C(u)$ and $l \in \Sigma$. To do that, for a fixed $u \in V(q)$ we need to check the edges between v and v_n for all $v \in C(u)$ and all $v_n \in C(u_n)$ where $u_n \in Nbr_q(u,l)$ by Definition 4.1. For fixed $u \in V(q)$, all neighbor sets $Nbr_q(u,l)$ are disjoint and cover all neighbors of u, and thus we look at each neighbor u_n of u only once in this computation. The number of edges between all $v \in C(u)$ and all $v_n \in C(u_n)$ is at most $O(|E(G)|)$ by Condition (b) of the Candidate Space definition in Section 3. Hence the neighbor-safety computation for all $u \in V(q)$ takes $\Sigma_{u \in V(q)}\{\deg(u) \times O(|E(G)|)\} = O(|E(q)||E(G)|)$ time. □

The time complexity above includes the computation of neighbor-safety, but it remains the same as the complexity of DP in [13].

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 [14] 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 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. Indeed, the fact that no more than three refinements are needed is also experimentally confirmed by [13].

To sum up, both extended DAG-graph DP in VEQ and simple DAG-graph DP in DAF [13] construct CS with three refinements, each of which processes DP along with the reverse topological order of query DAG. However, unlike simple DAG-graph DP, extended DAG-graph DP is a more general framework to which any necessary condition for an embedding can be added. We adopt neighbor-safety (i.e., Definition 2) as this necessary condition.

Filtering by neighbor-safety in VEQ and exploring only feasible partial mapping in VF3 [7] have two main differences. First, while VF3 computes feasibility sets on a query graph and a data graph, we apply neighbor-safety to CS that keeps only the edges between survived vertices in candidate sets. Neighbor-safety depends only on these edges, thereby resulting in high filtering power to remove unpromising candidates. Second, VF3 takes each partial mapping into account in feasibility rules whenever it tries to extend that partial mapping during the search (or enumeration) stage (because feasibility sets are computed from the current partial mapping), but neighbor-safety is a filtering technique used before the search stage.

A neighbor set is the same as a neighborhood label equivalent class (NLEC) [41]. Pseudo Star Isomorphism Constraint (PSIC) [41] is a necessary condition for an embedding, which generalizes Definition 2. Given a sequence of $Nbr_q(u,l)$, for $1 \leq i \leq |Nbr_q(u,l)|$, PSIC

incrementally compares the number (i.e., i) of the u's first i neighbors in that sequence with the size of the union of the neighbors' candidates which are adjacent to v. Meanwhile, Definition 2 compares only $|Nbr_q(u, l)|$ and the size of the union of all the neighbors' candidates adjacent to v in CS. Based on our empirical study, the improvement of the filtering power of PSIC over Definition 2 is negligible. In our work, we first combine the filtering method of Lemma 1 and DAG-graph DP in a single framework.

Neighbor-safety utilizes the label frequency of neighbors in compact auxiliary data structure CS. This does not incur an additional computational overhead to the heavy search stage. (Note that the preprocessing step takes polynomial time, whereas the search step takes exponential time in the worst case.)

5 Matching Order Based on Static Equivalence

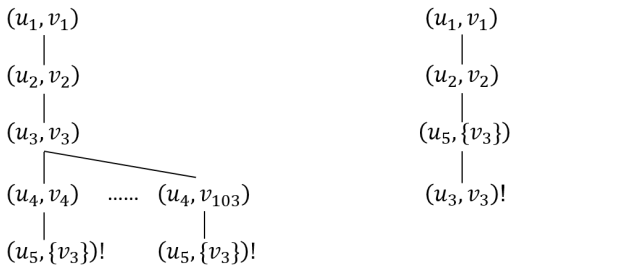
In this section we propose an improved adaptive matching order of query vertices by using static equivalence of the vertices.

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 [3,13] 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 [3]; 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 3 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 redundant search space by postponing a mapping conflict of u_3 and u_5 at v_3.



(a) Search tree of the existing algorithms with leaf decomposition

(b) Search tree of the matching order based on static equivalence

Fig. 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)

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 4 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 $|\mathrm{NEC}(u_5)| = |U_M(u_5)|$. 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 3 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.

As a new running example, we use a query graph q and a data graph G_3 in Figure 5. Note that v_3, v_4 and v_5 have different sets of neighbors in G_3. Between the two graphs, CS in Figure 6 is constructed, where v_3, v_4 and v_5 in $C(u_5)$ share neighbors in the CS (i.e., $\pi(u_5, v_3) = \pi(u_5, v_4) = \pi(u_5, v_5) = \{v_3, v_4, v_5\}$), while only v_3 and v_4 in $C(u_2)$ share neighbors in the 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 4 Given an (partial) embedding M^* in the subtree rooted at $M \cup \{(u, v_i)\}$, (partial) embedding $M_s^* \in \mathcal{T}_M(u, v_j)$ *symmetric* to M^* 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^*.

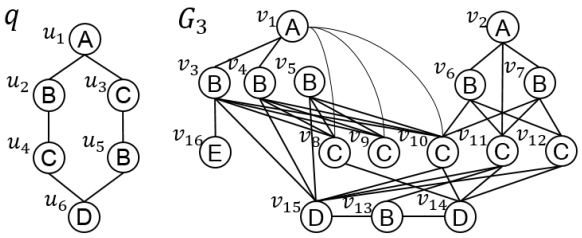


Fig. 5: A new query graph q and a data graph G_3.

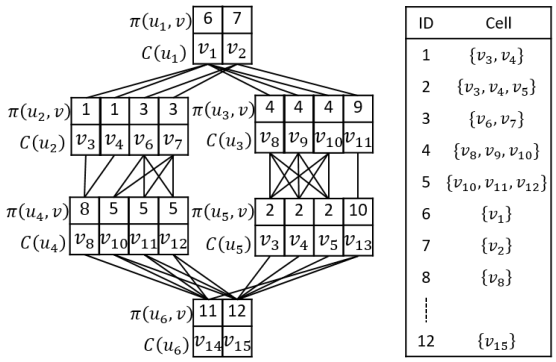


Fig. 6: CS on query graph q and data graph G_3 of Figure 5. Every cell $\pi(u, v)$ is represented as a unique ID according to a table above. Note that v_3, v_4 and v_5 in $C(u_5)$ share neighbors in the CS of Figure 5, though they have different sets of neighbors in the data graph.

Definition 5 The subtree rooted at $M \cup \{(u, v_i)\}$ and the subtree rooted at $M \cup \{(u, v_j)\}$ are *equivalent* in the following cases:

- 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^*, and vice versa.

Suppose that we are given a subtree T_i rooted at $M \cup \{(u, v_i)\}$ and its counterpart subtree T_j rooted $M \cup \{(u, v_j)\}$, as shown in Figures 7, 8, and 9.

Figure 7 shows the case in which no embedding was found in T_i. In T_i, u' could not be matched to v_i as v_i was already matched to u, and then u' was matched to v_j, but eventually no embedding was found in T_i. In T_j, the situation is the same as that of T_i except that the roles of v_i and v_j are exchanged. Then v_i and v_j are included in *negative cell* $\pi_M^-(u, v_i)$, which is defined in Definition 6.

Figure 8 shows the case that is similar to Figure 7 but an embedding was found in T_i. That is, u' was matched to v_j and then an embedding M^* was found in T_i. Again the situation in T_j is the same as that of

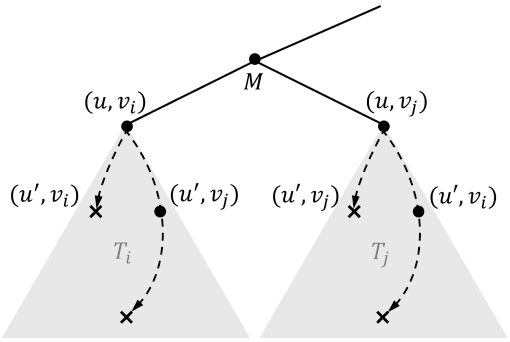


Fig. 7: Subtrees rooted at $M \cup \{(u, v_i)\}$ and $M \cup \{(u, v_j)\}$ are equivalent. Here, no embeddings are found. Negative cell $\pi^-_M(u, v_i)$ contains v_i and v_j.

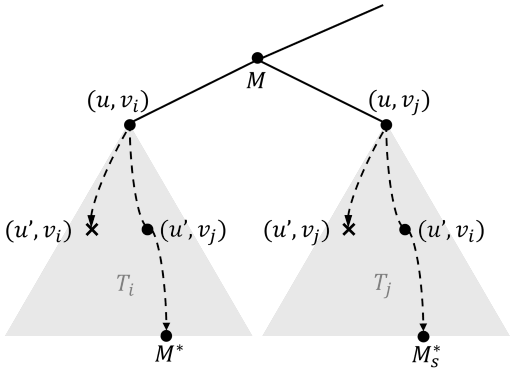


Fig. 8: Subtrees rooted at $M \cup \{(u, v_i)\}$ and $M \cup \{(u, v_j)\}$ are equivalent. Symmetric embeddings are found in these subtrees. Positive cell $\pi^+_M(u, v_i)$ contains v_i, v_j.

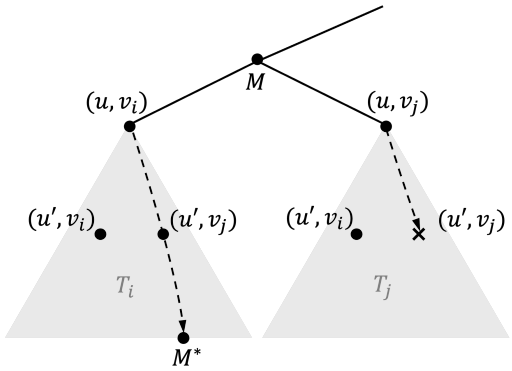


Fig. 9: Subtrees rooted at $M \cup \{(u, v_i)\}$ and $M \cup \{(u, v_j)\}$ are not equivalent. Cell $\pi(u', v_j)$ visited in T_i does not contain v_i, so v_j is included in delta set $\delta_M(u, v_i)$.

T_i except that the roles of v_i and v_j are exchanged. Hence the symmetric embedding M^*_s contains (u, v_j) and (u', v_i) instead of (u, v_i) and (u', v_j) in M^*. In this case, v_i and v_j are included in *positive cell* $\pi^+_M(u, v_i)$.

Figure 9 shows the case that an embedding M^* was found in T_i but the symmetric embedding M^*_s will not be found in T_j. In this case, v_i is not included in the cell $\pi(u', v_j)$, whereas v_i is included in $\pi(u', v_j)$ of Figures 7 and 8. In T_i, v_i has never been visited as the candidate of u' as v_i is not included in $\pi(u', v_j)$. And, u' was matched to v_j, which leads to an embedding M^*. In T_j, v_i will not be visited as the candidate of u' since v_i is not included in $\pi(u', v_j)$. And u' will not be matched to v_j as v_j is already matched to u, so the symmetric embedding M^*_s will not be found. If cell $\pi(u', v_j)$ visited in T_i does not contain v_i, then v_j is included in *delta set* $\delta_M(u, v_i)$.

Now we will formally define the condition that guarantees the equivalence.

Definition 6 Let $I_M(u, v_i)$ be the set of all mappings (u', v_i) that conflict with (u, v_i) at $v_i \in C_M(u)$ 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 $\pi(u', v') \not\ni v_i$. 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 set* $\delta_M(u, v_i) = \cup_{(u', v') \in O_M(u, v_i)} \pi(u', v')$. The *equivalence set* $\pi_M(u, v_i)$ *regarding* M is defined as follows:

$$\pi_M(u, v_i) = \begin{cases} \pi^-_M(u, v_i) & \text{if } \mathcal{T}_M(u, v_i) = \emptyset \\ \pi^+_M(u, v_i) & \text{otherwise} \end{cases} \quad (2)$$

Example 5 (Negative Cell) As a concrete example, Figure 10 is a search tree for a query graph q and a CS in Figure 6. 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 $\pi^-_M(u_2, v_3) = \pi(u_2, v_3) \cap \pi(u_5, v_3) = \{v_3, v_4\}$.

Example 6 (Positive Cell) As a concrete example in the search tree of Figure 10, 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 $\pi(u_4, v_{10}) = \{v_{10}, v_{11}, v_{12}\}$. In this exploration, we have also visited a mapping (u_3, v_{11}) such that $\pi(u_3, v_{11}) \not\ni v_{10}$, 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 $\pi^+_M(u_4, v_{10})$.

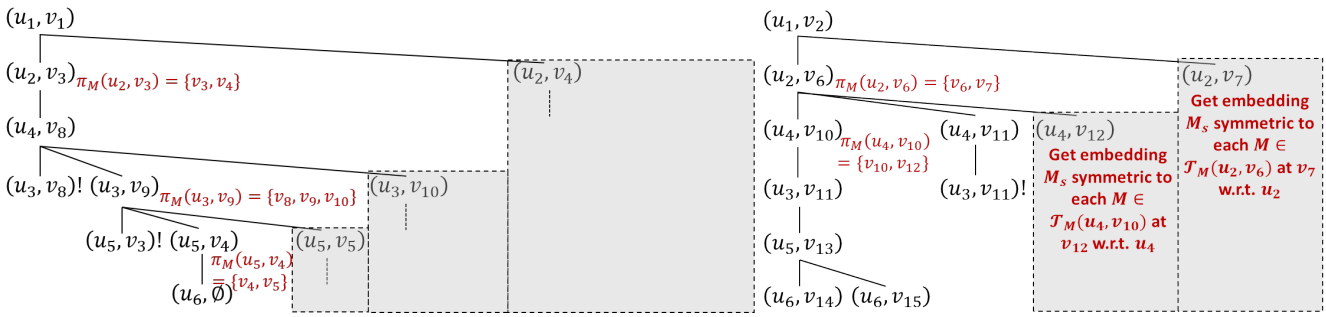


Fig. 10: Pruned search tree. Nodes enclosed by dashed boxes are pruned by dynamic equivalence

Now we claim that equivalence set $\pi_M(u, v_i)$ leads to the equivalence among the subtrees rooted at $M \cup \{(u, v_j)\}$ for every $v_j \in \pi_M(u, v_i)$.

Lemma 3 *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).*

Proof For $v_i \in C(u)$ and $v_j \in C(u)$, we need following ingredients.

(i) Two candidates v_i and v_j in $C(u)$ share neighbors.
(ii) For every vertex u' that makes a conflict with u at v_i in the subtree rooted at $M \cup (u, v_i)$, there exists $v_j \in C(u')$ that share neighbors with $v_i \in C(u')$.
(iii) For every vertex u'' mapped to v_j in the subtree rooted at $M \cup (u, v_i)$, there exists $v_i \in C(u'')$ that shares neighbors with $v_j \in C(u'')$.

The fact that v_i and v_j are in $\pi_M(u, v_i)$ can be rewritten as follows.

- If $\mathcal{T}_M(u, v_i) = \emptyset$, (i) and (ii) hold.
- Otherwise, (i), (ii), and (iii) hold.

Let $\mathcal{T}_M(u, v_i) \sqsubseteq \mathcal{T}_M(u, v_j)$ mean that 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^*. Then the definition of equivalence of subtrees can be described as follows.

- If $\mathcal{T}_M(u, v_i) = \emptyset$, then $\mathcal{T}_M(u, v_j) = \emptyset$, i.e., $\mathcal{T}_M(u, v_i) \sqsupseteq \mathcal{T}_M(u, v_j)$.
- Otherwise, $\mathcal{T}_M(u, v_i) \sqsubseteq \mathcal{T}_M(u, v_j)$ and $\mathcal{T}_M(u, v_i) \sqsupseteq \mathcal{T}_M(u, v_j)$.

We prove the lemma by contradiction. i.e., if $\mathcal{T}_M(u, v_i) \not\sqsupseteq \mathcal{T}_M(u, v_j)$ then $v_j \notin \pi_M(u, v_i)$. Let u' be the first query vertex that has the same label as u and appears after u in the matching order if there exists such a vertex; u otherwise.

Suppose that $\mathcal{T}_M(u, v_i) \not\sqsupseteq \mathcal{T}_M(u, v_j)$, i.e., there exists an embedding $M^* \in \mathcal{T}_M(u, v_j)$ not symmetric to any embeddings in $\mathcal{T}_M(u, v_i)$. That is, there exists a partial embedding M_2 that leads to M^* in the subtree rooted at $M \cup \{(u, v_j)\}$, but partial embedding M_1 symmetric to M_2 must not exist in the subtree rooted at $M \cup \{(u, v_i)\}$ or never leads to an embedding in $\mathcal{T}_M(u, v_i)$. Assume that $M_1 = M \cup \{(u, v_i), \ldots, (u', v_j)\}$ and $M_2 = M \cup \{(u, v_j), \ldots, (u', v_i)\}$ if $u' \neq u$; $M_1 = M \cup \{(u', v_i)\}$ and $M_2 = M \cup \{(u', v_j)\}$ otherwise. There exists a neighbor u_n of u' and its candidate $v_n \in C(u_n)$ such that M_2 extends to a mapping (u_n, v_n) which cannot be extended by M_1. This implies that $v_j \in C(u')$ is adjacent to $v_n \in C(u_n)$ but $v_i \in C(u')$ is not, which contradicts that $v_i, v_j \in C(u')$ share neighbors, i.e., the statement contradicts condition (ii) if $u' \neq u$; condition (i) if $u' = u$.

Suppose that $\mathcal{T}_M(u, v_i) \not\sqsubseteq \mathcal{T}_M(u, v_j)$. In the same way as above this assumption results in a contradiction to condition (iii) if $u' \neq u$; condition (i) if $u' = u$. □

Example 7 (Pruning by Negative Cells) Consider search tree in Figure 10 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 $M \cup \{(u_2, v_3)\}$.

Example 8 (Pruning by Positive Cells) Consider search tree in Figure 10 again. 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 $M^* \in \mathcal{T}_M(u_4, v_{10})$, 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
18                 π_M(u, v) ← π⁻_M(u, v)
19             else
20                 π_M(u, v) ← π⁻_M(u, v) − δ_M(u, v)
21             foreach v' ∈ π_M(u, v) do
22                 eq_M(u, v') ← v;
23                 Set v' ← equivalent for v' ∈ C_M(u);
24         else
25             Let M_p be parent node of (M⁻¹(v), v);
26             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 3 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 $M^*_s \in \mathcal{T}_M(u, v)$ symmetric to M^* 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 4 *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 $O(\deg(u)|V(G)||C_M(u)|)$.*

Proof We can obtain the cells $\pi(u, v)$ through a divide-and-conquer paradigm on the array of $C_M(u)$: (1) select a new pivot candidate $v_n \in C(u_n)$ for a neighbor u_n of u, (2) partition elements of the array into two sub-arrays according to whether each element $v \in C_M(u)$ has the pivot as a neighbor in CS or not, and (3) repeat the previous steps on each sub-array for the next pivot until the sub-array is a singleton or all possible pivots $v_n \in C(u_n)$ for all neighbors u_n of u are checked. There are $\deg(u)$ neighbors of u in q. For each neighbor u_n of u, there are $O(|V(G)|)$ neighbors in $C(u_n)$ of all $v \in C_M(u)$. Thus, the number of all possible pivots is $O(\deg(u)|V(G)|)$. As a result, 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 $|C_M(u)| \ll |C(u)|$.

We use Cell IDs for implementation efficiency and better presentation. In the implementation, we associate each distinct cell with a unique ID. Once $\pi(u, v)$ is obtained, the ID of this cell is cached for every $v' \in C(u)$ such that $v' \in \pi(u, v)$, so $\pi(u, v)$ can be reused and accessed through the ID when $v \in C(u)$ is visited later. For some candidates $v' \in \pi(u, v)$, we mark $v' \in \pi_M(u, v)$ as "equivalent" once we compute $\pi_M(u, v)$ so that we can reuse $\pi_M(u, v)$ as long as current partial embedding M stays the same.

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 [39, 18, 13]:

- 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.

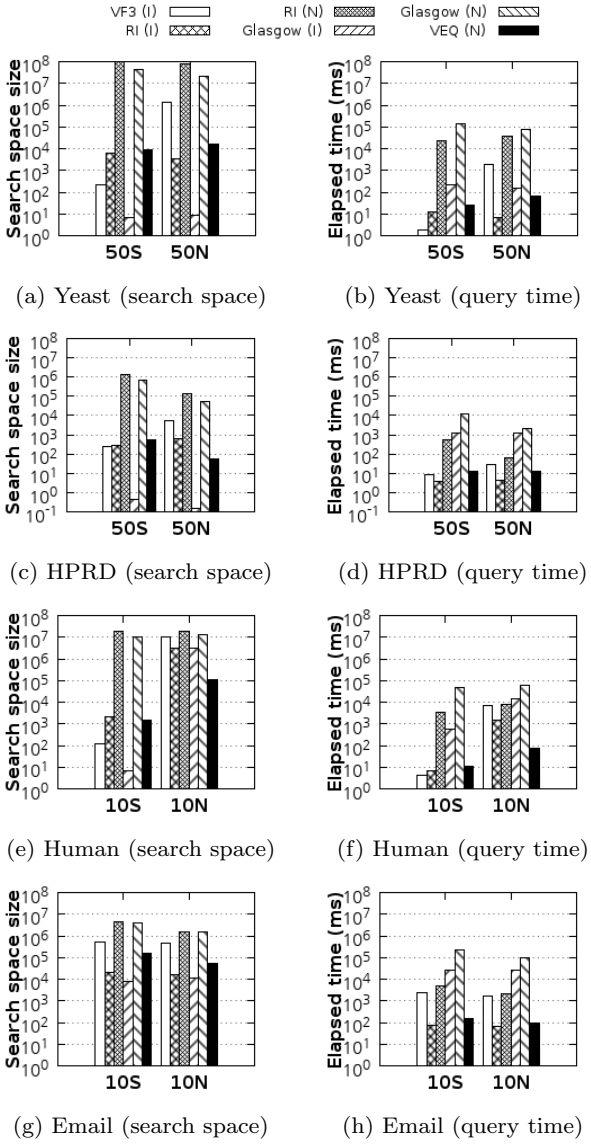
Although the indexing-filtering-verification approach for subgraph search such as Grapes apparently spends a large amount of time and space in indexing datasets, indexing time and index size will not be considered as metrics for the evaluation since all the other subgraph search algorithms process queries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

7.1 Induced vs Non-Induced

"Induced" subgraph matching and "non-induced" subgraph matching are different problems. Which one of the two problems is more difficult to solve? Since different algorithms use different techniques, it is not easy to answer the above question by comparing different algorithms. Fortunately, RI [5] and Glasgow [28] solve both the induced and non-induced subgraph matching problems, and thus we try to answer the above question by measuring the search space and query processing time of these algorithms in the induced and non-induced problems. We include VF3 [7] and VEQ_M in the experiment for reference.

Figure 11 shows the size (i.e., the number of nodes in the search tree) of search space and the query time for these algorithms, where (I) and (N) after the algorithm names denote "induced" and "non-induced", respectively. In this experiment, we run only the queries for which every algorithm finds all matches within the time limit of 10 minutes so that we can measure the search space size. RI (N) and Glasgow (N) have much larger search space than RI (I) and Glasgow (I), respectively, which indicates that non-induced subgraph matching is the problem with larger search space than induced subgraph matching. The gap of the search space between induced and non-induced algorithms therefore leads to the gap of the query processing time between them. VEQ_M consistently outperforms other non-induced algorithms.

Consequently, the two problems show a significant gap of difficulty even for the same algorithm. Hence, we mainly compare ours with other non-induced subgraph search or subgraph matching algorithms in the following experiments.



(a) Yeast (search space) (b) Yeast (query time)

(c) HPRD (search space) (d) HPRD (query time)

(e) Human (search space) (f) Human (query time)

(g) Email (search space) (h) Email (query time)

Fig. 11: Search space size and query processing time of the induced and non-induced subgraph matching algorithms. (I) and (N) after the algorithm names denote "induced" and "non-induced", respectively. In this experiment, we run only the queries for which every algorithm finds all matches within 10 minutes so that we can measure the search space size.

7.2 Subgraph Search

Since CFQL [39] significantly outperformed existing subgraph search algorithms, and Grapes [12] was generally the fastest in query processing among existing indexing-filtering-verification algorithms [39,18], 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 [13] to solve subgraph search, and include it (which will be called $\mathsf{DAF_S}$ in this section) in our comparisons.

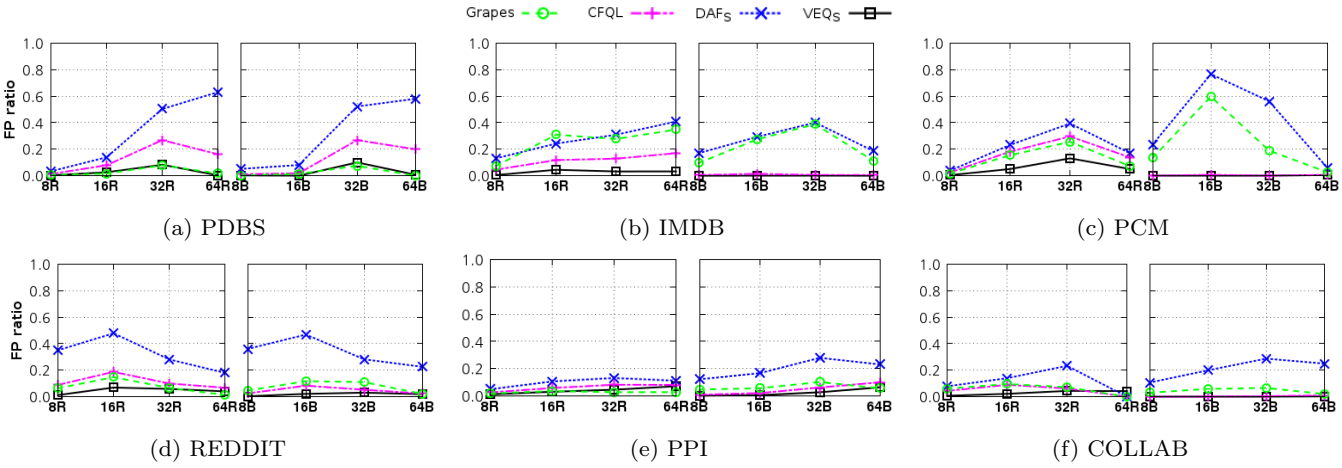
Table 2: Characteristics of real-world datasets for subgraph search where Σ is a set of distinct vertex labels

	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 [12,18,39], and IMDB, REDDIT, COLLAB provided by [47]. 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, which are random walk [18,39] and breadth first search (BFS) [44,39]. 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.

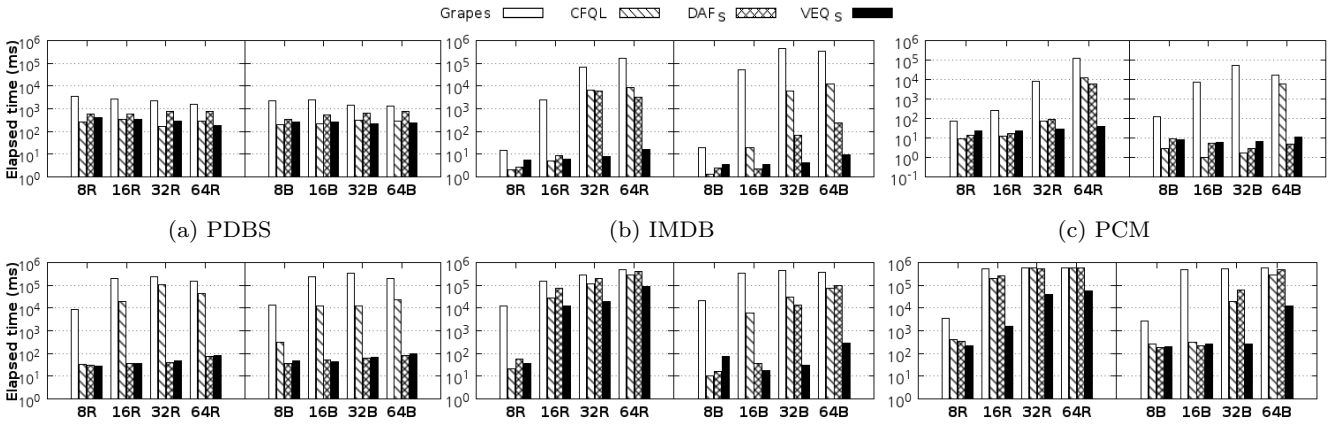
False Positive Ratio. Figure 12 shows the false positive ratio of the subgraph search algorithms on the real datasets (false positive ratio in Q_{64R} of COLLAB is missing, because no algorithms except $\mathsf{VEQ_S}$ finish any query in Q_{64R} of COLLAB within the time limit). While $\mathsf{DAF_S}$ is the worst in filtering false answers, $\mathsf{VEQ_S}$ is the best with average false positive ratio less than 0.1 in the most query sets. The big improvement of the false



(a) PDBS (b) IMDB (c) PCM

(d) REDDIT (e) PPI (f) COLLAB

Fig. 12: False positive ratio of subgraph search algorithms on real datasets



(a) PDBS (b) IMDB (c) PCM

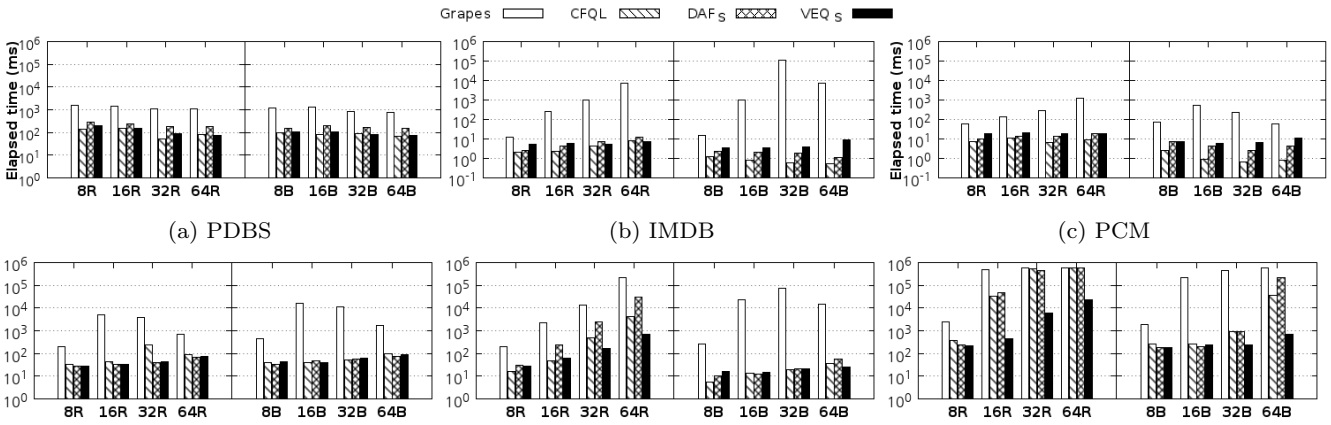
(d) REDDIT (e) PPI (f) COLLAB

Fig. 13: Arithmetic mean of query processing time for subgraph search algorithms on real datasets

positive ratio originates from extended DAG-graph DP that utilizes neighbor-safety.

Query Processing Time. Figure 13 shows the arithmetic mean of the query processing time of the algorithms. VEQS 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. VEQS is up to two orders of magnitude faster than DAFS, and up to three orders of magnitude faster than CFQL. VEQS outperforms Grapes up to five orders of magnitude in Q_{32B} of IMDB. However, the query processing time of VEQS 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 VEQS may incur an overhead.

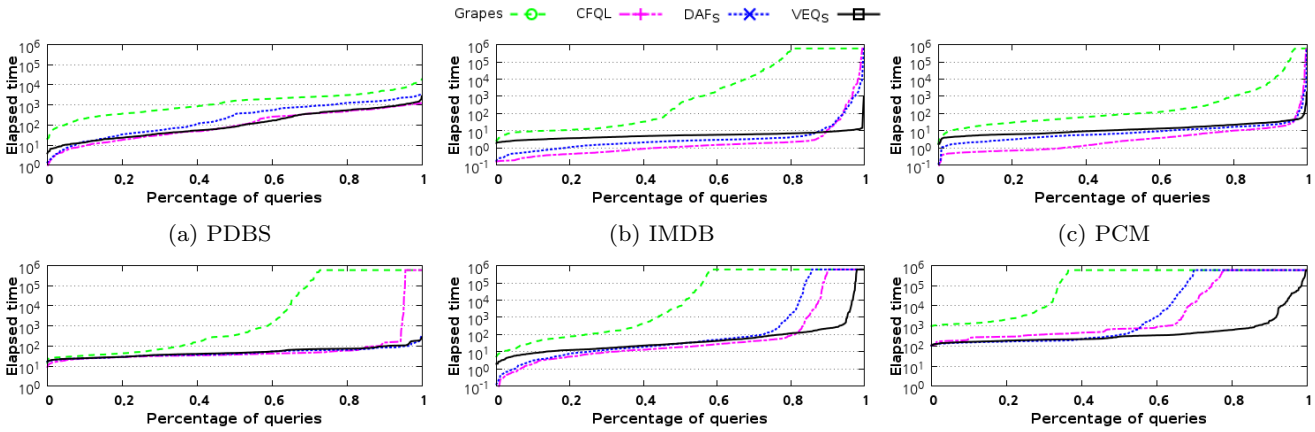
The query processing time of each algorithm varies a lot depending on the size of a query graph and the characteristic of a dataset. In general, the performance gap between VEQS and the others increases as the size of a query graph grows. While Grapes shows the stable performance on PDBS which is extremely sparse, its query processing time grows exponentially as the size of a query graph increases on the rest in Figure 13. 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, DAFS 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 VEQS 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).



(a) PDBS (b) IMDB (c) PCM

(d) REDDIT (e) PPI (f) COLLAB

Fig. 14: Geometric mean of query processing time of subgraph search algorithms on real datasets



(a) PDBS (b) IMDB (c) PCM

(d) REDDIT (e) PPI (f) COLLAB

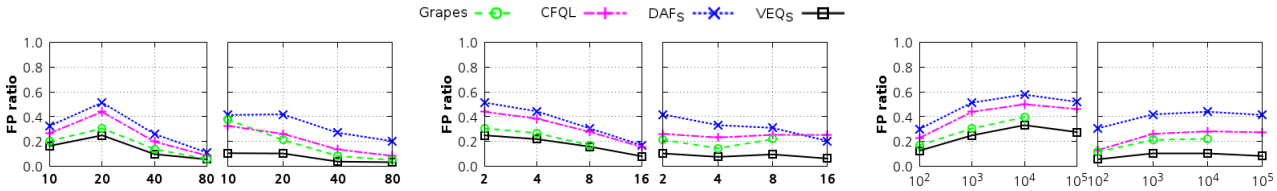
Fig. 15: Distribution of the query processing time of subgraph search algorithms on real datasets

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

	Grapes	CFQL	DAF$_S$	VEQ$_S$
PDBS	1.9 : 98.1	90.4 : 9.6	96.7 : 3.3	90.7 : 9.3
PCM	1.3 : 98.7	45.0 : 55.0	72.9 : 27.1	65.8 : 34.2
PPI	0.01 : 99.99	18.8 : 81.2	27.0 : 73.0	35.4 : 64.6
IMDB	2.6 : 97.4	21.1 : 78.9	36.9 : 63.1	52.9 : 47.1
REDDIT	0.1 : 99.9	13.0 : 87.0	96.1 : 3.9	90.2 : 9.8
COLLAB	0.7 : 99.3	28.9 : 71.1	34.3 : 65.7	52.6 : 47.4

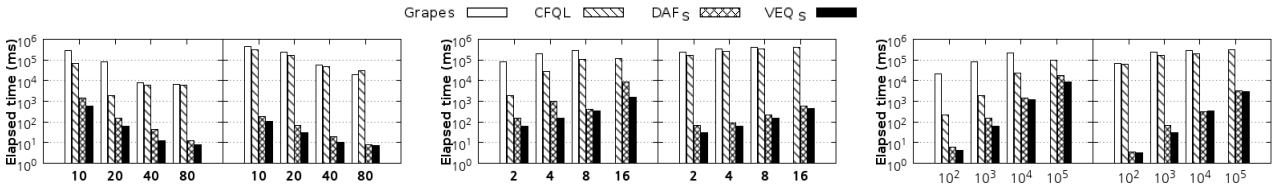
These results originate from a ratio of filtering time to verification time as shown in Table 3. For all the competing algorithms, verification takes exponential time in the worst case whereas filtering takes polynomial time. Indeed, in Table 3, Grapes spends most of query processing time in verifying 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.

For further quantification and analysis on the performance gap between the queries, we measure the geometric mean of the query processing time in Figure 14. We also present the distribution of the query processing time in Figure 15. For every algorithm, queries on each dataset are sorted in the ascending order of the query processing time of that algorithm in Figure 15. The distribution of the query processing time and the behavior of an algorithm vary by the dataset. In PDBS, the query processing time of every algorithm gradually increases. In IMDB and PCM, the query processing time of CFQL and DAF$_S$ is smaller than that of VEQ$_S$ for about 80% of the queries due to the overhead of VEQ$_S$. (This re-



(a) Varying $|\Sigma|$ (b) Varying a scaling factor s (c) Varying $|D|$

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



(a) Varying $|\Sigma|$ (b) Varying a scaling factor s (c) Varying $|D|$

Fig. 17: 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

sult is confirmed by the fact that the geometric mean of CFQL and DAFS is smaller than that of VEQS in Figure 14.) In contrast, VEQS shows relatively constant query processing time for most queries, unlike the query processing time of the others sharply increasing for hard queries. In REDDIT, Grapes and CFQL reach the time limit early for hard queries whereas VEQS shows stable performances. For both PPI and COLLAB, all the algorithms have extremely difficult queries with the query processing time over the time limit (i.e. 10 minutes). Grapes is the first to reach the time limit, and then DAFS reaches the limit, followed by CFQL. Meanwhile, VEQS has only a few queries whose elapsed times exceed the time limit.

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 edges) using Evograph [30], and assign labels to vertices based on a power law distribution. We vary following parameters:

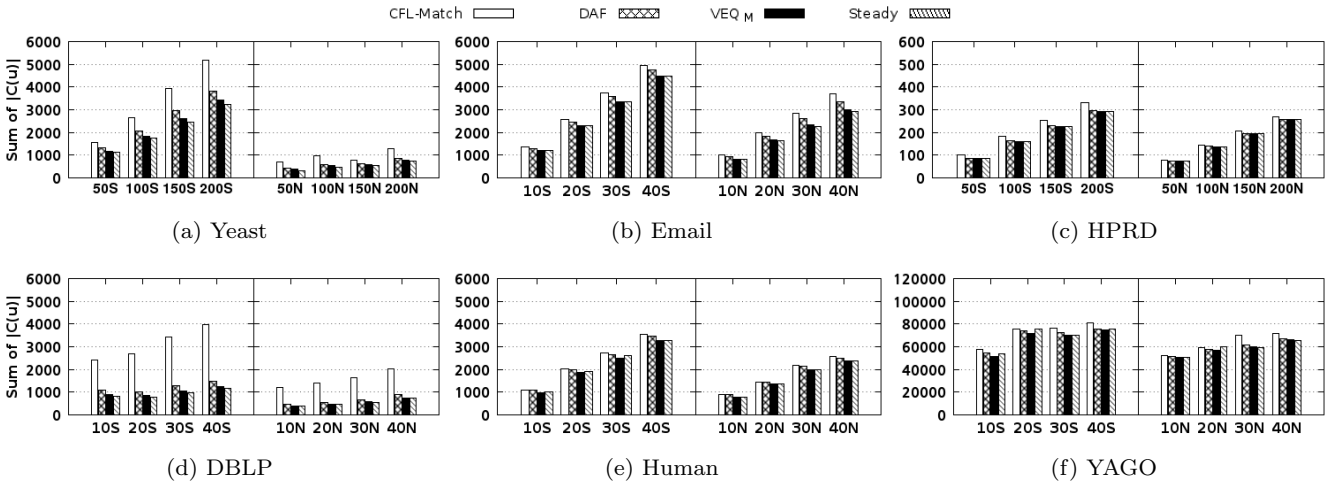
- The number of distinct labels in Σ: 10, 20, 40, 80
- A scaling factor s of a data graph in D: 2, 4, 8, 16
- The number of data graphs in D: $10^2, 10^3, 10^4, 10^5$

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 [18,39], 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 16. Grapes is unable to finish indexing data graphs with $s = 16$ or those with $|D| = 10^5$ due to excessive memory usage. VEQS 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 17. 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, VEQS is better than other algorithms in filtering out false answers, and takes a smaller portion of query processing time in verification. We observe



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

(d) DBLP (e) Human (f) YAGO

Fig. 18: Sizes of auxiliary data structures of subgraph matching algorithms

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.

7.3 Subgraph Matching

To evaluate the performance of our subgraph matching algorithm $\mathsf{VEQ_M}$, we compare $\mathsf{VEQ_M}$ with recent subgraph matching algorithms CFL-Match [3], DAF [13], RIfs [40] and GQLfs [40] from data management community, and Glasgow [28] from AI community.

Datasets. We test the algorithms against real-world datasets in Table 4, which were widely used in previous work [14,3,23,13]. 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 [24]. YAGO is an RDF dataset.

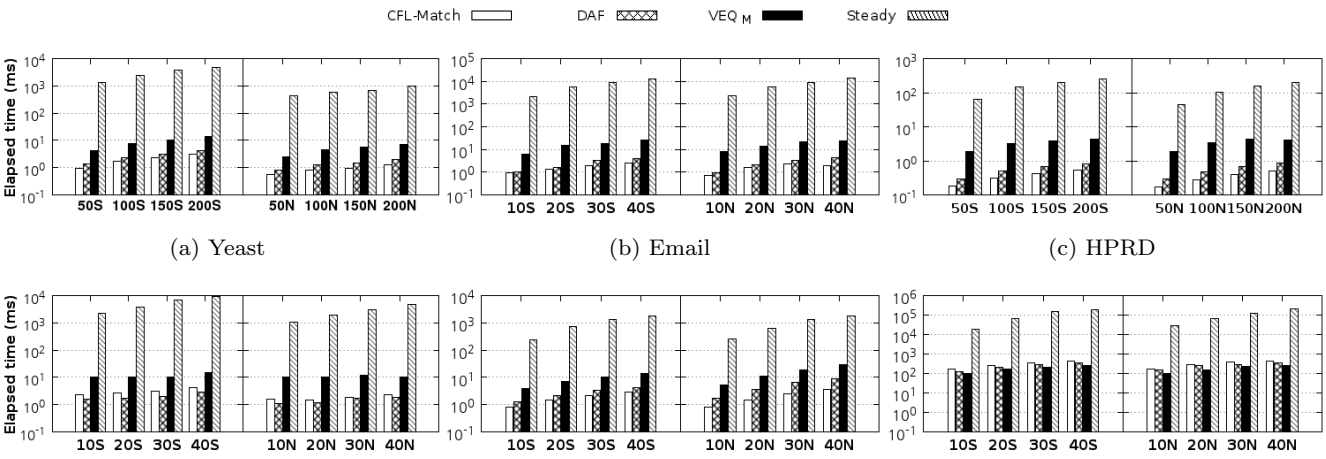
Table 4: Characteristics of real datasets for subgraph matching where Σ is a set of distinct vertex labels in G

G	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	Avg degree	$\lvert\Sigma\rvert$
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

Query Sets. We use the same experimental setting as [3] and [13]. 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 [40]. Steady repeats refining $C(u)$ to reach a *steady state*, in which for each $v \in C(u)$ and $u \in V(q)$, v satisfies the following constraint: for a neighbor u' of u, $C(u')$ and a set of v's neighbors have at least one vertex in common. Steady was used as an optimal CS in [40].

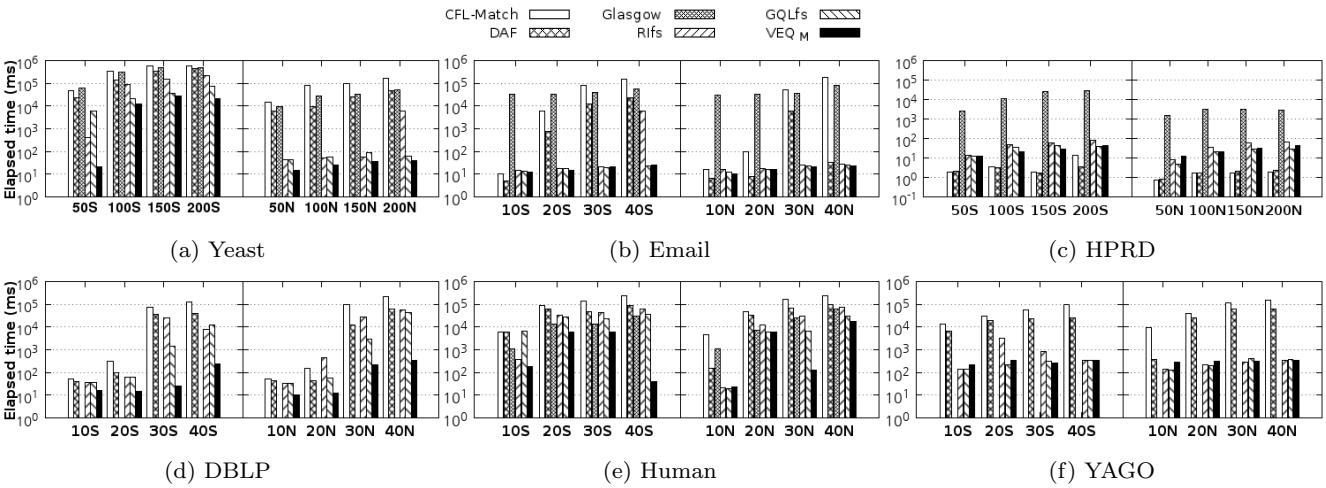
Figure 18 shows the average size of the auxiliary data structure for each algorithm and Steady. The smaller the size is, the smaller is the search space of an algo-



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

(d) DBLP (e) Human (f) YAGO

Fig. 19: Filtering (or preprocessing) time of the competing subgraph matching algorithms and Steady.



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

(d) DBLP (e) Human (f) YAGO

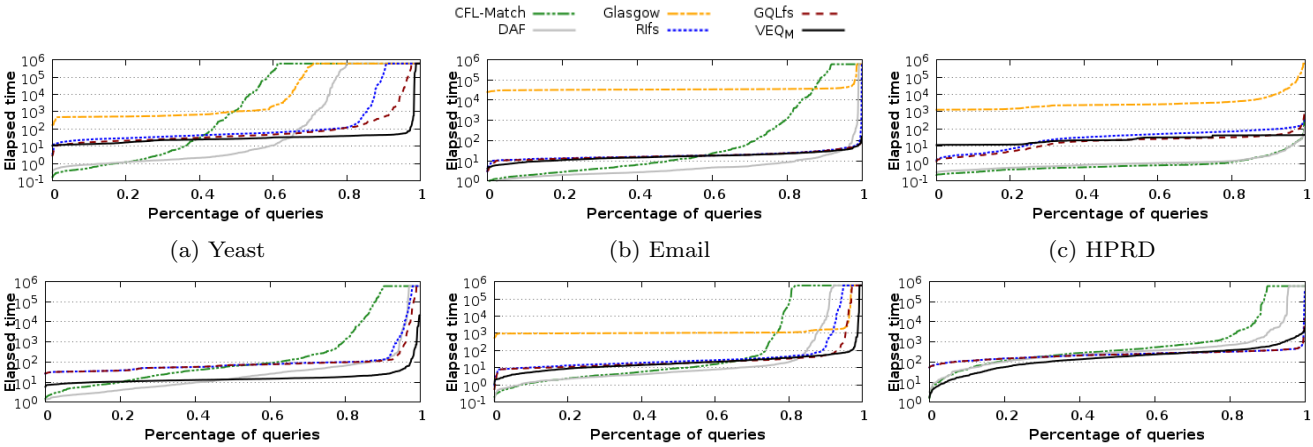
Fig. 20: Query processing time of subgraph matching algorithms on real datasets

rithm. The size of the auxiliary data structure grows as a query graph gets larger. $\mathsf{VEQ_M}$ 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. Compared to the size of CS in DAF, extended DAG-graph DP decreases the size by more than 10% in Yeast and Email, and by up to 20% in DBLP; in fact, DAF uses only simple DAG-graph DP, so for each $u \in V(q)$, $C(u)$ in CS of $\mathsf{VEQ_M}$ is a subset of that of DAF.

Figure 19 shows the filtering time of extended DAG-graph DP, CFL-Match, DAF, and Steady. On the one hand, extended DAG-graph DP usually takes slightly more time than CFL-Match or DAF due to the neighbor-safety computation, but this in turn results in more compact CS within reasonable time (< 10 ms in most cases except YAGO, and about 100 ms in YAGO). On the other hand, extended DAG-graph DP is up to more than three orders of magnitude faster than Steady (because our filtering uses refinements *three* times while Steady uses them indefinitely). As a result, the filtering method of VEQ is fast enough to be used in practice, while obtaining the size close to that of Steady.

Query Processing Time. Figure 20 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, $\mathsf{VEQ_M}$ generally outperforms GQLfs and RIfs, which is followed by DAF, CFL-Match, and Glasgow. In



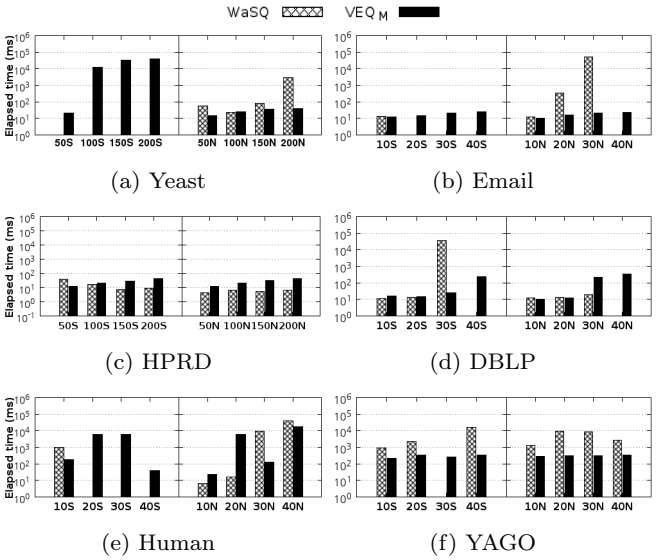
(a) Yeast (b) Email (c) HPRD

(d) DBLP (e) Human (f) YAGO

Fig. 21: Distribution of query processing time of subgraph matching algorithms on real datasets

particular, $\mathsf{VEQ_M}$ 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. $\mathsf{VEQ_M}$ 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 $\mathsf{VEQ_S}$, $\mathsf{VEQ_M}$ 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 $\mathsf{VEQ_M}$ 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 100ms, which means that they are easy instances for all the algorithms.

Figure 21 demonstrates the distribution of the query processing time of all the algorithms (the distribution is more reflective of the performance gap between queries than the geometric mean, thus only the distribution is presented here). For each algorithm, its query processing times are sorted in the ascending order so that faster (or easier) queries come earlier. For Yeast, Email, DBLP, and Human, $\mathsf{VEQ_M}$ takes slightly more time to process easy queries than CFL-Match and DAF as $\mathsf{VEQ_M}$ has the overhead of computation in filtering and pruning; however, $\mathsf{VEQ_M}$ performs better than the others for hard queries. Specifically in Yeast and Human, every algorithm reaches the time limit at a different percentage. CFL-Match first reaches the time limit, followed by Glasgow and DAF respectively, whereas $\mathsf{VEQ_M}$ barely touches the top for the rightmost few queries, followed by the runner-up GQLfs. Furthermore, $\mathsf{VEQ_M}$ does not even reach the time limit on Email and DBLP, i.e., the query processing time of $\mathsf{VEQ_M}$ is generally stable. For easy dataset HPRD, DAF and CFL-Match take less time than $\mathsf{VEQ_M}$ for most queries. In contrast, $\mathsf{VEQ_M}$ is generally steady in the query processing time whereas the running time of the other algorithms gradually increases for hard queries. In YAGO, $\mathsf{VEQ_M}$ is the fastest for a majority of queries. For hard queries, the elapsed time of $\mathsf{VEQ_M}$ gradually increases while that of the others sharply increases.



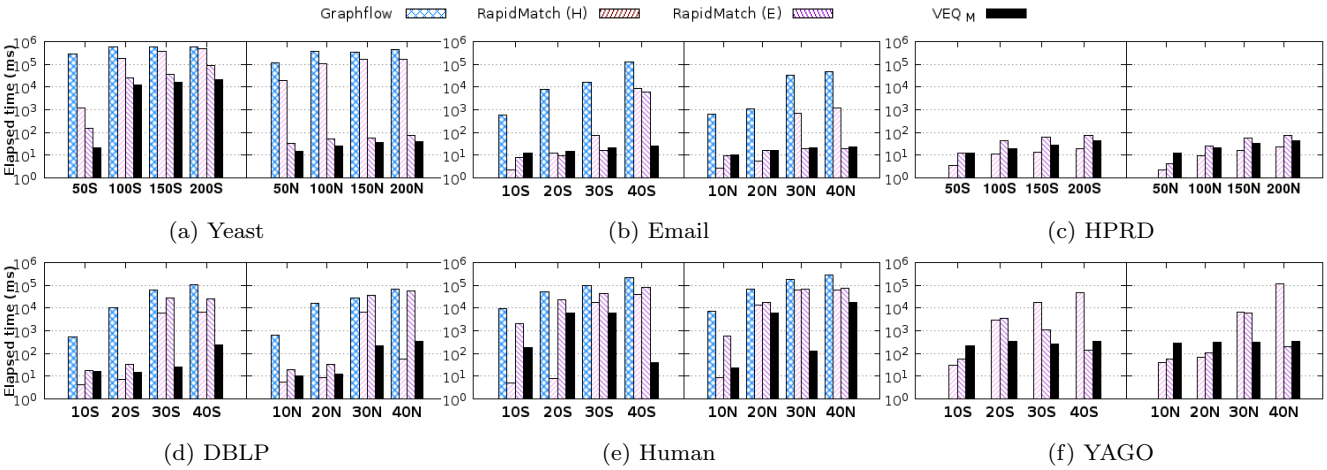
(a) Yeast (b) Email

(c) HPRD (d) DBLP

(e) Human (f) YAGO

Fig. 22: Query processing time of WaSQ ad $\mathsf{VEQ_M}$. Empty bars represent that WaSQ does not finish within the time limit.

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-



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

(d) DBLP (e) Human (f) YAGO

Fig. 23: Query processing time of VEQ_M and join-based subgraph matching algorithms on real datasets

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.

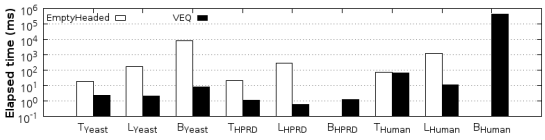


Fig. 24: Query processing time of EmptyHeaded and VEQ_M. T, L, B in the x-axis represent a triangle, lollipop, barbell query, respectively, e.g., T_{HPRD} denotes a triangle query in HPRD.

7.4 Comparison with an Workload-Aware Algorithm

We compare VEQ_M and WaSQ [25] in experiments. Note that our work and WaSQ tackle different problems. WaSQ solves *workload-aware subgraph matching*, i.e., given a query workload (a set of queries), WaSQ caches the embeddings of every query of the workload in advance, and then given a new query q, it reuses the query workload and the cached embeddings to efficiently find the embeddings of q.

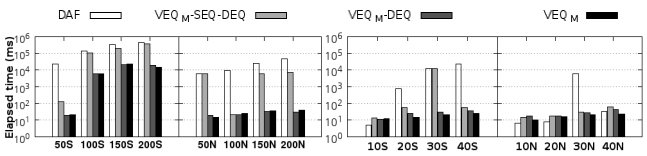
Figure 22 presents the query processing time of WaSQ and VEQ_M. As in other experiments of our work, we set a time limit of 10 minutes for each query. Since WaSQ takes a query workload as input, we set a time limit of WaSQ as 1,000 minutes for each query workload with 100 queries. The empty bars in the figure represent that WaSQ does not finish within the time limit. Although WaSQ reuses the results of previous queries and VEQ does not, the performance of VEQ_M is better than or comparable with WaSQ in most cases.

Modern graph query processing systems build the database of a given data graph in order to efficiently find matches of fixed patterns. Some systems are designed specifically for SPARQL queries or batch queries. Thus, our approach can be applied to the graph processing systems in two ways: (1) the information (e.g., neighbor label frequency or particular substructures such as triangles) found in a data graph can be computed and stored before a query graph is given, and (2) our approach can be extended to multiple query processing or RDF query processing.

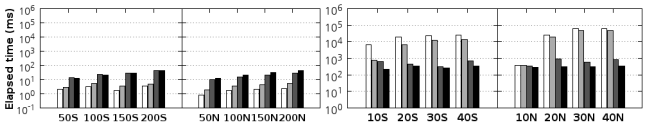
7.5 Comparison with Join-Based Algorithms

In this subsection, we compare VEQ_M against the existing join-based subgraph query engines such as EmptyHeaded [1], Graphflow [29], and RapidMatch [42]. These algorithms are derived from the join operations in DBMS, as a multi-way natural join can be represented by a graph in which an attribute and a relation correspond to a vertex and an edge, respectively. The source code of EmptyHeaded, Graphflow, and RapidMatch is publicly available at GitHub.

Figure 23 shows the query processing time of Graphflow, RapidMatch(H) (RapidMatch that finds homomorphisms), RapidMatch(E) (RapidMatch that finds embeddings), and VEQ_M. Graphflow is not included in the results of HPRD and YAGO, for which Graphflow could not generate *subgraph catalogues*. Overall, VEQ_M outperforms the others and it scales well for large graphs. In Yeast, VEQ_M consistently outperforms the others. In Human, VEQ_M consistently outperforms Graphflow and RapidMatch(E). In Email, DBLP, and YAGO, VEQ_M



(a) Yeast (b) Email



(c) HPRD (d) YAGO

Fig. 25: Query processing time of DAF and our variants for subgraph matching on real datasets

performs better than the others for large queries. RapidMatch(H) generally performs better than VEQ_M for some small queries on the datasets except Yeast, as RapidMatch(H) searches for homomorphisms that are not necessarily injective unlike embeddings (see Section 2), thus RapidMatch(H) is likely to take smaller search space to discover the first 10^5 matches of a small (or easy) query graph than VEQ_M that searches for embeddings.

Figure 24 shows the query processing time for EmptyHeaded and VEQ_M. Since EmptyHeaded runs in a docker container, VEQ was also experimented in the same container. EmptyHeaded is not optimized for large complex graph queries [1], and thus we used three representative queries (i.e., triangle, lollipop, and barbell queries) used in [1], which can be efficiently processed by the worst-case optimal join of EmptyHeaded. T, L, and B in the x-axis represent triangle, lollipop, and barbell queries, respectively, e.g., T_{HPRD} denotes that a triangle query for HPRD. Here, we run VEQ_M to find all embeddings of a query graph. VEQ_M takes less query processing time than EmptyHeaded in most cases.

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

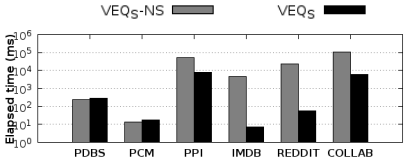


Fig. 26: Query processing time of VEQ_S-NS and VEQ_S.

Table 5: Average ratio (%) of filtering time to verification time of VEQ_S and VEQ_S-NS.

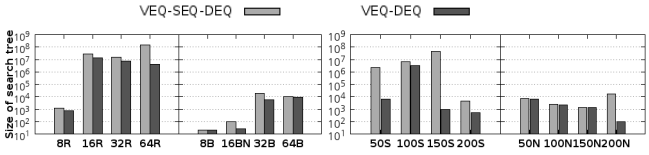
	VEQ_S-NS	VEQ_S
PDBS	89.6 : 10.4	90.7 : 9.3
PCM	63.8 : 36.2	65.8 : 34.2
PPI	11.5 : 88.5	35.4 : 64.6
IMDB	27.1 : 72.9	52.9 : 47.1
REDDIT	56.5 : 43.5	90.2 : 9.8
COLLAB	24.8 : 75.2	50.9 : 49.1

- VEQ_M-DEQ: using extended DAG-graph DP and the matching order based on static equivalence.
- VEQ_S and VEQ_M: using extended DAG-graph DP, the matching order based on static equivalence, and run-time pruning by dynamic equivalence.

Figure 25 shows the query processing time of these algorithms for subgraph matching.

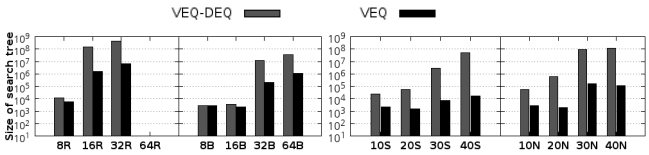
Effectiveness of Neighbor-Safety. For the subgraph matching problem, VEQ_M-SEQ-DEQ improves DAF_M by up to two orders of magnitude on Q_{50S} of Yeast, Q_{40S} Q_{30N} of Email, and Q_{100N} of Yeast. We compute the maximum $m_q = \max_{u \in V(q), l \in \Sigma} |Nbr_q(u, l)|$ for each query graph q. 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.

For subgraph search, VEQ_S improves VEQ_S-NS by up to two orders of magnitude on IMDB and REDDIT, and considerable performance gains are shown in PPI and COLLAB, though the neighbor-safety filtering slightly increases the query processing time on PDBS and PCM (see Figure 26). Note that VEQ_S-NS represents the algorithm exactly the same as VEQ_S except that the neighbor-safety filtering is turned off. We observe that whether filtering by neighbor-safety is turned on causes swings of the ratio of filtering time to verification time. Table 5 presents the average ratio of filtering time to verification time of VEQ_S and VEQ_S-NS. Obviously, VEQ_S consistently spends more portion of the query processing time in filtering than VEQ_S-NS. In particular, PPI, IMDB, and COLLAB, where the algorithms spend most time in verification, benefit



(a) PPI (b) Yeast

Fig. 27: Number of nodes in search trees of VEQ-SEQ-DEQ and VEQ-DEQ



(a) COLLAB (b) DBLP

Fig. 28: Number of nodes in search trees of VEQ-DEQ and VEQ

from filtering by neighbor-safety, since this technique contributes to considerably reducing the portion of the verification time.

Applying the neighbor-safety condition consistently decreases the false positive ratio as shown in Figure 12 and Figure 16, while slightly increasing the filtering time as shown in Figure 19. Here, the filtering method of DAF is the same as that of VEQ_S-NS.

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 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 25. 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 27 and 28).

We compare the number of search tree nodes of the matching order of DAF and those of the new matching order in Figure 27 (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 6 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 28. With dynamic equivalence turned on, the size of search space becomes consistently smaller. Table 7 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.

Statistical Analysis for Matching Order Based on Static Equivalence. Our matching order reduces search space by taking advantage of two cases, i.e., (1) $|NEC(u)| > |U_M(u)|$ and (2) $|NEC(u)| = |U_M(u)|$, in the first bullet of "New Matching Order" in Section 5. NEC does not need to be non-singleton in order to be pruned; we backtrack if a singleton NEC has no candidate vertex (i.e., when $|NEC(u)| = 1$ and $|U_M(u)| = 0$).

We count the number of these cases and measured the size of search space reduced by our matching order based on static equivalence (in Table 6, the reduced ratios are quite high in many cases). For further analysis, Table 8 summarizes the likelihood and effect of the new matching order, where symbols ">" and "=" denote the ratio (unit: %) of the number of extendable vertices u such that $|NEC(u)| > |U_M(u)|$ and $|NEC(u)| = |U_M(u)|$, respectively, among all extendable vertices $u \in V(q)$ in the search tree. "Else" denotes the ratio of the remaining extendable vertices. Note that $|NEC(u)| = |U_M(u)|$ occurs more frequently than $|NEC(u)| > |U_M(u)|$ in most cases.

The average number of search tree nodes reduced per extendable vertex u when $|NEC(u)| \geq |U_M(u)|$ is denoted by "#reduced/seq". Even though the ratio of $|NEC(u)| \geq |U_M(u)|$ is small since this occurs only on degree-one vertices, #reduced/seq is large for many cases, which leads to relatively high reduced ratios in

Table 6: Ratio (unit: %) of the number of search tree nodes reduced by the new matching order

	Yeast								PPI							
Query	50S	100S	150S	200S	50N	100N	150N	200N	8R	16R	32R	64R	8B	16B	32B	64B
Reduced	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 7: Ratio (unit: %) of the number of search tree nodes pruned by dynamic equivalence

	DBLP								COLLAB							
Query	10S	20S	30S	40S	10N	20N	30N	40N	8R	16R	32R	64R	8B	16B	32B	64B
Pruned	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

Table 8: Likelihood and effect of the new matching order. ">" and "=" denote the ratio (unit: %) of the number of extendable vertices u such that $|NEC(u)| > |U_M(u)|$ and $|NEC(u)| = |U_M(u)|$, respectively, among all extendable vertices. "Else" denotes the ratio of the remaining extendable vertices. "#reduced/seq" is the number of search tree nodes reduced per u when $|NEC(u)| \geq |U_M(u)|$

	Query	>	=	Else	#reduced/seq
Yeast	50S	0.0	2.0	98.0	35,459.3
	100S	0.4	8.8	90.8	9.7
	150S	0.2	2.4	97.4	1,597,710.7
	200S	0.1	2.8	97.0	204.7
Yeast	50N	18.5	29.0	52.4	0.4
	100N	15.0	2.7	82.3	0.8
	150N	0.0	1.6	98.4	2.9
	200N	0.0	2.8	97.2	2,736.6
PPI	8R	5.6	31.3	63.1	5.4
	16R	10.1	59.2	30.7	8.6
	32R	4.7	40.8	54.5	13.0
	64R	2.4	9.2	88.4	347.8
PPI	8B	0.6	19.2	80.2	0.1
	16B	3.5	26.8	69.7	4.8
	32B	9.6	24.4	66.0	10.8
	64B	0.1	1.3	98.6	79.9

Table 9: Likelihood and effect of dynamic equivalence. "Negative" and "Positive" denote the ratio (unit: %) of the number of unmapped extendable candidates $v \in C_M(u)$ that satisfies the first condition and the second condition, respectively, of Definition 5 in Section 6 among all unmapped extendable candidates visited in the search tree. "Else" denotes the ratio of the remaining unmapped extendable candidates. "#pruned/deq" means the number of search tree nodes pruned per negative or positive v

	Query	Negative	Positive	Else	#pruned/deq
DBLP	10S	26.6	0.6	72.8	23.5
	20S	22.6	5.4	71.9	93.0
	30S	1.6	48.2	50.2	429.6
	40S	0.4	53.0	46.7	2,650.9
DBLP	10N	38.8	0.1	61.1	32.6
	20N	21.4	12.9	65.7	533.3
	30N	0.1	59.9	40.0	394.1
	40N	0.1	64.6	35.3	613.7
COLLAB	8R	3.6	0.0	96.4	7.9
	16R	41.1	0.0	58.9	113.9
	32R	36.3	0.0	63.7	101.8
	64R				
COLLAB	8B	0.0	0.0	100.0	1.0
	16B	4.2	0.0	95.8	6.3
	32B	24.5	0.0	75.5	169.4
	64B	52.0	0.0	48.0	28.7

Table 2. Note that #reduced/seq on sparse queries is higher than that on non-sparse queries, because sparse query graphs generally have more degree-one vertices.

Statistical Analysis for Run-Time Pruning by Dynamic Equivalence. Previously Table 7 presents the size of pruned search space with high pruned ratios in most cases. For further analysis, we count the number of run-time equivalence that occurs in backtracking in order to measure the likelihood of the equivalence. Table 9 summarizes the likelihood, where "Negative" and "Positive" denote the ratio of the number of unmapped extendable candidates $v \in C_M(u)$ that satisfy the first condition and the second condition, respectively, of Definition 5 in Section 6 among all unmapped extendable candidates visited in the search tree. "Else" denotes the ratio of the remaining unmapped extendable candidates, and "#pruned/deq" means the number of search tree nodes pruned per negative or positive v.

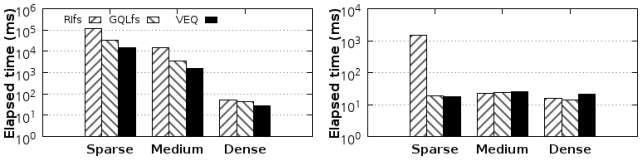
In general, run-time equivalence on unmapped $v \in C_M(u)$ is more likely to happen for larger or denser query graphs. Relatively high ratios of Negative+Positive cases combined with high #pruned/deq in Table 9 lead to very high pruned ratios in Table 7. Unlike subgraph matching on DBLP, the positive ratio is zero for subgraph search on COLLAB, where we find up to one

Table 10: Likelihood and effect of dynamic equivalence for the core structures of the query graphs of Table 9. Here, the core structures are given as the input of the experiments of Table 9.

	Query	Negative	Positive	Else	#pruned/deq
DBLP	10S	0.1	44.4	55.5	4.2
	20S	0.5	34.4	65.1	11.5
	30S	1.1	33.7	65.2	1,196.5
	40S	22.9	18.9	58.2	2,571.3
DBLP	10N	0.1	39.9	60.0	6.65
	20N	0.2	39.9	59.9	14.7
	30N	1.4	45.3	53.2	90.0
	40N	2.1	32.8	65.1	745.4
COLLAB	8R	0.01	0.0	99.99	504.4
	16R	0.1	0.0	99.88	99,067.8
	32R	2.4	0.0	97.6	696,123.4
	64R				
COLLAB	8B	0.3	0.0	99.7	17.0
	16B	6.2	0.0	93.8	5.6
	32B	19.8	0.0	80.2	2.30
	64B	15.2	0.0	84.8	62.77

Table 11: Size distribution of distinct cells and distinct equivalence sets (unit: %). "Eq" below stands for equivalence sets.

Size	COLLAB Cells	COLLAB Eq	DBLP Cells	DBLP Eq
1	77.50	12.00	73.52	7.74
2 − 10	19.25	85.17	26.13	84.75
11 − 20	2.01	1.76	0.35	7.51
21 − 30	1.14	1.09	0.001	
31 − 40	0.11	0.0002		
41 − 50	0.0003	1.95×10^{-6}		
51 − 60	1.03×10^{-6}	1.45×10^{-8}		



(a) Yeast (b) Email

Fig. 29: Varying the degree of query graphs

embedding. Hence, we can make full use of dynamic equivalence in subgraph matching.

Table 10 presents the likelihood and effect of dynamic equivalence for the *core structures* [3,41] of the query graphs of Table 9. Here, the core structures are given as the input of the Table 9 experiments. For small queries ($Q_{10S}, Q_{20S}, Q_{10N}, Q_{20N}$) of DBLP, "Positive" accounts for higher percentage than that of Table 9, which indicates that the vertices in the core structures are more likely to become positive cells. For large queries ($Q_{30S}, Q_{40S}, Q_{30N}, Q_{40N}$) of DBLP, "Positive" accounts for lower percentage. For every query set of COLLAB, "Else" cases are more than 80%, unlike Table 9. Meanwhile, for random-walk query sets of COLLAB, the number of search tree nodes pruned per positive v is by up to two or three orders of magnitudes larger than that of Table 9, which indicates that a core vertex in a random-walk query may prune more search space than a non-core vertex by using negative cells.

Size of Cells and Equivalence Sets. Candidate vertices in a cell must have the same neighbors in CS, and an equivalence set must be a subset of the cell. Then how many vertices a cell or an equivalence set generally contains in the real datasets? To answer this question, we measure the size of distinct cells and distinct equivalence sets for all the queries on COLLAB and DBLP. For COLLAB, the average sizes of a cell and an equivalence set are 2.12 and 3.23, respectively. For DBLP, their sizes are 1.62 and 3.74, respectively. The average size of an equivalence set is larger than that of a cell, because a lot of distinct subsets of a large cell becomes equivalence sets with size ranging 2-10, which are the majority among the sizes of equivalence sets, whereas 1 is the majority of cell sizes. This phenomenon is confirmed by Table 11 that shows the distribution of cells and equivalence sets in percentage. More than 70% of cells have size of 1 in both COLLAB and DBLP, while more than 80% of equivalent sets have size ranging between 2 and 10, which we can take advantage of in the run-time pruning method. The frequency of a size generally decreases as the size increases. In particular, 1.03×10^{-6}% of cells (i.e., one cell) are in the 51-60 size range, and 1.45×10^{-8}% of equivalent cells (i.e., three cells) range in size from 51 to 60, which rarely happens.

7.7 Varying Degree of Query Graphs

We conduct the performance study of query graphs with the same number of vertices but different average degrees. Figure 29 shows the query processing time of query graphs with different degrees. We extracted query graphs from each data graph. Sparse, medium, and dense query graphs have the average degree ≤ 3, between 3 and 6, and ≥ 6, respectively. The query processing time decreases as the degree grows in Yeast, but it is quite insensitive to degrees in other datasets, which can be observed in Figure 25 as well.

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.

Our approach can be applied to directed graphs. Suppose that we are given a directed query graph q and a directed data graph G. Then we regard a directed query graph as an undirected graph, from which we build a query DAG such that each edge is labeled with 1 if a direction newly assigned in the DAG matches the direction in the input query graph; 0 otherwise. From an edge "(u, v) with edge label x" in the query DAG we get a directed edge (u, v) if x is 1; a directed edge (v, u) if x is 0.

The algorithm in the paper already deals with general undirected graphs (including undirected cycles) as input. Hence it can handle cyclic directed graphs by the above transformation.

Acknowledgements

Hyunjoon Kim was partly 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)) and the research fund of Hanyang University(HY-202100000003161). Kunsoo Park was 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 the National Research Foundation of Korea(NRF) grant (No. NRF-2021R1A2B5B03001551) and Institute of Information communications Technology Planning Evaluation(IITP) grant (No.2018-0-01398).

Conflict of interest

Seok-Hee Hong works in the same university as Alan Fekete of the editorial board. Wook-Shin Han has a conflict-of-interest with Kyu-Young Whang in the editorial board.

References

1. Aberger, C.R., Lamb, A., Tu, S., Nötzli, A., Olukotun, K., Ré, C.: Emptyheaded: A relational engine for graph processing. ACM Transactions on Database Systems (TODS) **42**(4), 1–44 (2017)
2. Bhattarai, B., Liu, H., Huang, H.H.: Ceci: Compact embedding cluster index for scalable subgraph matching. In: Proceedings of the 2019 International Conference on Management of Data, pp. 1447–1462 (2019)
3. Bi, F., Chang, L., Lin, X., Qin, L., Zhang, W.: Efficient Subgraph Matching by Postponing Cartesian Products. In: Proceedings of ACM SIGMOD, pp. 1199–1214 (2016)
4. Bonnici, V., Ferro, A., Giugno, R., Pulvirenti, A., Shasha, D.: Enhancing graph database indexing by suffix tree structure. In: IAPR International Conference on Pattern Recognition in Bioinformatics, pp. 195–203. Springer (2010)
5. Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D., Ferro, A.: A subgraph isomorphism algorithm and its application to biochemical data. BMC bioinformatics **14**(7), 1–13 (2013)
6. Cannataro, M., Guzzi, P.H.: Data management of protein interaction networks, vol. 17. John Wiley & Sons (2012)
7. Carletti, V., Foggia, P., Saggese, A., Vento, M.: 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), 804–818 (2017)
8. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence **26**(10), 1367–1372 (2004)
9. Di Natale, R., Ferro, A., Giugno, R., Mongiovì, M., Pulvirenti, A., Shasha, D.: Sing: Subgraph search in non-homogeneous graphs. BMC bioinformatics **11**(1), 96 (2010)
10. Fan, W.: Graph pattern matching revised for social network analysis. In: Proceedings of ICDT, pp. 8–21 (2012)
11. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co. (1979)
12. Giugno, R., Bonnici, V., Bombieri, N., Pulvirenti, A., Ferro, A., Shasha, D.: Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. PloS one **8**(10), e76911 (2013)
13. Han, M., Kim, H., Gu, G., Park, K., Han, W.S.: Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In: Proceedings of ACM SIGMOD, pp. 1429–1446 (2019)
14. Han, W.S., Lee, J., Lee, J.H.: Turbo iso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In: Proceedings of ACM SIGMOD, pp. 337–348 (2013)
15. Han, W.S., Lee, J., Pham, M.D., Yu, J.X.: igraph: a framework for comparisons of disk-based graph indexing techniques. Proceedings of the VLDB Endowment **3**(1-2), 449–459 (2010)

16. He, H., Singh, A.K.: Graphs-at-a-time: Query Language and Access Methods for Graph Databases. In: Proceedings of ACM SIGMOD, pp. 405–418 (2008)
17. Kankanamge, C., Sahu, S., Mhedbhi, A., Chen, J., Salihoglu, S.: Graphflow: An active graph database. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 1695–1698 (2017)
18. Katsarou, F., Ntarmos, N., Triantafillou, P.: Performance and scalability of indexed subgraph query processing methods. Proceedings of the VLDB Endowment **8**(12), 1566–1577 (2015)
19. Kim, H., Choi, Y., Park, K., Lin, X., Hong, S.H., Han, W.S.: Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In: Proceedings of ACM SIGMOD, pp. 925–937 (2021)
20. Kim, J., Shin, H., Han, W.S., Hong, S., Chafi, H.: Taming subgraph isomorphism for rdf query processing. Proceedings of the VLDB Endowment **8**(11) (2015)
21. Kim, K., Seo, I., Han, W.S., Lee, J.H., Hong, S., Chafi, H., Shin, H., Jeong, G.: Turboflux: A fast continuous subgraph matching system for streaming graph data. In: Proceedings of ACM SIGMOD, pp. 411–426 (2018)
22. Klein, K., Kriege, N., Mutzel, P.: Ct-index: Fingerprint-based graph indexing combining cycles and trees. In: Proceedings of IEEE ICDE, pp. 1115–1126 (2011)
23. Lee, J., Han, W.S., Kasperovics, R., Lee, J.H.: An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. Proceedings of the VLDB Endowment **6**(2), 133–144 (2012)
24. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data (2014)
25. Liang, Y., Zhao, P.: Workload-aware subgraph query caching and processing in large graphs. In: Proceedings of IEEE ICDE, pp. 1754–1757 (2019)
26. McCreesh, C., Prosser, P., Solnon, C., Trimble, J.: When subgraph isomorphism is really hard, and why this matters for graph databases. Journal of Artificial Intelligence Research **61**, 723–759 (2018)
27. McCreesh, C., Prosser, P., Trimble, J.: Heuristics and really hard instances for subgraph isomorphism problems. In: IJCAI, pp. 631–638 (2016)
28. McCreesh, C., Prosser, P., Trimble, J.: The glasgow subgraph solver: using constraint programming to tackle hard subgraph isomorphism problem variants. In: International Conference on Graph Transformation, pp. 316–324. Springer (2020)
29. Mhedhbi, A., Salihoglu, S.: Optimizing subgraph queries by combining binary and worst-case optimal joins. Proceedings of the VLDB Endowment **12**(11), 1692–1704 (2019)
30. Park, H., Kim, M.S.: Evograph: an effective and efficient graph upscaling method for preserving graph properties. In: Proceedings of ACM SIGKDD, pp. 2051–2059 (2018)
31. Pržulj, N., Corneil, D.G., Jurisica, I.: Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. Bioinformatics **22**(8), 974–980 (2006)
32. Qiao, M., Zhang, H., Cheng, H.: Subgraph matching: on compression and computation. Proceedings of the VLDB Endowment **11**(2), 176–188 (2017)
33. Ren, X., Wang, J.: Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. Proceedings of the VLDB Endowment **8**(5), 617–628 (2015)
34. Ren, X., Wang, J.: Multi-query optimization for subgraph isomorphism search. Proceedings of the VLDB Endowment **10**(3), 121–132 (2016)
35. Rivero, C.R., Jamil, H.M.: Efficient and scalable labeled subgraph matching using sgmatch. Knowledge and Information Systems **51**(1), 61–87 (2017)
36. Sahu, S., Mhedhbi, A., Salihoglu, S., Lin, J., Özsu, M.T.: The ubiquity of large graphs and surprising challenges of graph processing. Proceedings of the VLDB Endowment **11**(4), 420–431 (2017)
37. Shang, H., Zhang, Y., Lin, X., Yu, J.X.: Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. Proceedings of the VLDB Endowment **1**(1), 364–375 (2008)
38. Snijders, T.A., Pattison, P.E., Robins, G.L., Handcock, M.S.: New specifications for exponential random graph models. Sociological methodology **36**(1), 99–153 (2006)
39. Sun, S., Luo, Q.: Scaling up subgraph query processing with efficient subgraph matching. In: Proceedings of IEEE ICDE, pp. 220–231 (2019)
40. Sun, S., Luo, Q.: In-memory subgraph matching: An in-depth study. In: Proceedings of ACM SIGMOD, pp. 1083–1098 (2020)
41. Sun, S., Luo, Q.: Subgraph matching with effective matching order and indexing. IEEE Transactions on Knowledge and Data Engineering (2020)
42. Sun, S., Sun, X., Che, Y., Luo, Q., He, B.: Rapidmatch: a holistic approach to subgraph query processing. Proceedings of the VLDB Endowment **14**(2), 176–188 (2020)
43. Ullmann, J.R.: An Algorithm for Subgraph Isomorphism. Journal of the ACM **23**(1), 31–42 (1976)
44. Wang, J., Ntarmos, N., Triantafillou, P.: Graphcache: a caching system for graph queries pp. 13–24 (2017)
45. Wang, J., Ren, X., Anirban, S., Wu, X.W.: Correct filtering for subgraph isomorphism search in compressed vertex-labeled graphs. Information Sciences **482**, 363–373 (2019)
46. Yan, X., Yu, P.S., Han, J.: Graph indexing: a frequent structure-based approach. In: Proceedings of ACM SIGMOD, pp. 335–346 (2004)
47. Yanardag, P., Vishwanathan, S.: Deep graph kernels. In: Proceedings of ACM SIGKDD, pp. 1365–1374 (2015)
48. Zhang, S., Li, S., Yang, J.: GADDI: Distance Index Based Subgraph Matching in Biological Networks. In: Proceedings of ACM EDBT, pp. 192–203 (2009)
49. Zhao, P., Han, J.: On Graph Query Optimization in Large Networks. Proceedings of the VLDB Endowment **3**(1-2), 340–351 (2010)
50. Zhao, P., Yu, J.X., Philip, S.Y.: Graph indexing: Tree+ delta>= graph. In: Proceedings of VLDB, pp. 938–949 (2007)
51. Zou, L., Chen, L., Yu, J.X., Lu, Y.: A novel spectral coding in a large graph database. In: Proceedings of EDBT, pp. 181–192 (2008)