creative commons
COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

Ph.D. DISSERTATION

Fast Graph Query Processing Algorithms Using Dynamic Programming

동적 프로그래밍을 이용한 빠른 그래프 쿼리 프로세싱 알고리즘

AUGUST 2020

DEPARTMENT OF ELECTRICAL ENGINEERING
AND COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Hyunjoon Kim

Fast Graph Query Processing Algorithms Using Dynamic Programming

동적 프로그래밍을 이용한 빠른 그래프 쿼리 프로세싱 알고리즘

지도교수 박 근 수

이 논문을 공학박사학위논문으로 제출함

2020 년 6 월

서울대학교 대학원

전기 · 컴퓨터 공학부

김 현 준

김현준의 박사학위논문을 인준함

2020 년 6 월

위 원 장	문 봉 기	(인)
부위원장	박 근 수	(인)
위 원	김 선	(인)
위 원	강 유	(인)
위 원	나 중 채	(인)

Abstract

Fast Graph Query Processing Algorithms Using Dynamic Programming

Hyunjoon Kim
Department of Electrical Engineering
and Computer Science
College of Engineering
The Graduate School
Seoul National University

Over the last several decades, a great deal of efforts have been made to develop practical solutions for NP-hard graph query processing problems because of diverse graph data publicly available. Despite such efforts, the existing algorithms still show a limited scalability in handling large and/or many graphs. In this thesis we consider three important and well-known graph query processing problems, which are subgraph query processing, subgraph matching, and supergraph search.

First, we propose fast algorithms for subgraph query processing and subgraph matching. We describe three advanced techniques including dynamic programming. Experiments on real-world and synthetic datasets show that our algorithms are faster than state-of-the-art subgrpah query processing and subgraph matching algorithms by up to orders of magnitude in terms of query processing time.

Second, we develop a fast and scalable algorithm for the supergraph search problem. We use four novel techniques including dynamic programming. Extensive experiments with real datasets show that our approach outperforms

state-of-the-art algorithms by up to orders of magnitude in terms of indexing time and query processing time.

Keywords: graph query processing; subgraph search; subgraph query processing; supergraph search; subgraph matching; subgraph isomorphism; dynamic programming; backtracking; adaptive matching order
Student Number: 2013-23112

Contents

List of Figures

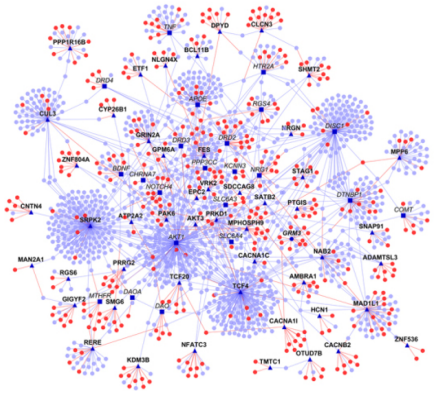
List of Tables

Chapter 1

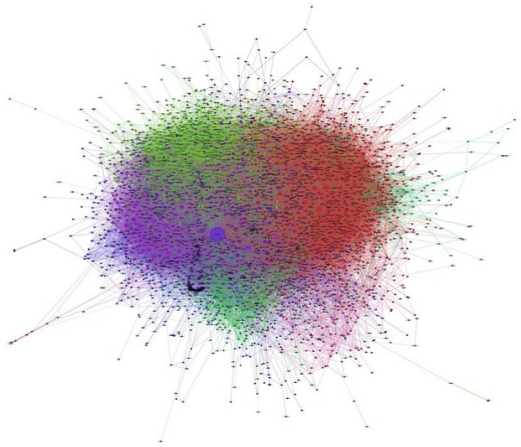
Introduction

1.1 Background

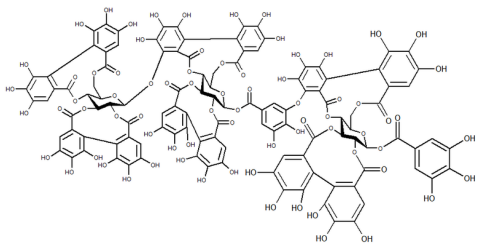
Over the last several decades, much research has been carried out on practical graph query processing as various public graph data attracted great attention in numerous application domains [67]. The public graph data can be categorized into two types depending on their applications: large graphs such as social networks and Resource Description Framework (RDF) data, and smaller graphs such as chemical compounds and protein-protein interaction (PPI) networks. On the one hand, researchers have been motivated to develop scalable and efficient algorithms to analyze the large graphs. 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, there are two fundamental graph query processing problems in smaller graphs, i.e., *subgraph search* (or *subgraph query processing*) [42, 5, 26, 74] and *supergraph search* [8, 98, 107, 11, 51]. Given a query graph Q and a set D of data graphs, subgraph search is to find all the data graphs that contain Q as subgraphs. Supergraph search is to retrieve all the data graphs that are contained



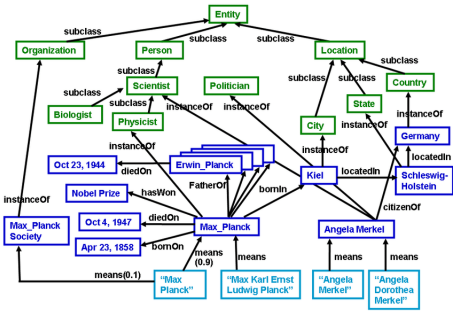
(a) Schizophrenia PPI network [23]



(b) Twitter network [47]



(c) Chemical structure of raspberry ellagitannin [87]



(d) YAGO RDF data [86]

Figure 1.1: Real-world datasets for graph query processing applications.

in Q as subgraphs.

In this study, we focus on the these graph query processing problems. They are widely present in real-world applications (see Figure 1.1) such as social network analysis [17, 72], RDF query processing [39], chemical compound search in cheminformatics [108], PPI network analysis in bioinformatics [79, 80, 7], shape matching in image processing [62, 8], and malware detection [2]. For example, given a PPI network of one species and a database of structural motifs, researchers in bioinformatics find the motifs contained in the PPI network [79, 80, 7]. In cheminformatics, molecules are modeled as undirected labeled

graphs where vertices represent atoms and edges represent chemical bonds, and supergraph search is used in the process of synthesizing new compounds with well-known chemical molecules [108]. Indeed, the National Institutes of Health (NIH) provides a web user interface for supergraph search and subgraph search of chemical compounds in an open chemistry database called PubChem[78].

These problems are NP-hard since they include finding subgraph isomorphism which is an NP-hard problem [25]. That is, solving the problems is a bottleneck in overall performance of the applications. Such applications give challenges to these problems: fast response time and good scalability for a large number of graphs and/or large-sized graphs. For instance, there are more than 96 million chemical compounds in PubChem, which also contains large molecules such as nucleotides and carbohydrates.

To address these limitations, we propose fast algorithms for subgraph query processing and subgraph matching. We describe three advanced techniques including dynamic programming. Experiments on real and synthetic datasets show that our algorithms are faster than state-of-the-art algorithms by up to orders of magnitude in terms of query processing time.

We also develop a fast and scalable algorithm for the supergraph search problem. We use four novel techniques including dynamic programming. Extensive experiments with real datasets show that our approach outperforms state-of-the-art algorithms by up to orders of magnitude in terms of indexing time and query processing time.

1.2 Organization

The remainder of the thesis is organized as follows. Chapter 2 provides problem statements and related work. Chapter 3 describes fast algorithms for subgraph search and subgraph matching. Chapter 4 gives a fast and scalable algorithm for the supergraph search problem. Finally, Chapter 5 concludes the thesis.

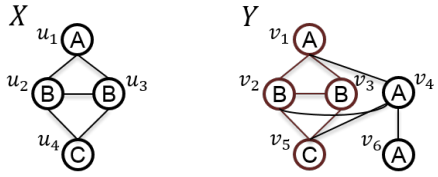
Chapter 2

Graph Query Processing

2.1 Preliminaries

We focus on undirected, connected, and labeled graphs. Our methods can be easily applied to directed or disconnected graphs with multiple labels on vertices or 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 $l_g : V(g) \cup E(g) \to \Sigma$ that assigns a label to each vertex or edge, where Σ is a set of labels (for simplicity, we consider graphs with labels only on vertices in Chapter 3). 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.

We will use the directed acyclic graph (DAG) as a tool to build an index for multiple data graphs. A DAG g' for a graph g is defined as a DAG that is built from g by assigning directions to the edges of g (e.g., g and its reverse g^{-1} in Figure 2.2 are DAGs of g_1 in Figure 4.2). A vertex is a root if it has no incoming edges while a vertex is a leaf if it has no outgoing edges. A DAG g' is a *rooted DAG* if there is only one vertex $r \in V(g')$ (i.e., root) that has no incoming edges. A vertex u' is a descendant of u if g' contains a path from u to



(a) X (b) Y

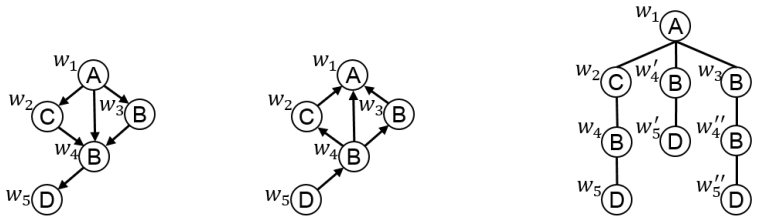
Figure 2.1: Embeddings of X in Y.

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.

The *height* of a rooted DAG g' is the maximum distance between the root and any other vertex in g', where the distance between two vertices is the number of edges in a shortest path connecting them. Let Child(u) denote a set of $u' \in V(g)$ that have incoming edges from u. Let Parent(u) denote a set of $u' \in V(g)$ that have outgoing edges to u.

Definition 2.1.1. *Given a graph $X = (V(X), E(X), l_X)$ and a graph $Y = (V(Y), E(Y), l_Y)$, an embedding of X in Y is a mapping $M : V(X) \to V(Y)$ such that (1) M is injective (i.e., $M(u) \neq M(u')$ for $u \neq u'$ in $V(X)$), (2) $l_X(u) = l_Y(M(u))$ for every $u \in V(X)$, and (3) $(M(u), M(u')) \in E(Y)$ and $l_X(u, u') = l_Y(M(u), M(u'))$ for every $(u, u') \in E(X)$.*

Given two graphs X and Y with labels only on vertices, an embedding of X in Y is the same as the above definition except that (3) $(M(u), M(u')) \in E(X)$ for every $(u, u') \in E(Y)$. Given X and Y in Figure 2.1, there are two embeddings of X in Y, i.e., $\{(u_1, v_1), (u_2, v_2), (u_3, v_3), (u_4, v_5)\}$ and $\{(u_1, v_1), (u_2, v_3), (u_3, v_2), (u_4, v_5)\}$. We say that X is *subgraph-isomorphic* to Y, denoted by $X \subseteq Y$, if there exist an embedding of X in Y. An embedding of an induced subgraph of X in Y is called a *partial embedding*. A mapping that satisfies (2) and (3) is called a *homomorphism*, i.e., it may not be injective. For the sake of traceability, we enumerate the mapping pairs in M in the order in which they are added to



(a) g (b) g^{-1} (c) Path tree of g

Figure 2.2: DAGs and path tree.

M.

The *path tree* of a rooted DAG g is defined as the tree g_T such that each root-to-leaf path in g_T corresponds to a distinct root-to-leaf path in g, and g_T shares common prefixes of its root-to-leaf paths (e.g., Figure 2.2c is the path tree of g in Figure 2.2a). For a rooted DAG g with root u, a *weak embedding* M' of g at $v \in V(Q)$ is defined as a homomorphism of the path tree of g such that $M'(u) = v$.

An unvisited (i.e., unmapped) vertex u of a DAG g in a mapping M is called *extendable* regarding M if all parents of u are matched in M. A *DAG ordering* always selects an extendable vertex as the next vertex to map.

2.2 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.

Subgraph search and subgraph matching are closely related to each other [74]. 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 [25], the two problems are NP-hard.

Supergraph Search. Given a query graph Q and a set D of data graphs, the *supergraph search problem* is to find all data graphs in D that are subgraphs of Q. That is, supergraph search is to compute the answer set $A_Q = \{g_i \in D \mid g_i \subseteq Q\}$. For example, given a set D of data graphs and a query graph Q_1 in Figure 4.2, supergraph search finds a set $A_{Q_1} = \{g_1, g_3\}$ of answer graphs, each of which is contained in Q_1 as a subgraph. Since subgraph isomorphism is NP-complete [25], the supergraph search problem is NP-hard.

2.3 Related Work

Subgraph Search. Plenty of early algorithms [92, 102, 109, 42, 15, 5, 26] for subgraph search adopted the *indexing-filtering-verification* strategy: (1) given a set D of data graphs, proper 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 do 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. These algorithms can be classified into two groups, feature mining approach and feature enumeration approach, depending on their methods to extract features [35, 74].

First, in feature mining approaches, common features frequently appeared in data graphs are extracted. gIndex [92] extracts frequent subgraphs from data graphs, and build a prefix tree from these features. Tree+Δ [102] mines frequent trees up to predetermined size, and store them as a hash table. These approaches are well-known to be costly in index construction [32, 35].

Second, all features up to a user-defined size are enumerated and indexed

in feature enumeration approaches. GCode [109] enumerates all paths, and produces vertex *signatures* in data graphs by using the paths. CT-index [42] enumerates tree and cycle features, whereas SING [15], GraphGrepSX [5], and Grapes [26] lists 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 whole search space for false graphs with no embedding found in verification; however, index construction of these approaches generally takes a great deal of time and space.

Some researchers recently selected a filtering-verification strategy without index construction. In CFQL [74], they leverage 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 the index-based filtering, CFQL outperforms other indexing-filtering-verification algorithms, benefiting from the filtering power and efficient verification technique of existing subgraph matching algorithms.

Subgraph Matching. Since Ullmann [83] introduced a backtracking method to address subgraph isomorphism, a lot of subgraph matching algorithms [13, 69, 101, 33, 99, 31, 4, 30] have been suggested based on backtracking [83, 48]. 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 backtracking is applied based on the matching order. Although these algorithms were designed based on this general framework, they vary significantly in performance, which relies on a filtering method, a matching order, and a technique to prune out a search space during backtracking.

Some early subgraph matching algorithms such as Ullmann [83], VF2 [13], QuickSI [69], and SPath [101] independently obtain each candidate set by using local filters that consider neighborhood of vertices; however, some algorithms such as GraphQL [33], $\text{Turbo}_{\text{iso}}$ [31], CFL-Match [4] and DAF [30] build auxiliary data structures on a query and data graph in order to get small candidate sets and produce effective matching orders by estimating as precise search cost as possible.

Most subgraph matching algorithms generate precomputed and fixed matching orders, i.e., global matching orders, which are exploited throughout whole search process. Although $\text{Turbo}_{\text{iso}}$ chooses a different matching order for each region in search space, the matching order is constant inside each region. SPath and DAF determine a matching order regarding each partial embedding, i.e., adaptive matching order, which dynamically selects a next query vertex to match that has the minimum expected cost.

Some algorithms eliminate redundant computations of the search process originated from the nature of backtracking. For example, DAF prunes out redundant partial embeddings that will not lead to embeddings in the future by utilizing the knowledge (i.e., failing sets) gained from past exploration in search space.

Supergraph Search. Extensive research has been done to develop efficient solutions for supergraph search. The general approach in previous work is as follows: (1) an index is constructed for a set D of data graphs, and (2) given a query graph Q, a set A_Q of answer data graphs is computed. Due to the NP-hardness of supergraph search, several existing algorithms (including CIndex [8], GPTree [98], and PrefIndex [107]) adopt the *filtering-and-verification* framework in which their indices are exploited to first filter some false answers to obtain a set of candidate graphs, and then each candidate graph is verified whether it is a subgraph of Q by a subgraph isomorphism test. However, these solutions have a significant overhead of data mining techniques (e.g., frequent subgraph mining)

to extract common substructures from data graphs in indexing. Moreover, they suffer from high cost of verification for each data graph in a candidate set.

In order to address these problems, IGQuery [11] heuristically finds some answer graphs by using its index, and then runs a filtering-and-verification method in a set of remaining data graphs to reduce the cost of both filtering and verification. First, IGQuery merges a set of data graphs into an *integrated graph (IG)* by depth-first search based on the frequency of edges in IG (without using any mining techniques) so that IG contains each data graph as a subgraph. To reduce the candidate graphs that will go through the verification, IGQuery finds a common subgraph Q' between the query graph and IG again by depth-first search, and then directly outputs the data graphs which are subgraphs of Q', i.e., direct inclusion. Then, IGQuery runs filtering and verification for the remaining data graphs as follows (a feature graph is a common subgraph of some data graphs. A set of feature graphs are found when IG is built): 1) for each feature graph, perform a subgraph isomorphism test to check whether it is a subgraph of the query graph; if it is not, filter all data graphs which contain the feature graph as subgraphs, 2) for each unfiltered data graph, run a subgraph isomorphism test with the query graph.

Unlike the existing work, DGTree [51] filters and searches at once in a single query processing algorithm. First, it constructs a tree called DGTree where each node consists of a unique feature graph and the data graphs which contain the feature graph. The feature graph of a node is always the feature graph of its parent plus one edge, and all data graphs appear as the feature graphs of leaf nodes. During DGTree construction, all (or some) embeddings (see Definition 2.1.1) of each feature graph in the data graphs are computed, which takes exponential time in the worst case. In query processing, one traverses DGTree, and at each node finds all embeddings of the corresponding feature in the query graph in order to decide whether or not to filter the data graphs that contains the feature. If there are no embeddings, the data graphs are pruned. If one

arrives at a leaf node, the data graph is added to the answer set.

Much research has also been conducted to develop the efficient solutions for the problems related to supergraph search. Subgraph matching [31, 4, 30, 58] is one of the classic problems in graph analysis. Several studies of similarity search on supergraph containment were proposed to approximately solve the supergraph search problem [70, 96]. Recently, probabilistic supergraph search has been studied to handle uncertain data graphs with probability on edges or vertices [81, 100].

Other Work. Some papers focus on comprehensive techniques that can be applied to any subgraph matching algorithm. BoostIso [64] compresses a data graph by merging similar vertices to accelerate the performance. For efficient batch query processing of subgraph matching, $\mathsf{MQO}_{\mathsf{subiso}}$ [65] computes a query execution order so that cached intermediate results can be exploited. In [58], a new matching order of query vertices (or edges) is designed specifically for a DBMS by using a set of database operators.

Chapter 3

Subgraph Query Processing and Subgraph Matching

Given a data graph G and a query graph q, the subgraph matching problem is to find all embeddings of q in G. Given a query graph q and a set D of data graphs, subgraph search (or subgraph query processing) is to find all the data graphs in D that contain q as subgraphs.

Though the subgraph search problem and the subgraph matching problem are closely related to each other, the research on each problem had been separately conducted until recently. Existing work on subgraph search [92, 102, 109, 42, 15, 5, 26] mainly adopted the *indexing-filtering-verification* strategy: (1) given a set D of data graphs, proper 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 do 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 [31, 4, 30] proposed algorithms based on a *preprocessing-enumeration* framework in Figure 3.1: an auxiliary data structure on a query graph and a data

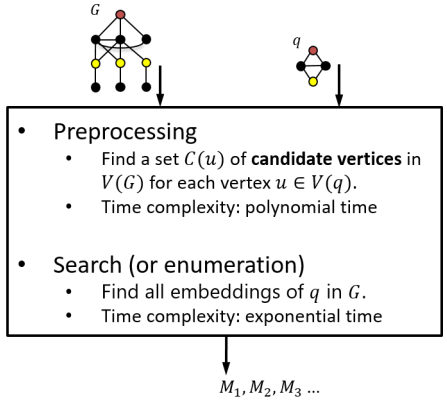


Figure 3.1: General framework of state-of-the-art subgraph matching algorithms.

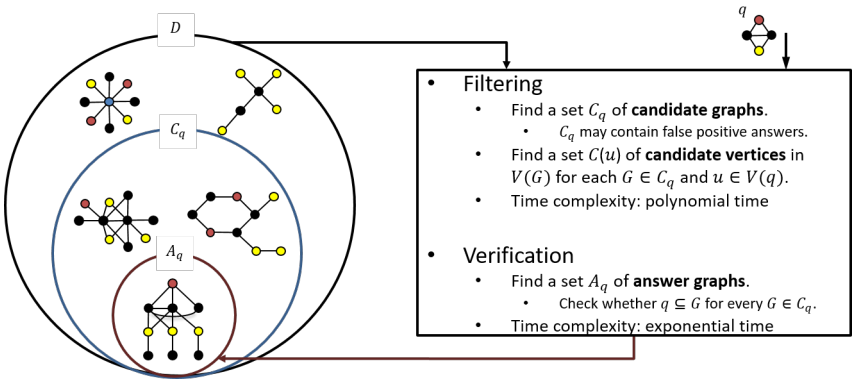


Figure 3.2: General framework of state-of-the-art subgraph search algorithms.

graph is constructed in a preprocessing stage, and all matches of the query graph are found by using the data structure in an enumeration stage (or search stage). State-of-the-art subgraph matching algorithms take polynomial time for preprocessing while they take exponential time for enumeration in the worst case. These algorithms substantially improved the query processing time by taking advantage of compact candidate sets and efficient matching orders obtained from the auxiliary data structures. Researchers recently utilized existing subgraph matching algorithms to efficiently solve the subgraph search problem [74]. They apply the preprocessing and enumeration methods of existing subgraph matching algorithms to the filtering stage and the verification stage,

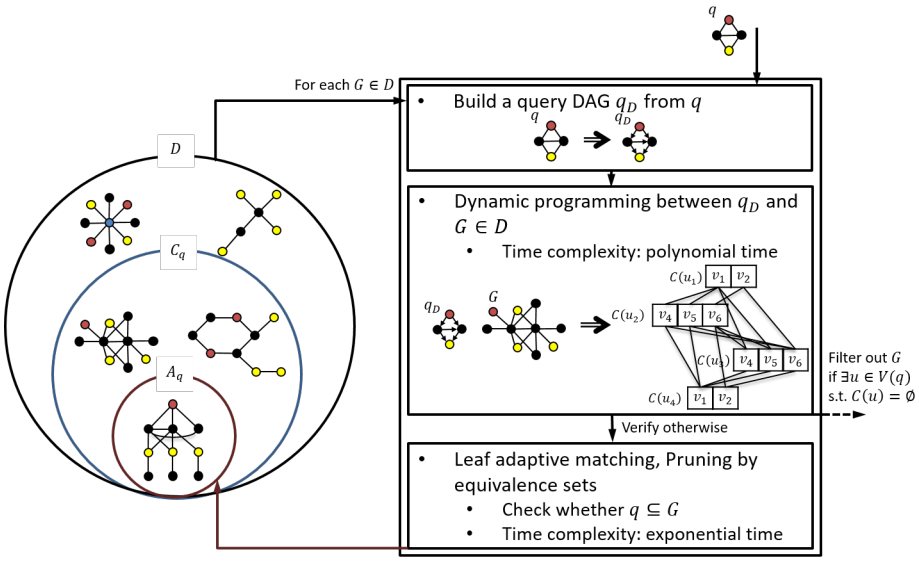


Figure 3.3: Overview of our subgraph search algorithm.

respectively, in the general subgraph search framework of Figure 3.2. (Note that filtering takes polynomial time whereas verification takes exponential time in the figure, so verification may not finish within reasonable time.) Despite its performance improvement over the existing indexing-filtering-verification approaches, it showed limited response time and scalability in dealing with large query graphs or many data graphs.

In this chapter, we introduce a new subgraph search algorithm [37] using three main techniques in order to address the limitations as in Figure 3.3. Our main contributions are the following: (1) we propose an efficient filtering method that builds a compact auxiliary data structure on q and $G \in D$ to obtain as few candidates as possible by using dynamic programming; (2) we suggest an improved adaptive matching order, i.e., leaf adaptive matching, for all query vertices, which leads to a smaller search space in the verification phase; (3) we propose a novel technique, i.e., pruning by equivalence sets, to prune out the search space by using the knowledge gained from the auxiliary data structure and the exploration on a search tree. We conduct extensive experiments on

Table 3.1: Summary of state-of-the-art subgraph matching algorithms where a query graph q and a data graph G are given.

Preprocessing		
Algorithm	Data structure	Technique
CFL-Match [4]	CPI	refinement using a spanning tree of q
DAF [30]	CS	DP between a query DAG and G
$\mathsf{ELP}_{\mathsf{SM}}$ [37]	CS	DP between a query DAG and G + filtering by neighbor-safety

Search (enumeration)			
Algorithm	Matching order	Decomposition	Pruning
CFL-Match [4]	global	core-forest-leaf	N/A
DAF [30]	adaptive	leaf	failing sets
$\mathsf{ELP}_{\mathsf{SM}}$ [37]	adaptive	N/A	failing sets, equivalence sets

several well-known real datasets as well as synthetic datasets to compare our approach with existing algorithms (we modify the state-of-the-art subgraph matching algorithm DAF [30] to run subgraph search, and also compare ours with the modified DAF). In addition, the three techniques for subgraph search in turn lead to an improved algorithm [37] 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.

Table 3.1 summarizes state-of-the-art subgraph matching algorithms where a query graph q and a data graph G are given as input. In the preprocessing stage, all the algorithms build auxiliary data structures which keep a set of candidate vertices in G for each $u \in V(q)$. While CFL-Match refines its auxiliary data structure CPI by using a spanning tree built from q, DAF and $\mathsf{ELP}_{\mathsf{SM}}$ construct auxiliary data structures CS on q and G by using dynamic programming (DP) between a query DAG of q and G. Unlike DAF, $\mathsf{ELP}_{\mathsf{SM}}$ applies a new

Table 3.2: Summary of competing subgraph search algorithms.

Algorithm	Indexing	Filtering	Verification
Grapes [26]	trie	index	VF2
CFQL [74]	N/A	Preprocessing of CFL-Match	Search of GraphQL
DAF [30]	N/A	Preprocessing of DAF	Search of DAF
$\mathsf{ELP}_{\mathsf{SS}}$ [37]	N/A	Preprocessing of $\mathsf{ELP}_{\mathsf{SM}}$	Search of $\mathsf{ELP}_{\mathsf{SM}}$

filtering technique using *neighbor-safety* to DP in order to filter out more unpromising candidate vertices of $u \in V(q)$. In the search stage, DAF and $\mathsf{ELP}_{\mathsf{SM}}$ adopt adaptive matching orders whereas CFL-Match uses a global matching order based on the core-forest-leaf decomposition, in which q is decomposed into a dense subgraph (i.e., core), a forest, and degree-one (i.e., leaf) vertices so that we first match vertices of a core, then vertices of a forest, and finally leaf vertices. Unlike the leaf decomposition strategy of DAF that always matches leaf vertices in the last, $\mathsf{ELP}_{\mathsf{SM}}$ can adaptively match leaf vertices in our matching order to reduce the search space. Moreover, we add a new technique to prune out a part of the search space by using *equivalence sets* in $\mathsf{ELP}_{\mathsf{SM}}$.

Competing subgraph search algorithms are summarized in Table 3.2. Here, we modify the subgraph matching algorithm DAF [30] to solve subgraph search (which will be called DAF). While Grapes builds a trie that consists of paths extracted from D in indexing, CFQL, DAF and $\mathsf{ELP}_{\mathsf{SS}}$ process a query graph q without the indexing stage. In the filtering stage, they build auxiliary data structures on q and each $g \in D$, and filter false answers by using the data structures, whereas Grapes performs filtering by using the trie. In the verification stage, Grapes and CFQL adopt well-known subgraph matching algorithms VF2 and GraphQL, respectively. DAF and $\mathsf{ELP}_{\mathsf{SS}}$ use the methods modified from the subgraph matching algorithms DAF and $\mathsf{ELP}_{\mathsf{SM}}$, respectively, where we find up to an embedding of q in each $G \in D$.

The rest of the chapter is organized as follows. Section 3.1 gives an overview of our approach. Section 3.2 introduces our filtering technique, Section 3.3 describes our query vertex matching order, and Section 3.4 presents a new technique to prune out a part of search space. Section 3.5 shows the experimental results of competing algorithms.

3.1 Algorithm Overview

For simplicity of presentation, we focus on undirected and connected graphs with labeled vertices. Our techniques can be easily extended to directed or disconnected graphs with labeled edges. A graph $g = (V(g), E(g), L_g)$ consists of a set $V(g)$ of vertices, a set $E(g)$ of edges, and a labeling function $L_g : V(g) \rightarrow \Sigma$ that assigns a label to each vertex where Σ is a set of labels.

Algorithm 1: SubgraphSearch

Input: a query graph q, a set D of data graphs

Output: a set A_q of answer graphs

```
1 A_q ← ∅;
2 foreach data graph G ∈ D do
3     if SubgraphIsomorphism(q, G) then
4         A_q ← A_q ∪ {G};
```

```
1 Function SubgraphIsomorphism (q, G)
2     q_D ← BuildDAG(q, G);
3     CS, isFiltered ← ConstructCandidateSpace(q, q_D, G);
4     if isFiltered then return false;
5     else return Search(q_D, CS);
```

Algorithm 1 outlines the overall procedure of our subgraph search algorithm. Given a query graph q and a set D of data graphs, we run SubgraphIsomorphism for each data graph $G \in D$. SubgraphIsomorphism returns true if q

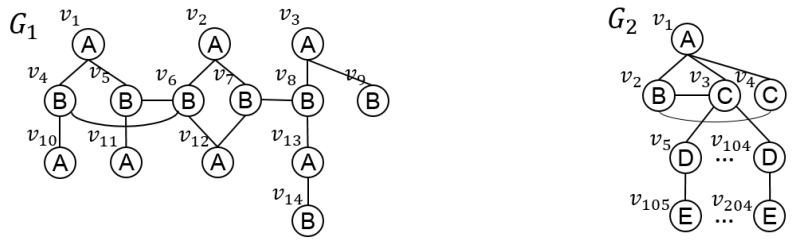


Figure 3.4: A set D of data graphs.

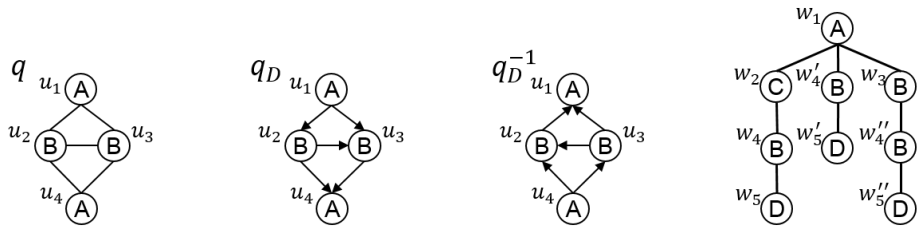


Figure 3.5: A query graph q, query DAGs q_D and q_D^{-1} built from q, and a path tree $T(q_D)$ of q_D.

is subgraph isomorphic to G; false otherwise.

SubgraphIsomorphism consists of three procedures. First, BuildDAG is invoked to build a query DAG q_D, where *query DAG* q_D is defined as a rooted 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 3.5 are query DAGs (a root of q_D is selected as in [30]). It also finds neighbor equivalence class (NEC) among all degree-one vertices in q, and merge the vertices in the same NEC to a single vertex in q_D, where NEC is a set of query vertices which have the same label and the same neighbors [31]. In Figure 3.7, a query DAG q_D is a rooted DAG built from q. Note that NEC vertex u_4 in q_D corresponds to u_4' and u_5' in q, i.e., $\text{NEC}(u_4) = \{u_4', u_5'\}$.

Then ConstructCandidateSpace is executed to build an auxiliary data structure CS on q and G. CS is an auxiliary data structure used in [30], but our CS construction is different from that of [30]. 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 3.2).

Finally, SEARCH returns true if it finds an embedding of q in G; false otherwise. It matches query vertices based on our adaptive matching order using q_D and CS (Section 3.3). Furthermore, we propose a new technique to prune repetitive search space by utilizing *equivalence sets* (Section 3.4). We also apply failing sets of [30] in our algorithm.

In order to tackle the subgraph matching problem for a query graph q and a data graph G, we run SUBGRAPHISOMORPHISM such that SEARCH is modified to find all embeddings of q in G (Section 3.4).

3.2 Filtering by Neighbor-Safety

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

Candidate Space. A *candidate space* (CS) on q and G consists of the candidate set $C(u)$ for each $u \in V(q)$, and edges between the candidates as follows:

1. 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 specified below.)

2. There is an edge between $v \in C(u)$ and $v' \in C(u')$ if and only if $(u, u') \in E(q)$ and $(v, v') \in E(G)$.

For example, Figure 3.6a shows a CS on q in Figure 3.5 and G_1 in Figure 3.4. Five candidates $v_1, v_2, v_3, v_{12}, v_{13}$ are in $C(u_1)$, and there is an edge between $v_{13} \in C(u_1)$ and $v_8 \in C(u_2)$. 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 [30] (Figure 3.5). A *weak embedding* M of a rooted DAG q with root u at $v \in V(G)$ is defined as a homomorphism of $T(q)$ such that $M(u) = v$.

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

1. There is a *weak embedding* M of a sub-DAG q_u at v (i.e., a homomorphism of $T(q_u)$ such that $M(u) = v$) in the CS.

2. Any necessary condition $h(u, v)$ (other than Condition 1) for an embedding that maps u to v is true in the CS.

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

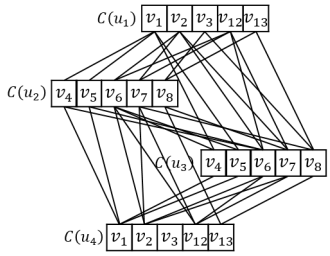
$$D[u, v] = \begin{cases} 1 & \text{if } \bigwedge_{u_c \in \text{Child}(u)} f(D[u_c, \bullet], v) \wedge h(u, v) \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

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 then applying h separately.

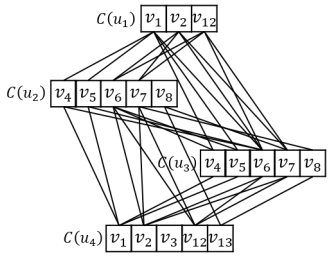
After dynamic programming, the new candidate set is computed as follows: v is in the new $C(u)$ if and only if $D[u, v] = 1$. (Note that candidate sets $C(u)$ serve as a compact representation of D.) This optimization technique will be called *extended DAG-graph DP*. Let the optimization such that $h(u, v)$ is omitted from Recurrence (3.1) be *simple DAG-graph DP*.

Now we define a necessary condition h for an embedding.

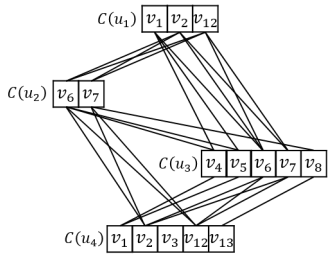
Definition 3.2.1. *For each vertex $u \in V(q)$ and a label $l \in \Sigma$, a neighbor set $Nbr_q(u, l)$ is a set of neighbors of u labeled with l. For each vertex $v \in C(u)$ and a label $l \in \Sigma$, a neighbor set $Nbr_{\mathsf{CS}}(u, v, l)$ is defined as $\cup_{u_n \in Nbr_q(u,l)} \{v_n \in C(u_n) \mid v_n$ is adjacent to $v \in C(u)$ in CS$\}$.*



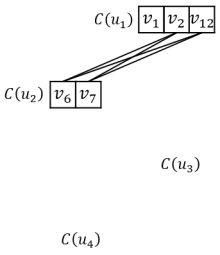
(a) Initial CS



(b) After refinement of $C(u_1)$ using neighbor-safety



(c) After refinement of $C(u_1)$ and $C(u_2)$ using neighbor-safety



(d) CS after extended DAG-graph DP

Figure 3.6: Extended DAG-graph DP over CS on q in Figure 3.5 and G_1 in Figure 3.4 using neighbor-safety.

Definition 3.2.2. *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 $l \in \Sigma$, $|Nbr_q(u,l)| \leq |Nbr_{\mathsf{CS}}(u,v,l)|$.*

For example, in a query graph q of Figure 3.5, $Nbr_q(u_1, B) = \{u_2, u_3\}$, and $Nbr_q(u_2, A) = \{u_1, u_4\}$. In CS of Figure 3.6a, $Nbr_{\mathsf{CS}}(u_1, v_3, B) = \{v_8\}$, and $Nbr_{\mathsf{CS}}(u_2, v_8, A) = \{v_3, v_{13}\}$. According to Definition 3.2.2, v_3 is not neighbor-safe regarding u_1 since $Nbr_q(u_1, B) > Nbr_{\mathsf{CS}}(u_1, v_3, B)$, whereas v_8 is neighbor-safe regarding u_2.

Lemma 3.2.1. *Suppose that we are given a CS on q and G. For each vertex $u \in V(q)$, mapping u to a candidate vertex $v \in C(u)$ which is not neighbor-safe regarding u cannot lead to an embedding of q in G.*

Proof. Suppose that there exist $u \in V(q)$ and $v \in C(u)$ not neighbor-safe re-

garding u such that mapping u to v leads to an embedding M of q in G. 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 assumption that M is an embedding because M is not injective (i.e., $M(u_i) = M(u_j)$ for $u_i \neq u_j$). Thus our assumption is incorrect, and mapping u to v that is not neighbor-safe regarding u cannot lead to an embedding of q in G. □

By Lemma 3.2.1 we define $h(u,v)$ such that $h(u,v) = 1$ if v is neighbor-safe regarding u; $h(u,v) = 0$ otherwise.

Lemma 3.2.2. *Given a* CS *on q and G, the time complexity of extended DAG-graph DP on the* CS *is $O(|E(q)| \times |E(G)|)$.*

The time complexity above includes the computation of neighbor-safety, but it is the same as the complexity of DP in [30]. 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 3.2.1. 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 2 of the Candidate Space definition. 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, where $\deg(u)$ is the degree of u.

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 [31] can remove $v \in C(u)$ such that there is a label $l \in \Sigma$ that satisfies $|Nbr_q(u, l)| > |Nbr_G(v, l)|$. To take advantage of bit operations, NLF of $V(g)$ in [4, 30] is implemented as a bit array with $|\Sigma||V(g)|$ bits where each bit represents 0 if $Nbr_g(u, l) = \emptyset$ or 1 otherwise (i.e., 1-bit lightweight NLF filter). In our approach, 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 $|Nbr_q(u, l)| > |Nbr_G(v, l)|$, which is more effective in filtering than the 1-bit lightweight NLF filter in our empirical study. Figure 3.6a illustrates an initial CS on a query graph q in Figure 3.5 and a data graph G_1 in Figure 3.4.

Since DP is executed based on a query DAG, we will use the rooted DAG q_D and its reverse q_D^{-1} (in Figure 3.5) 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. However, in Figure 3.6a, CS after one or two steps of simple DAG-graph DP is the same as the initial CS. In the third step, we perform extended DAG-graph DP using q_D^{-1}. For example, given q_D^{-1} in Figure 3.5 and CS in Figure 3.6a, we refine $C(u_1)$ first, and then refine $C(u_2)$, and so on. After the refinement of $C(u_1)$ in Figure 3.6b, v_3 and v_{13} are removed from $C(u_1)$ since they are not neighbor-safe regarding u_1. After the refinement of $C(u_2)$ in Figure 3.6c, therefore, v_8 is removed from $C(u_2)$ since there is no $v_c \in C(u_1)$ adjacent to v_8. At the same time, v_4 and v_5 are removed since they are not neighbor-safe regarding u_2.

Finally, we terminate if there exists an empty candidate set, i.e., $C(u) = \emptyset$. For example, after applying extended DAG-graph DP to Figure 3.6a, we get the CS in Figure 3.6d and there is no candidate left in $C(u_3)$, so we can be sure that there is no embedding of q. If no candidate sets are empty after multiple execution of optimization, we get the final CS. We can repeat extended DAG-graph DP with alternating q_D and q_D^{-1} until no changes occur in candidate sets,

Figure 3.7: A query graph q, and a query DAG q_D where NEC of q is merged.

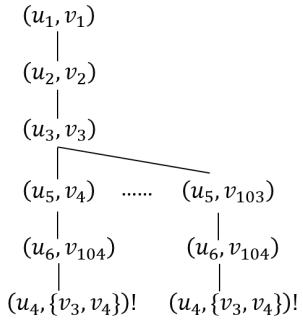
but one step of extended DAG-graph DP after two steps of simple DAG-graph DP is enough based on our empirical study.

3.3 Leaf Adaptive Matching

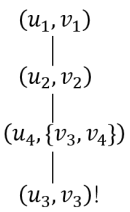
Suppose that we are trying to extend a partial embedding M in the search process. An unmapped vertex u of a query DAG q_D in M is called *extendable* regarding M if all parents of u are matched in M, and the set $C_M(u)$ of *extendable candidates* of u regarding M is defined as a set of vertices $v \in C(u)$ adjacent to $M(u_p)$ in CS for every parent u_p of u [30]). Like DAF [30], we select an extendable vertex u as the next vertex to match 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 [4, 30] adopt leaf decomposition strategy in which the vertices in a query graph are decomposed into the set of degree-one vertices and the rest, and the degree-one vertices are matched after the remaining vertices are matched. This method generally helps postponing redundant Cartesian product [4]; 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.

For example, consider a query DAG q_D of q in Figure 3.7 and a data graph G_2 in Figure 3.4. There is no embedding of q in G_2. Recall that neighbor



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



(b) Search tree of leaf adaptive matching

Figure 3.8: Search trees of two different adaptive matching orders where $(u, v)!$ means a mapping conflict (i.e, v is already matched so u cannot be mapped to v).

equivalence class (NEC) vertex u_4 in q_D corresponds to u'_4 and u'_5 in q, i.e., $\text{NEC}(u_4) = \{u'_4, u'_5\}$. The search trees in Figure 3.8 illustrate the search process. A node (u, v) corresponds to the last mapping pair of a partial embedding M, so 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 so 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 the search tree in Figure 3.8a, leaf vertices u_4 and u_6 are eventually matched after the remaining 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_5 and u_6; however, none of partial embeddings lead to embeddings. Therefore, matching u_5 and u_6 to all their extendable candidates causes huge redundant search space by postponing a mapping conflict of u_3 and u_4 at v_3.

Leaf Adaptive Matching. We propose an improved adaptive matching order to select the next extendable vertex. In our adaptive matching order, 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 that has $|\text{NEC}(u)|$ unvisited extendable candidates in $C_M(u)$, then we select u as the next vertex.
- Otherwise, we select a degree-one vertex as the next vertex if there are only degree-one extendable vertices; an extendable vertex u such that $|C_M(u)|$ is minimum otherwise [30].

Consider the search tree of leaf adaptive matching in Figure 3.8b. Given a partial embedding $M = \{(u_1, v_1), (u_2, v_2)\}$ and $C_M(u_4) = \{v_3, v_4\}$, we choose u_4 as the next vertex to match since there are $|\text{NEC}(u_4)|$ unvisited extendable candidates in $C_M(u_4)$. After we extend M to $M \cup \{(u_4, \{v_3, v_4\})\}$, 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_4 at v_3 as early as possible without matching u_5 and u_6.

3.4 Pruning by Equivalence Sets

In this section we develop a new technique to remove some parts of a search tree. Once we visit a new node (i.e., a new partial embedding) M, we next explore the subtree rooted at M and come back to node M. By utilizing a common property shared by candidates and some 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 3.4.1. *Suppose that we are given a CS on q and G. For a vertex $u \in V(q)$ and two candidate vertices v_i and v_j in $C(u)$, we say that v_i and v_j share neighbors if for every neighbor u_n of u in q, v_i and v_j have common neighbors in $C(u_n)$. Then a cell $\pi(u, v)$ is defined as a set of vertices $v' \in C(u)$ that share neighbors with v in CS.*

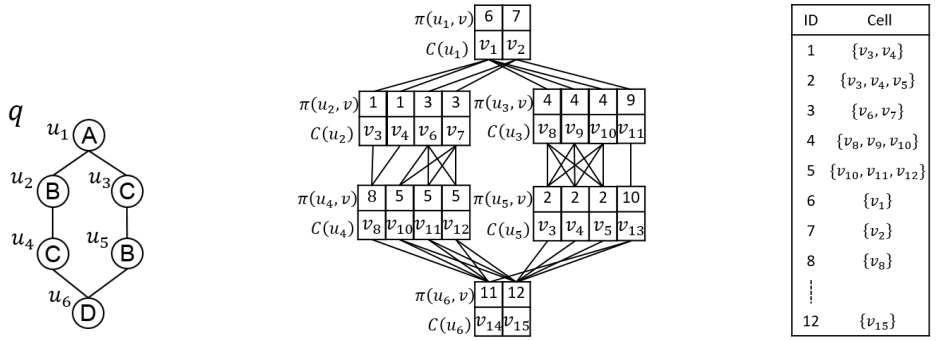


Figure 3.9: A query graph q and CS. Every cell $\pi(u, v)$ is represented as a unique ID according to a table above.

As a new running example, we use a query graph q and the CS in Figure 3.9. For example, v_3, v_4 and v_5 in $C(u_5)$ share neighbors in CS, so $\pi(u_5, v_3) = \pi(u_5, v_4) = \pi(u_5, v_5) = \{v_3, v_4, v_5\}$, while only v_3 and v_4 in $C(u_2)$ share neighbors in CS, i.e., $\pi(u_2, v_3) = \pi(u_2, v_4) = \{v_3, v_4\}$.

Assume in the rest of this section that we are given a partial embedding M, an extendable vertex $u \in V(q)$, and $v_i \in C_M(u)$ after the exploration of the subtree rooted at $M \cup \{(u, v_i)\}$. Let $\mathcal{T}_M(u, v_i)$ denote the set of embeddings in the subtree rooted at $M \cup \{(u, v_i)\}$. We will define below a set $\pi_M(u, v_i)$ of vertices $v_j \in C_M(u)$ *equivalent* to v_i, which is a subset of $\pi(u, v_i)$. We aim to avoid exploring the subtree rooted at $M \cup (u, v_j)$ in which no embeddings will be found if $\mathcal{T}_M(u, v_i) = \emptyset$; otherwise (i.e., there has been at least one embedding in $\mathcal{T}_M(u, v_i)$), obtain all embeddings in $\mathcal{T}_M(u, v_j)$ without exploring the subtree rooted at $M \cup (u, v_j)$.

Definition 3.4.2. *Let $I_M(u, v_i)$ be the set of all mappings (u', v_i) that conflict with (u, v_i) at v_i in the subtree rooted at $M \cup (u, v_i)$, and $U_M(u, v_i)$ be the set of all mappings (u', v') visited in the subtree such that $v_i \notin \pi(u', v')$. A negative cell $\pi^-_M(u, v_i)$ regarding M is $\pi(u, v_i) \cap \{\cap_{(u', v_i) \in I_M(u, v_i)} \pi(u', v_i)\}$ if there was at least one mapping conflict at v_i in the subtree; $\pi(u, v_i)$ otherwise. A*

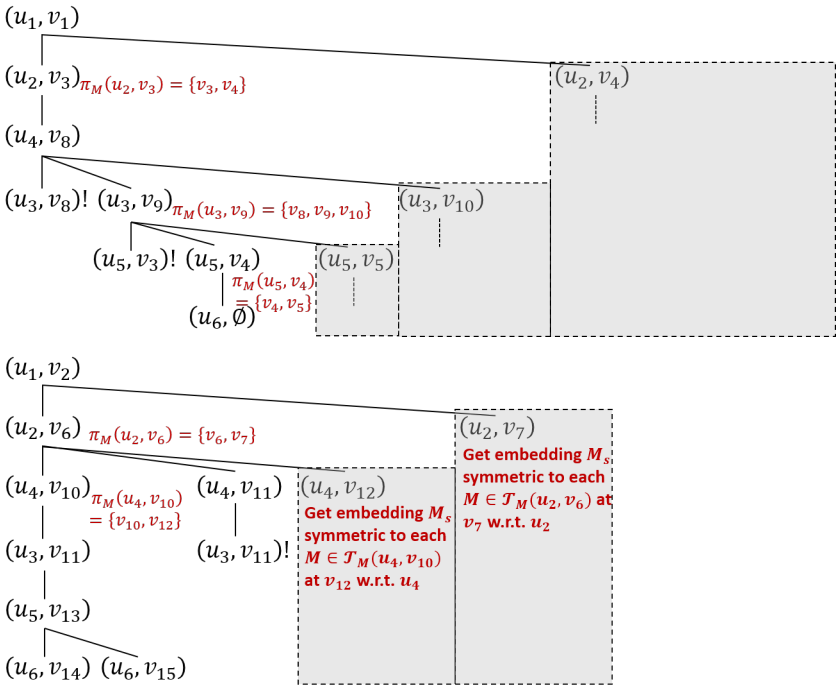


Figure 3.10: Pruned search tree. Nodes enclosed by dashed boxes are pruned by equivalence sets.

positive cell $\pi^+_M(u, v_i)$ *regarding* M *is* $\pi^-_M(u, v_i) - \delta_M(u, v_i)$ *where* $\delta_M(u, v_i) = \cup_{(u', v') \in U_M(u, v_i)} \pi(u', v')$. *The equivalence set* $\pi_M(u, v_i)$ *regarding* M *is defined as* $\pi^-_M(u, v_i)$ *if* $\mathcal{T}_M(u, v_i) = \emptyset$; $\pi^+_M(u, v_i)$ *otherwise.*

Figure 3.10 is a search tree for a query graph q and a CS in Figure 3.9. 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 $M \cup (u_3, v_9)$. An 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, so $\pi_M(u_2, v_3)$ is $\pi^-_M(u_2, v_3) = \pi(u_2, v_3) \cap \pi(u_5, v_3) = \{v_3, v_4\}$. Suppose that we just came back to node $M \cup \{(u_4, v_{10})\}$ where $M = \{(u_1, v_2), (u_2, v_6)\}$ after the exploration of

the subtree rooted at $M \cup (u_4, v_{10})$. Since there were no mapping conflicts during the exploration, $\pi^-_M(u_4, v_{10})$ is $\pi(u_4, v_{10}) = \{v_{10}, v_{11}, v_{12}\}$. However, we visited a mapping (u_3, v_{11}) such that $v_{10} \notin \pi(u_3, v_{11})$, so $\pi^+_M(u_4, v_{10}) = \pi^-_M(u_4, v_{10}) - \delta_M(u_4, v_{10}) = \{v_{10}, v_{12}\}$ 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 $\pi^+_M(u_4, v_{10})$.

Definition 3.4.3. *For every embedding $M^* \in \mathcal{T}_M(u, v_i)$, an embedding M^*_s symmetric to M^* at $v_j \in C(u)$ is $M^* - \{(u, v_i)\} \cup \{(u, v_j)\}$ if v_j is not matched in M^*; $M^* - \{(u, v_i), (u', v_j)\} \cup \{(u, v_j), (u', v_i)\}$ if v_j is matched to u' in M^*.*

Lemma 3.4.1. *If the subtree rooted at $M \cup \{(u, v_i)\}$ has no embeddings (i.e., $\mathcal{T}_M(u, v_i) = \emptyset$), $M \cup \{(u, v_j)\}$ will not lead to an embedding for each $v_j \in \pi_M(u, v_i)$. Otherwise (i.e., the subtree has at least one embedding), there exists an embedding M^*_s symmetric to every embedding $M^* \in \mathcal{T}_M(u, v_i)$ at v_j for each $v_j \in \pi_M(u, v_i)$.*

Consider the search tree in Figure 3.10 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 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 to the sibling $M \cup \{(u_2, v_4)\}$ of node $M \cup \{(u_2, v_3)\}$.

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

Search Process. MATCHING in Algorithm 2 is our search process to find all embeddings of q in the CS where the new adaptive matching order in Section 3.3 and Lemma 3.4.1 are applied. Similarly to DAF [30], it extends a partial embedding M based on DAG ordering: 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 that of DAF as follows.

On one hand, we select the next extendable vertex u among multiple extendable vertices based on our new adaptive matching order in Section 3.3 (line 3).

On the other hand, our new pruning technique is added. For every $v \in C_M(u)$, v is initialized as 'inequivalent' (line 4). Let $eq_M(u, v)$ be the vertex in $\pi_M(u, v)$ that has been already matched with u in the extension of M. For each unvisited candidate $v \in C_M(u)$, we report an embedding M^*_s symmetric to M^* at v for every embedding M^* extended from $M \cup \{(u, eq_M(u, v))\}$, and continue 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). By Definition 3.4.2, all the ancestors (u_a, v_a) of (u, v)

Algorithm 2: MATCHING(q_D, CS, M)

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

such that $v_a \notin \pi(u, v)$ should be visited in line 13, but in our implementation we visit only the ancestors such that $v_a \notin \pi(u, v)$ and $\pi(u_a, v_a) \cap \pi(u, v) \neq \emptyset$ for efficiency because $v' \in \pi(u, v)$ such that $v' \notin \pi(u_a, v_a)$ does not need to be considered to compute a positive cell $\pi^+(u_a, v_a)$ which is a subset of $\pi(u_a, v_a)$. 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 as $M \cup \{(u, v)\}$; $\pi^+_M(u, v) = \pi^-_M(u, v) - \delta_M(u, v)$ 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 $\pi^-_{M_p}(M^{-1}(v), v)$ regarding M_p (where M_p is the parent node of $(M^{-1}(v), v)$) (lines 23-24).

For subgraph search, we modify Algorithm 2 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, we do not need to find symmetric embeddings, thus for an extendable vertex u and an unvisited extendable candidate $v \in C_M(u)$ such that v is equivalent, we continue (line 8 is removed). Finally, $\pi_M(u, v)$ always represents $\pi^-_M(u, v)$ (line 17) so the computation of $\delta_M(u, v)$ is no longer needed (lines 13-14 are removed). Then we invoke MATCHING$(q_D, \mathsf{CS}, \emptyset)$ instead of SEARCH(q_D, CS) in Algorithm 1.

In our implementation, we do not need to compute $\pi(u, v)$ for every $u \in V(q)$ and $v \in C(u)$ since we may visit only some $v \in C(u)$ and terminate after finding an embedding. Hence, a cell $\pi(u, v)$ is computed not right after CS construction, but the first time when $v \in C(u)$ has become an extendable candidate such that $v \in C_M(u)$.

3.5 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++. Ex-

periments are conducted on a machine running CentOS Linux 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 graph 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 [74, 35, 30]:

- 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 candidate 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 which are solved by at least one of the competing algorithms.

- Ratio of filtering time to verification time: this ratio shows that how much filtering or verification time accounts for in query processing time.

Even though Grapes apparently spends a large amount of time and space in indexing datasets, indexing time or index size will not be considered as metrics

for the evaluation since all the other subgraph search algorithms process queries without indexing.

3.5.1 Subgraph Search

Since CFQL [74] significantly outperformed existing subgraph search algorithms, and Grapes [26] was generally the fastest in query processing among existing indexing-filtering-verification approaches [74, 35], we select these two algorithms to be compared with our subgraph search algorithm $\mathsf{ELP}_{\mathsf{SS}}$. Furthermore, we modify the state-of-the-art subgraph matching algorithm DAF [30] to solve subgraph search, and also compare $\mathsf{ELP}_{\mathsf{SS}}$ with the modified DAF (which will be called DAF in this section).

Table 3.3: Characteristics of real-world datasets for subgraph search where Σ is a set of distinct vertex labels.

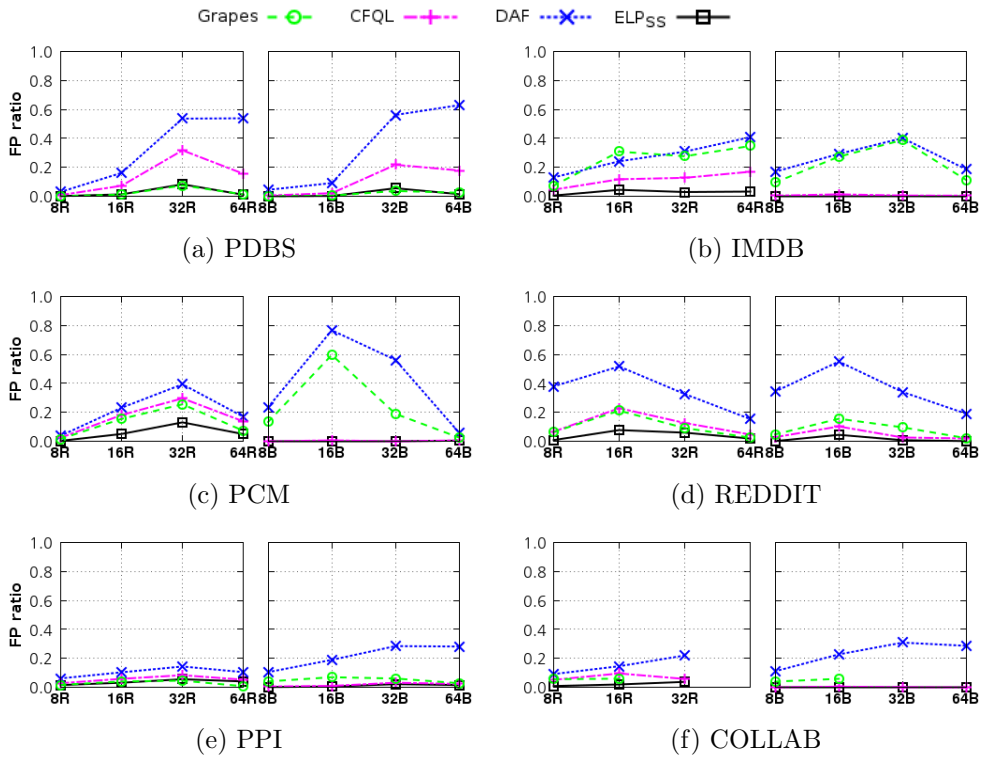
	Dataset		Average per graph			
	$\|D\|$	$\|\Sigma\|$	$\|V(G)\|$	$\|E(G)\|$	degree	$\|\Sigma\|$
PDBS	600	10	2,939	3,064	2.06	6.4
PCM	200	21	377	4,340	23.01	18.9
PPI	20	46	4,942	26,667	10.87	28.5
IMDB	1,500	10	13	66	10.14	6.9
REDDIT	4,999	10	509	595	2.34	10.0
COLLAB	5,000	10	74	2,457	65.97	9.9

Real Datasets. Experiments are conducted on real-world datasets, which are PDBS, PCM, PPI used in [26, 35, 74], and IMDB, REDDIT, COLLAB provided by [93]. 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 in which a vertex represents an actor, and an edge exists between two actors if they take

part in the same movie. 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 3.3.

Query Sets. In order to comprehensively examine the algorithms, we adopt two query generation methods similar to those in previous studies, which are random walk [35, 74] and breadth-first search (BFS) [85, 74]. 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 in the random-walk method as follows: (1) select a graph from D uniformly at random, (2) select a vertex uniformly at random from the selected graph, (3) perform a random walk from the selected vertex until we visit i distinct edges, from which we extract a subgraph with these edges. The BFS method is the same as the random-walk method except the third step: (3) perform a BFS from the selected vertex, and whenever all neighbors of a current vertex v are visited, extract edges between v and the neighbors, and edges between the neighbors and visited vertices until we extract i distinct edges. Since query graphs are generated from each dataset, they keep the same statistical characteristics as the dataset.

False Positive Ratio. Figure 3.11 presents the false positive ratio of the subgraph search algorithms on the real datasets. While DAF is the worst in filtering false answers, $\mathsf{ELP}_{\mathsf{SS}}$ is the best with average false positive ratio less than 0.1 in the most query sets. The big improvement of the false positive ratio originates from extended DAG-graph DP that utilizes neighbor-safety. Between Grapes and CFQL, Grapes is better than CFQL at filtering in PDBS (which is the sparsest among the datasets and has a small number of distinct labels per graph), whereas CFQL is more powerful in IMDB (which is dense and has a small number of distinct labels); in fact, IMDB has the largest average den-



(a) PDBS (b) IMDB

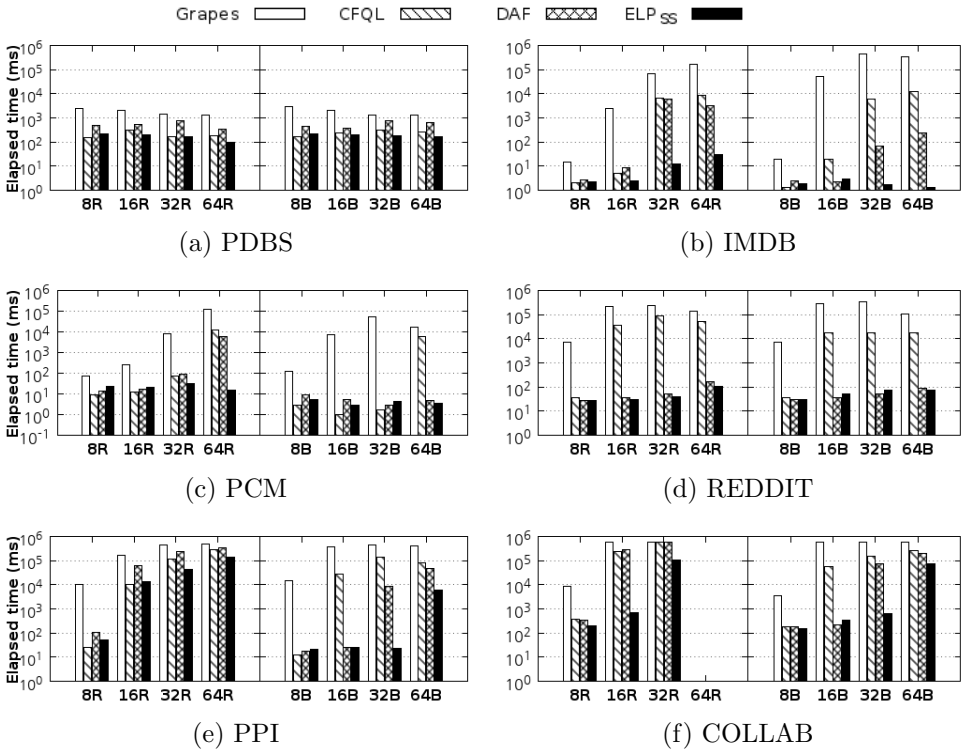
(c) PCM (d) REDDIT

(e) PPI (f) COLLAB

Figure 3.11: False positive ratio of the subgraph search algorithms on the real datasets.

sity, i.e., $\frac{2|E(G)|}{|V(G)|(|V(G)|-1)}$. For the remaining datasets, the false positive ratio of Grapes is lower than that of CFQL on random-walk query sets while that of CFQL is lower on BFS query sets. These phenomena may stem from the different strengths of the two algorithms: (1) Grapes extracts all paths of up to a fixed length from a query graph where path features are effective in filtering for a path-like query, and (2) CFQL is good at filtering candidates of a query vertex with a high degree by checking all edges with its neighbors.

Query Processing Time. Figure 3.12 shows the average query processing time of the algorithms (no algorithm finishes a query in Q_{64R} of COLLAB within the time limit). ELP_{SS} is generally the fastest except 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 leaf adaptive matching shrunk by



(a) PDBS (b) IMDB

(c) PCM (d) REDDIT

(e) PPI (f) COLLAB

Figure 3.12: Query processing time of the subgraph search algorithms on the real datasets.

equivalence sets. $\mathsf{ELP_{SS}}$ is up to two orders of magnitude faster than DAF, and up to three orders of magnitude faster than CFQL. $\mathsf{ELP_{SS}}$ outperforms Grapes up to five orders of magnitude in Q_{32B} and Q_{64B} of IMDB. However, the query processing time of $\mathsf{ELP_{SS}}$ 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, so exploiting extended DAG-graph DP or the pruning technique of $\mathsf{ELP_{SS}}$ may incur an overhead instead.

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 $\mathsf{ELP_{SS}}$ 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 exponentially grows as the size of a

query graph increases on the rest in Figure 3.12. Spikes in the query processing time of large queries are also observed in the results of CFQL for all the datasets other than PDBS. However, DAF takes nearly constant query processing time in REDDIT as well as PDBS, and shows more stable performance than CFQL and Grapes in the rest. The query processing time of $\mathsf{ELP}_{\mathsf{SS}}$ remains steady as the size of a query graph increases in all the datasets except PPI and COLLAB. The spikes in query processing time of $\mathsf{ELP}_{\mathsf{SS}}$ are generally shown in the results for large query graphs of PPI and COLLAB.

Table 3.4: Average ratio of filtering time to verification time (%).

	Grapes	CFQL	DAF	$\mathrm{ELP}_{\mathrm{SS}}$
PDBS	2.2 : 97.8	89.9 : 10.1	96.8 : 3.2	92.0 : 8.0
PCM	1.3 : 98.7	45.0 : 55.0	72.9 : 27.1	85.7 : 14.3
PPI	0.01 : 99.99	18.7 : 81.3	28.0 : 72.0	43.9 : 56.1
IMDB	2.6 : 97.4	21.1 : 78.9	36.9 : 63.1	67.6 : 32.4
REDDIT	0.2 : 99.8	21.9 : 78.1	94.0 : 6.0	95.3 : 4.7
COLLAB	0.7 : 99.3	19.2 : 80.8	34.1 : 65.9	49.9 : 50.1

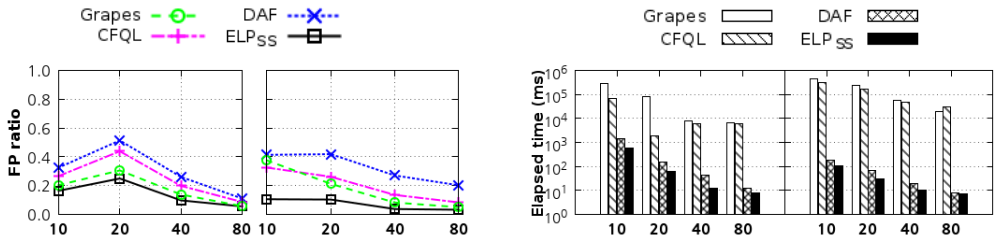
These results originate from a ratio of filtering time to verification time as shown in Table 3.4. For all the competing algorithms, verification takes exponential time in the worst case whereas filtering takes polynomial time. Indeed, in Table 3.4, Grapes spends the most of query processing time verifying 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 makes up over 60% of its query processing time on PPI, IMDB and COLLAB, the ratio of verification time consistently smaller than that of CFQL. Unlike the other algorithms, $\mathsf{ELP}_{\mathsf{SS}}$ spends the verification time less than or comparable with the filtering time, which confirms its steadier performance than the others.

Sensitivity Analysis. We evaluate the algorithms by varying several characteristics of a set D of synthetic data graphs. We generate each data graph $G \in D$ by upscaling an input data graph of PPI with 2008 edges using Evograph [63], and assign labels to vertices based on a power law distribution. We vary following parameters:

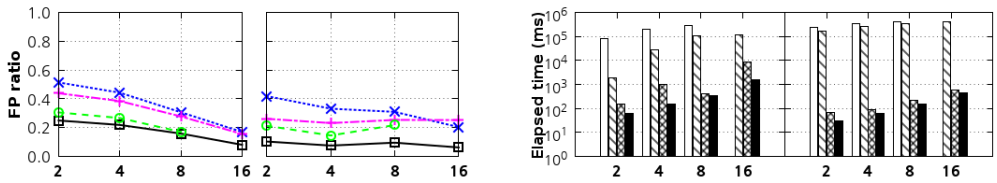
- The number of distinct labels in Σ: 10, 20, 40, 80
- A scaling factor s of a data graph in D: 2, 4, 8, 16
- The number of data graphs in D: $10^2, 10^3, 10^4, 10^5$

where s indicates that $|E(G)|$ is s times as many as 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 [35, 74], 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.

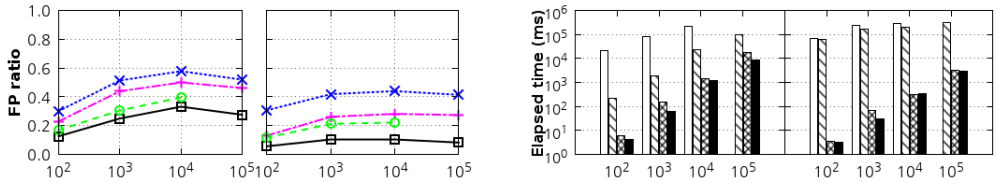
False Positive Ratio. False positive ratio of the algorithms on the synthetic datasets is displayed in the left column of Figure 3.13. Since Grapes is unable to finish indexing data graphs with $s = 16$ or those with $|D| = 10^5$ due to excessive memory usage, we compare only the remaining three algorithms on these data graphs. $\mathsf{ELP}_{\mathsf{SS}}$ consistently outperforms the others regarding false positive ratio. A runner-up in most cases is Grapes, followed by CFQL. Overall, the false positive ratio falls 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 falls especially for the random-walk query sets as the size of data graphs (i.e., a scaling factor s) gets larger. Larger data graphs are more likely to contain a query graph as a subgraph, thus the false positive ratio can decrease as the number of candidate graphs increases. Since



(a) Varying $|\Sigma|$



(b) Varying a scaling factor s



(c) Varying $|D|$

Figure 3.13: False positive ratio and query processing time on the synthetic datasets. The results of Q_{16R} and Q_{16B} are shown in the left and right, respectively, of each figure.

the algorithms filter each data graph independently, varying $|D|$ should not affect their filtering power; indeed, except processing the random-walk query set on a small number of data graphs, i.e., $|D| = 100$, the false positive ratio remains relatively constant for different numbers of data graphs.

Query Processing Time. Query processing time of the algorithms on the synthetic datasets is shown in the right column of Figure 3.13. The order of the algorithms from the fastest to the slowest is ELP$_{SS}$, DAF, CFQL, and Grapes. The query processing time falls 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. BFS queries with high-degree vertices generally take less query processing time than random-walk queries for $\mathsf{ELP}_{\mathsf{SS}}$ and DAF since they take advantage of the high degree of query vertices in filtering.

Table 3.5: Average ratio of filtering time to verification time for the random-walk query sets (where s = scaling factor).

		Grapes	CFQL	DAF	ELP_{SS}
$\|\Sigma\|$	10	0.01 : 99.99	0.2 : 99.8	14.3 : 85.7	24.6 : 75.4
	20	0.02 : 99.98	2.9 : 97.1	51.1 : 48.9	70.1 : 29.9
	40	0.1 : 99.9	0.1 : 99.9	42.7 : 57.3	84.2 : 15.8
	80	0.1 : 99.9	0.1 : 99.9	97.2 : 2.8	98.5 : 1.5
s	2	0.02 : 99.98	2.9 : 97.1	51.1 : 48.9	70.1 : 29.9
	4	0.01 : 99.99	0.2 : 99.8	9.9 : 90.1	47.5 : 52.5
	8	0.02 : 99.98	0.2 : 99.8	58.8 : 41.2	61.0 : 39.0
	16		0.3 : 99.7	8.2 : 91.8	42.3 : 57.7
$\|D\|$	10^2	0.01 : 99.99	0.8 : 99.2	78.9 : 21.1	92.8 : 7.2
	10^3	0.02 : 99.98	2.9 : 97.1	51.1 : 48.9	70.1 : 29.9
	10^4	0.1 : 99.9	1.4 : 98.6	67.8 : 32.2	33.3 : 66.7
	10^5		3.7 : 96.3	37.1 : 62.9	65.5 : 34.5

The explanations above are confirmed by Table 3.5, which shows the average ratio of filtering time to verification time on each synthetic dataset. CFQL and DAF spends most of the query processing time in verification. The ratio of the verification time of DAF and $\mathsf{ELP}_{\mathsf{SS}}$ decreases as $|\Sigma|$ increases. As in Table 3.4, $\mathsf{ELP}_{\mathsf{SS}}$ spends shorter time in verification than the others for most datasets in

Table 3.5.

To summarize, $\mathsf{ELP}_{\mathsf{SS}}$ is better than other algorithms in filtering out false answers, and takes a smaller portion of query processing time in verification. We observe in the experiments that verification generally takes more time in a false positive answer than an answer, because an algorithm has to explore the whole search space to verify that there are no embeddings in the false positive graph while terminating as soon as it finds an embedding in the answer graph. Therefore, a smaller number of false positive answers results in fewer attempts to explore the whole search space of false positive graphs. Nevertheless, finding an embedding in an answer graph can sometimes cost a lot in the verification phase. Hence a more advanced verification technique can quickly find an embedding of a query graph in each answer graph by avoiding frequent backtracking. Consequently, lowering false positive answers (by extended DAG-graph DP) and reducing search space (by leaf adaptive matching and equivalence sets) lead to shorter verification time, resulting in the improvement of overall performance.

3.5.2 Subgraph Matching

To evaluate the performance of our subgraph matching algorithm $\mathsf{ELP}_{\mathsf{SM}}$, we compare $\mathsf{ELP}_{\mathsf{SM}}$ with two state-of-the-art algorithms CFL-Match [4] and DAF [30].

Datasets. We test the algorithms against real datasets in Table 3.6, which were widely used in previous work [31, 4, 48, 30]. Yeast, HPRD and Human are protein-protein interaction networks. Email communication network and DBLP collaboration network are obtained from Stanford Large Network Dataset Collection [50]. As no label information is available for Email and DBLP, we randomly assigned a label out of 20 distinct labels to each vertex. YAGO is an RDF dataset, so we transformed it into a graph dataset by applying the *type-aware transformation* proposed in [39].

Query Sets. We use the same experimental setting as [4] and [30]. We generate

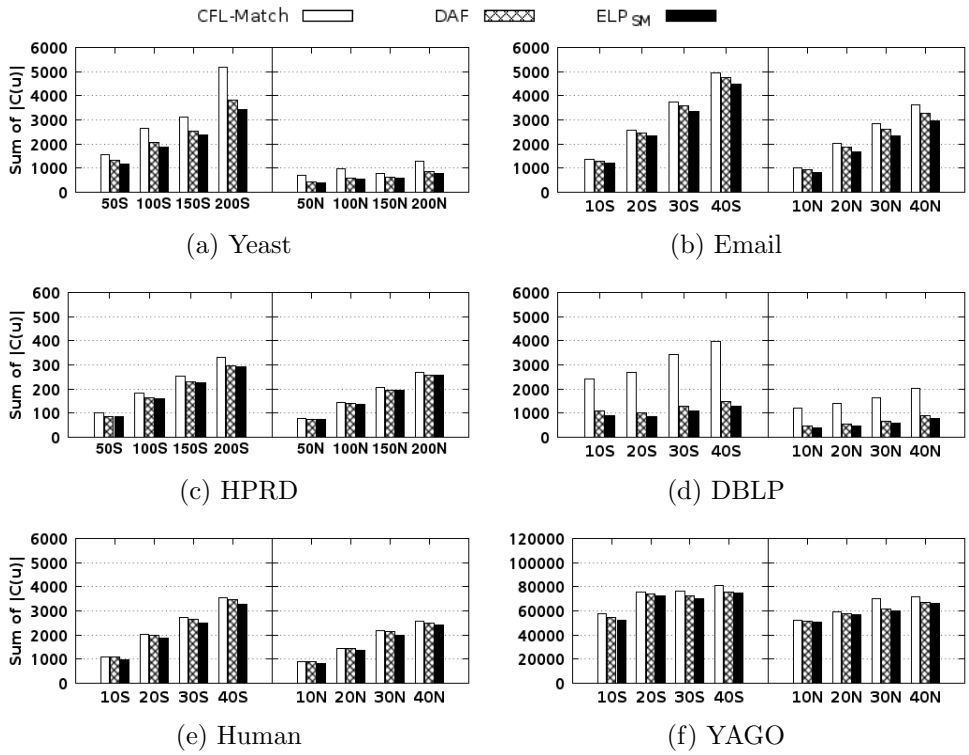
Table 3.6: Characteristics of real-world datasets for subgraph matching where Σ is a set of distinct vertex labels in G.

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

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. Figure 3.14 displays the average size of the auxiliary data structure for each algorithm. The smaller the size is, the smaller the search space of an algorithm is. The size of the auxiliary data structure grows as a query graph gets larger. $\mathsf{ELP}_{\mathsf{SM}}$ consistently has a smaller number of candidates than DAF or CFL-Match, thanks to extended DAG-graph DP. Compared to the size of CS in DAF, extended DAG-graph DP decreases the size by more than 10% in Yeast, Email and DBLP; in fact, DAF uses only simple DAG-graph DP, so for each $u \in V(q)$, $C(u)$ in CS of $\mathsf{ELP}_{\mathsf{SM}}$ is a subset of that of DAF.

Query Processing Time. Figure 3.15 shows the average query processing time of the algorithms. Due to the three main techniques described in the



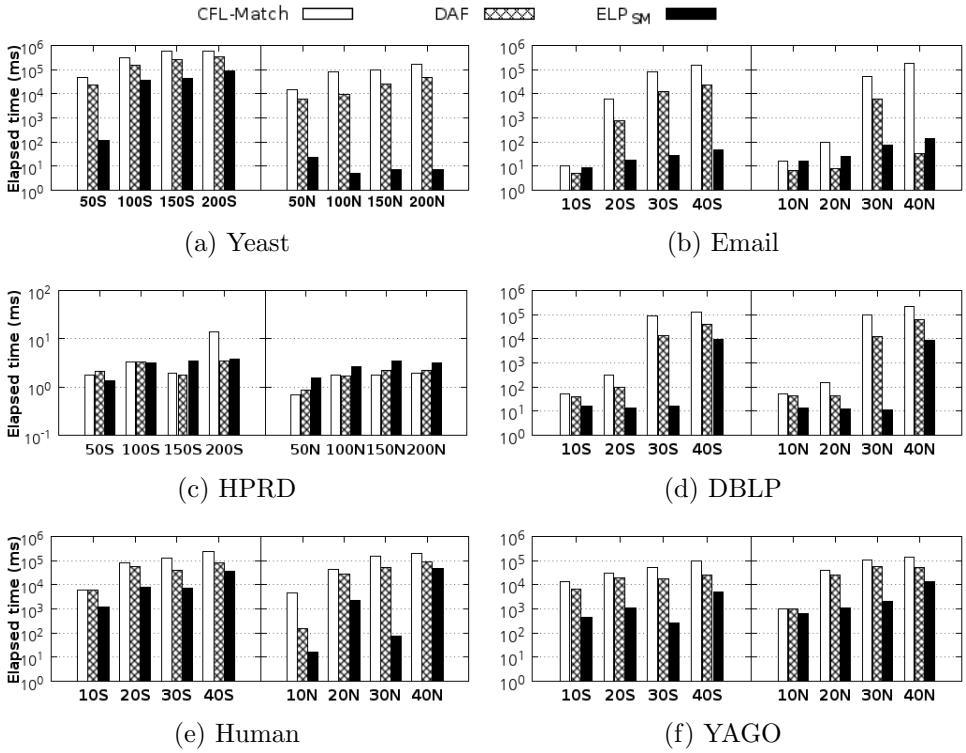
(a) Yeast (b) Email

(c) HPRD (d) DBLP

(e) Human (f) YAGO

Figure 3.14: Sizes of auxiliary data structures of the subgraph matching algorithms.

previous sections, $\mathsf{ELP_{SM}}$ outperforms DAF which is followed by CFL-Match. In particular, $\mathsf{ELP_{SM}}$ is more than four orders of magnitude faster than CFL-Match in Q_{100N}, Q_{150N} and Q_{200N} of Yeast, and more than three orders of magnitude faster than DAF in Q_{30N} of Yeast and DBLP. Different from $\mathsf{ELP_{SS}}$, $\mathsf{ELP_{SM}}$ searches a data graph for multiple embeddings, so it can output numerous symmetric embeddings at once by using equivalence sets. However, the query processing time of $\mathsf{ELP_{SM}}$ 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, so most queries of HPRD finishes within 10ms, which means that they are easy instances for all the algorithms. Therefore lightweight



(a) Yeast (b) Email

(c) HPRD (d) DBLP

(e) Human (f) YAGO

Figure 3.15: Query processing time of the subgraph matching algorithms on real datasets.

simple approaches may take shorter time for these queries.

Since we find a large 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. On these datasets, CFL-Match and DAF spend 99.5-99.999% and 98.9-99.999%, resepctively, of query processing time in search. However, search accounts for 75.6-99.98% in $\mathsf{ELP}_{\mathsf{SM}}$, which consistently has a smaller percentage than the others for each dataset. As a result, our strategy to obtain compact candidate sets and to reduce search space gives rise to an efficient subgraph matching algorithm.

Chapter 4

Supergraph Search

Given a query graph Q and a set D of data graphs, supergraph search is to retrieve all the data graphs in D that are contained in Q as subgraphs. Most supergraph search algorithms adopted the indexing-filtering-verification framework in Figure 4.1. (1) From a set D of data graphs, the algorithms construct indexes before processing queries, (2) given a query graph q, the data graphs not contained in q as a subgraphs are filtered out by using the indexes in a filtering phase, and (3) a subgraph isomorphism test is performed against every remaining candidate graph in a verification phase.

Although various techniques have been exploited in the existing solutions, even the state-of-the-art algorithms (i.e., IGQuery and DGTree) show several limitations to efficiently handle real-world data.

First, index construction is computationally expensive in some existing work. Frequent subgraph mining or its variants are commonly used to extract common subgraphs shared by many data graphs in the process of building indices [91, 8]; however, these data mining techniques are costly and not scalable to deal with large data graphs. DGTree searches the data graphs for embeddings of a feature graph corresponding to every node to build the tree-structured in-

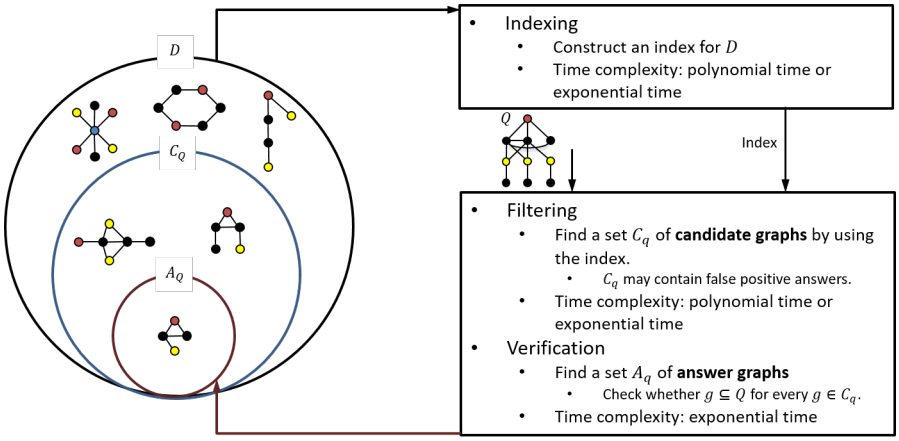
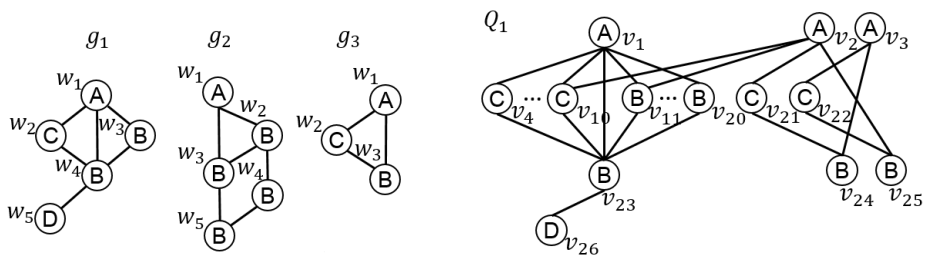


Figure 4.1: General framework of existing supergraph search algorithms.

dex, which takes exponential time in the worst case.

Second, filtering methods in some previous work are costly, which may degrade overall query processing performance. For example, in the feature-based filtering process of IGQuery, one searches the query graph for an embedding of every feature graph (i.e., a subgraph isomorphism test), and thus it also takes exponential time.

Third, search methods in the existing algorithms can cause redundant computations. For example, given a set D of data graphs and a query graph Q_1 in Figure 4.2, we can find a partial embedding $M_1 = \{(w_1, v_1), (w_2, v_4), (w_4, v_{23})\}$ of g_1 in Q_1 and $M_3 = \{(w_1, v_1), (w_2, v_4), (w_3, v_{23})\}$ of g_3 in Q_1, where M_1 and M_3 overlap in Q_1. However, in IGQuery, once a set of candidate graphs is obtained by the direct inclusion and feature-based filtering, every candidate graph is verified by a subgraph isomorphism test; thus, an embedding of every candidate graph is independently computed although (partial) embeddings of some candidate graphs in Q may overlap each other. The DGTree traversal may spend redundant search space to find all partial embeddings of each answer graph. For example, DGTree finds 7×10 partial embeddings of g_1 in Q_1, i.e., $M_1 = \{(w_1, v_1), (w_2, v_{4-10}), (w_3, v_{11-20}), (w_4, v_{23})\}$, each of which can extend to an embedding, but finding one of them is enough to solve this problem.



(a) A set D of data graphs (b) A query graph

Figure 4.2: Data graphs and a query graph.

Contributions. To address the limitations above, we introduce the following new ideas, which lead to a faster and scalable algorithm IDAR [38] for supergraph search.

First, we propose an efficient index construction method called *DAG integration*, in which we build a data DAG from each data graph and merge data DAGs into an *integrated DAG* (IDAG) I. DAG integration has following advantages over existing work. Unlike DGTree and some indexing methods based on frequent subgraph mining, DAG integration takes polynomial time. Compared to the depth-first search of IGQuery, we can compactly merge a data DAG into I in a topological order of the DAG, guided by the *similarities* between vertices in the data DAG and I.

Second, we propose an auxiliary data structure called *integrated candidate space* (ICS), which will serve as a complete search space to find all embeddings of data graphs in Q. By applying dynamic programming between IDAG and graph during ICS construction, we can efficiently filter false answers. ICS construction has conceptually the same effect as combining multiple CS's, where CS is an auxiliary data structure used in [30] to find all embeddings of *one* graph.

Third, we introduce a new supergraph search method *active-first search* and a new adaptive matching order *relevance-size order*. We extend a partial

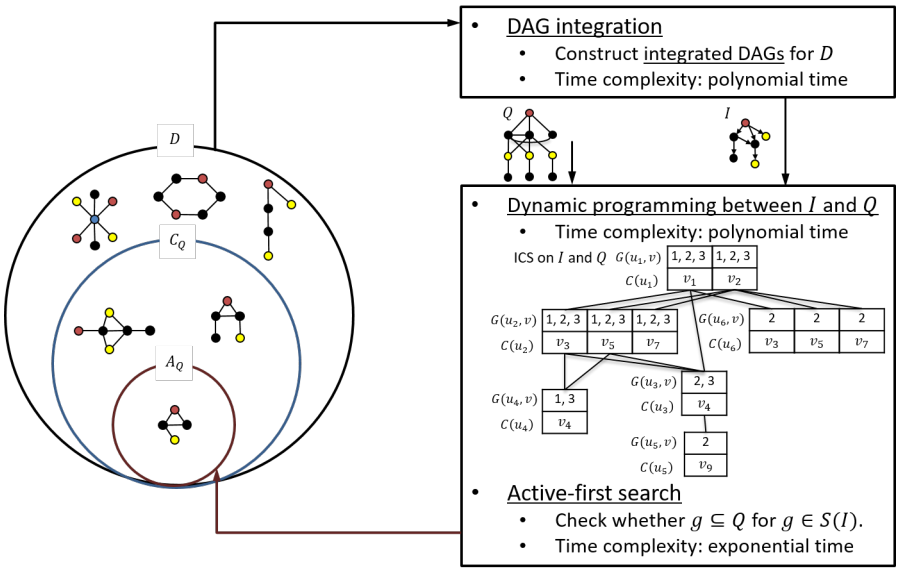


Figure 4.3: Overview of our supergraph search algorithm.

mapping from $V(I)$ to $V(Q)$ in this framework to find an embedding of each candidate graph merged to I. Intuitively, only the vertices in $V(I)$ relevant to the candidate graphs that share the partial mapping (i.e., *active vertices*) can be the next vertices to match in active-first search. Based on relevance-size order, among vertices that can be matched, we first select a next vertex such that the extended partial mapping covers as many partial embeddings of candidate graphs as possible. This search method can save redundant search space by finding at most one embedding of every candidate graph and keeping a large overlap among partial embeddings of candidate graphs.

Figure 4.3 illustrates the overview of our algorithm, which includes the three techniques above, i.e., DAG integration, dynamic programming between I and Q, and active-first search.

Table 4.1 summarizes five supergraph search algorithms. The first four algorithms mine a set F of frequent subgraphs as features from a set D of data graphs by using frequent subgraph mining. In the filtering stage, for each $f \in F$ these algorithms filter the data graphs that contain f not contained in a query

Table 4.1: Summary of existing supergraph search algorithms (where SI represents a subgraph isomorphism test).

Algorithm	Indexing	Filtering	Verification
CIndex [8]	frequent subgraphs	SI on feature	SI on data graph
GPTree [98]	frequent subgraphs	SI on feature	SI on data graph
PrefIndex [107]	frequent subgraphs	SI on feature	SI on data graph
IGQuery [11]	frequent subgraphs & integrated graph	direct inclusion & SI on feature	SI on data graph
DGTree [51]	tree	tree traversal	
IDAR [38]	IDAGs	IDAG-graph DP	active-first search

graph Q. Finally, they conduct a subgraph isomorphism test to verify if each remaining data graph is contained in Q. IGQuery performs the same process as these algorithms, but in addition it integrates every $g \in D$ to an integrated graph IG in indexing, and finds some answers by exploiting the IG (i.e., direct inclusion) before filtering out false answers. DGTree builds a tree of features in indexing. It filters and verifies simultaneously by traversing the tree. IDAR integrates every $g \in D$ to multiple IDAGs. In the filtering stage, IDAR performs dynamic programming between an IDAG and Q, which takes polynomial time unlike the other algorithms. In the verification stage, IDAR uses a new search method that maps IDAG vertices to query vertices, called active-first search.

Organization. The remainder of the paper is organized as follows. Section 4.1 presents a brief overview of our algorithm. Section 4.2 describes DAG integration by which we merge input data graphs. Section 4.3, 4.4 and 4.5 present new techniques used in query processing. Section 4.6 discusses the results of performance evaluation.

4.1 Algorithm Overview

We first describe our index construction algorithm over a set D of data graphs to build a set of integrated DAGs (IDAGs) in Algorithm 3, which follows the procedures below.

Algorithm 3: BUILDINDEX

Input: a set D of data graphs

Output: a set D^* of integrated DAGs

```
1 D' ← ∅;
2 foreach data graph g ∈ D do
3     g' ← BUILDDAG(g); D' ← D' ∪ {g'};
4 Π ← PARTITION(D');
5 foreach for each set in Π do
6     I ← empty integrated DAG;
7     foreach data DAG g' in the set do
8         B ← BOTTOMUPSIM(g', I);
9         FINDMERGING(g', I, B);
```

1. Initially, BUILDDAG is called to build a rooted DAG g' from g for every data graph $g \in D$. In BUILDDAG, we first select a root, which will be merged into a root of IDAG. Since the root is the first vertex in IDAG to match in the search process, we prefer the root of g' to have an infrequent label in D and a large degree for better pruning; thus, the root r of g' is selected as $r \leftarrow \text{argmin}_{u \in V(g)} \frac{\text{freq}(l_g(u), \text{NLPF}(u))}{\deg_g(u)}$, where a neighbor label-pair of u is a pair of labels $(l_g(u'), l_g(u, u'))$ for an adjacent vertex u' of u, NLPF(u) is the frequency of u's distinct neighbor label-pairs, e.g., in Figure 4.4a NLPF(w_1) in g_1 is $\{(B, 1) : 1, (C, 2) : 1\}$ (where edge labels 1 and 2 represent a solid line and a dashed line, respectively), and freq(l, x) is the frequency of a pair of vertex label l and NLPF x. In order to build

g', we traverse g in a BFS order from r, and direct all edges from upper levels to lower levels. We refer to these DAGs for data graphs in D as *data DAGs*.

2. PARTITION takes a set D' of data DAGs as input, and outputs disjoint sets Π of data DAGs in D'. We define the *property* of the root r of a data DAG g' as $(l_{g'}(r), \text{NLPF}(r))$. In PARTITION, we first compute the property p of the root for every data DAG $g' \in D'$, and divide D' into sets such that data DAGs with different root properties are in different sets. We sort all data DAGs with a same property in the ascending order of height (a data DAG with smaller number of vertices comes first among the data DAGs with the same height). Next, we equally divide the sorted data DAGs (with a same property p) into $\gamma|hgt(p)|$ sets where $hgt(p)$ is a set of distinct heights of the data DAGs with property p, and γ is a constant ($\gamma = 0.5$).

3. For each set, we integrate every g' in the set to I in FINDMERGING guided by similarity scores between the vertices of g' and I, which are computed in BOTTOMUPSIM and FINDMERGING (Section 4.2).

Selecting which IDAG a given data DAG should be integrated to based on the similarity scores causes a considerable overhead in our experiments, while its benefit over the current partitioning method based on the root property and the DAG height is minor. This implies that the root property and the DAG height are effective measures in estimating the similarities of DAGs in low cost.

For simplicity of presentation, g will denote a data graph or a data DAG from the next section.

The overall framework of query processing is shown in Algorithm 4, which takes a query graph Q and a set D^* of IDAGs, and finds an answer set A_Q for Q. For every IDAG I, we go through two steps as follows.

Algorithm 4: SUPERGRAPHSEARCH

Input: query graph Q, a set D^* of integrated DAGs

Output: answer set A_Q

```
1 A_Q ← ∅;
2 foreach I ∈ D* do
3     ICS ← BUILDICS(I, Q);
4     f ← ∅;
5     BACKTRACK(I, ICS, f, A_Q);
```

1. First, BUILDICS is invoked to build an ICS by using dynamic programming between an IDAG and a graph (Section 4.3). We will show that finding an embedding of each data graph $g \in D$ in Q is equivalent to finding an embedding of g in the ICS.

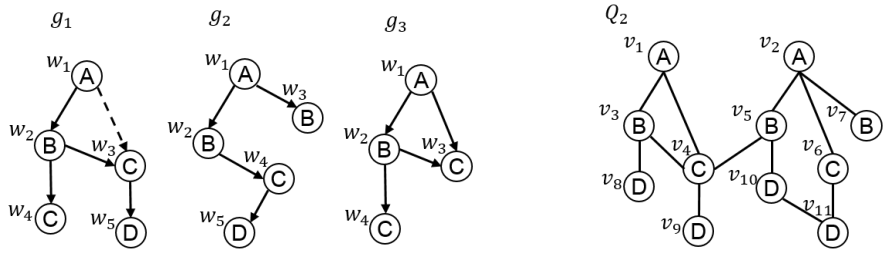
2. Next, we find an embedding of each data graph g integrated to I in the ICS by BACKTRACK. We use a new search technique based on active-first search (Section 4.4) and relevance-size order (Section 4.5).

4.2 DAG Integration

In this section we describe a technique called *DAG integration* in which we construct an IDAG from multiple data graphs.

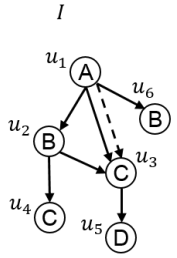
Integrated DAG. Given a set of DAGs, we merge all the DAGs into an *integrated DAG (IDAG)* $I = (V(I), E(I), l_I, D_I, S(I))$ that consists of a set $V(I)$ of integrated vertices, a set $E(I)$ of directed edges, a labeling function l_I that assigns a label to each integrated vertex. Additionally, IDAG I has a set $D_I(u)$ of DAGs merged to each integrated vertex $u \in V(I)$, and a set $D_I(u, u', x)$ of DAGs merged to each edge $(u, u', x) \in E(I)$ where x is an edge label. We also associate with each IDAG I the set $S(I)$ of data graphs integrated to I.

Figure 4.5 shows the IDAG constructed from a set D' of data DAGs in



(a) A set D' of data DAGs. (b) A query graph.

Figure 4.4: Data DAGs and a query graph.



u	$D_I(u)$
u_1	g_1, g_2, g_3
u_2	g_1, g_2, g_3
u_3	g_1, g_2, g_3
u_4	g_1, g_3
u_5	g_1, g_2
u_6	g_2

(u, u', x)	$D_I(u, u', x)$
$(u_1, u_2, 1)$	g_1, g_2, g_3
$(u_1, u_3, 1)$	g_3
$(u_1, u_3, 2)$	g_1
$(u_1, u_6, 1)$	g_2
$(u_2, u_3, 1)$	g_1, g_2, g_3
$(u_2, u_4, 1)$	g_1, g_3
$(u_3, u_5, 1)$	g_1, g_2

Figure 4.5: The integrated DAG for D'.

Figure 4.4a (where we integrate the data DAGs with different root properties for simplicity). The IDAG keeps the set of data DAGs integrated to each integrated vertex and edge as in Figure 4.5. For instance, an edge (w_1, w_3) in g_1 is merged to the edge $(u_1, u_3, 2)$ in the IDAG, i.e., $D_I(u_1, u_3, 2) = \{g_1\}$.

Definition 4.2.1. *Given a DAG g and an IDAG I, a merging of g to I is defined as an embedding $h : V(g) \to V(I) \cup V_{new}$ where V_{new} is a set of newly created vertices during the integration of g to I.*

For example, a merging h_3 of g_3 in Figure 4.4a is an embedding $\{(w_1, u_1), (w_2, u_2), (w_3, u_3), (w_4, u_4)\}$ from g_3 to I in Figure 4.5.

Suppose that we are given a DAG g_k and an IDAG I_{k-1} to which a set $D' = \{g_1, g_2, ..., g_{k-1}\}$ of DAGs are integrated. In our integration method, we integrate g_k to I_{k-1} so that the merging of g_k to I_{k-1} becomes the embedding of g_k in I_k. A merging of an induced subgraph of g_k to I_{k-1} is called a *partial*

merging.

Definition 4.2.2. *Suppose that we are given a partial merging h and an extendable vertex w. The set of mergeable candidates of w regarding h is defined as $C_h(w) = \bigcup_{w_p \in Parent(w)} \hat{N}_w^{w_p}(h(w_p))$, where $\hat{N}_w^{w_p}(u_p)$ represents the set of children u of u_p in $V(I)$ such that $l_g(w) = l_I(u)$.*

Integration Framework. Based on a DAG ordering and the definition of the mergeable candidates, our new integration framework finds a merging of g to I as follows.

1. Select an extendable vertex w regarding the current partial merging h.
2. Extend h by merging w to an unmapped $u \in C_h(w)$ if such a vertex exists in $C_h(w)$; to a newly created integrated vertex u otherwise, and recurse.

Two questions arise: (1) Among all extendable vertices regarding h, which vertex should be extended first? (2) To which unmapped mergeable candidate in $C_h(w)$ should we merge w? In our integration method, we select an extendable vertex w and a mergeable candidate u at once such that the *similarity* score $sim(w, u)$ is maximum among all possible pairs of extendable vertices and their mergeable candidates. We will compute $sim(w, u)$ by using bottom-up similarity and top-down similarity.

Example 4.2.1. Figure 4.6 displays the integration of DAGs in Figure 4.4a. Assume that we just merged w_1 into u_1 during integration of g_2 to I_1 in Figure 4.6a. We select w_2 between two gray extendable vertices w_2 and w_3, and merge w_2 into u_2, because $sim(w_2, u_2)$ is the largest. Note that the sub-DAG rooted at w_2 is similar to the sub-DAG rooted at u_2. In Figure 4.6b, assume that we just merged w_2 into u_2 during integration of g_3 to I_2. Since $sim(w_3, u_3)$ is the largest, we select w_3 rather than w_4 and merge w_3 into u_3. Note that the reverse sub-DAG rooted at w_3 (consisting of w_3, w_2, and w_1) is similar to the reverse sub-DAG rooted at u_3.

Algorithm 5: FindMerging

Input: a DAG g, an IDAG I, bottom-up similarity B

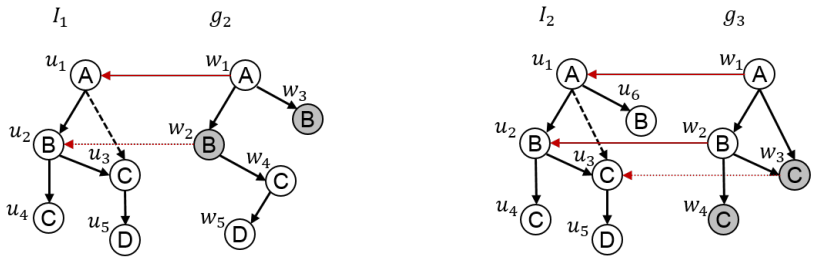
```
1  Update the root u_r of I; h ← {(w_r, u_r)};
2  while there is an extendable vertex in V(g) do
3      (w, u) ← arg max_{(w',u')} sim(w', u') for every extendable
                 vertex w' and u' ∈ C_h(w')
4      if u is matched in h then
5          Remove u from C_h(w);
6          if C_h(w) = ∅ then
7              Create a new vertex û of I; h ← h ∪ {(w, û)};
8      else
9          Update u; h ← h ∪ {(w, u)};
```

Definition 4.2.3. *For each $w \in V(g)$ and $u \in V(I)$, the bottom-up similarity of w and u is defined as $B(w,u) = score(w,u) + \sum_{w_c \in Child(w)} \max_{u_c \in Child(u)} \{B(w_c, u_c)\}$, where $score(w,u)$ is 1 if $l_g(w) = l_I(u)$; 0 otherwise.*

Definition 4.2.4. *Suppose that we are given a partial merging h of g to I, and an extendable vertex $w \in V(g)$. The top-down similarity of w and $u \in C_h(w)$ regarding h is defined as $T_h(w,u) = score(w,u) + \sum_{w_p \in Parent(w)} \{T_h(w_p, h(w_p))\}$. The similarity of w and $u \in C_h(w)$ is defined as $sim(w,u) = B(w,u) + T_h(w,u)$.*

Intuitively, $B(w,u)$ measures how much the sub-DAG rooted at w and the sub-DAG rooted at u are similar, while $T_h(w,u)$ measures the similarity between the reverse sub-DAG rooted at w and that rooted at u.

Bottom-up similarity B is computed in BottomUpSim in Algorithm 3. In BottomUpSim, the bottom-up similarity of every $w \in V(g)$ and $u \in V(I)$ is computed in a reverse topological order of DAG g, i.e., w is processed after all its children in g are processed.



(a) The moment w_1 is just merged into u_1 during integration of g_2 to I_1.

(b) The moment w_2 is just merged into u_2 during integration of g_3 to I_2.

Figure 4.6: Example of DAG integration.

Top-down similarities are computed and a DAG g is integrated into an IDAG I at the same time in Algorithm 5. First we set a partial merging h to $\{(w_r, u_r)\}$, and update u_r by adding g to $D_I(u_r)$, where w_r and u_r are the roots of g and I, respectively (line 1). While there is an extendable vertex in $V(g)$, we select an extendable vertex w such that $sim(w, u)$ is the maximum among all pairs of extendable vertices and their mergeable candidates (lines 2-3). If u is already matched in h, u is removed from $C_h(w)$ (line 4-5); moreover, we create a new integrated vertex $\hat{u}$ and extend h accordingly if $C_h(w) = \emptyset$ (lines 6-7), or repeat the same process from line 2 if $C_h(w) \neq \emptyset$. Otherwise, we update u and extend h by adding (w, u) to h (lines 8-9). When we create or update u, g is added to $D_I(u)$, and $D_I(h(w_p), u, l_g(w_p, w))$ for each $w_p \in \text{Parent}(w)$. We maintain a max priority queue to get the maximum $sim(w, u)$. Once a vertex $w \in V(g)$ becomes extendable due to an extension of h, $C_h(w)$ and top-down similarities between w and $u \in C_h(w)$ are immediately computed.

Back to Figure 4.6a of Example 4.2.1, we select w_2 between two gray extendable vertices w_2 and w_3 because $T_h(w_2, u_2) = T_h(w_3, u_2)$ but $B(w_2, u_2) > B(w_3, u_2)$. In Figure 4.6b of Example 4.2.1, we select a pair (w_3, u_3) and merge w_3 to u_3 because bottom-up similarities of candidate pairs are the same but $T_h(w_3, u_3) > T_h(w_4, u_3)$.

Lemma 4.2.1. *Given a DAG g and an IDAG I, the time and space complexities of DAG integration of g to I are $O(|E(g)||E(I)|+|V(g)||V(I)|\log(|V(g)||V(I)|))$ and $O(|V(g)||V(I)|)$, respectively.*

Since DGTree keeps a feature graph (i.e., a distinct subgraph of data graphs) at its node, DGTree construction takes exponential time for finding embeddings of a feature graph in data graphs (as described in related work of Section **??**). In contrast, DAG integration can effectively merge data graphs into IDAGs in polynomial time, as IGQuery merges data graphs into an IG. Nevertheless, DAG integration differs from the graph integration of IGQuery which uses a depth-first search. In DAG integration, vertices with high similarity are selected to merge based on DAG ordering, which leads to a compact integration for subsequent filtering and search steps.

4.3 Dynamic Programming between IDAG and Graph

DAF [30] constructs an auxiliary data structure called CS (Candidate Space) consisting of candidate vertices and corresponding edges, which serves as a complete search space to find all embeddings of *one* graph. To deal with multiple data graphs at once, we propose an auxiliary data structure called the *integrated candidate space* (ICS). ICS construction, which takes an IDAG I and a query graph Q as input, has conceptually the same effect as combining multiple CS's. By applying dynamic programming to ICS construction, we can efficiently filter false answers and obtain a complete search space for all embeddings of remaining data graphs in Q.

ICS. Given an IDAG I and a query graph Q, *ICS on I and Q* consists of the following.

- For each $u \in V(I)$, a set $C(u)$ of candidate vertices is a set of vertices $v \in V(Q)$ which u can be mapped to. $C(u)$ is a subset of $C_{\text{ini}}(u)$, where $C_{\text{ini}}(u)$ is the set of vertices $v \in V(Q)$ such that $l_I(u) = l_Q(v)$.

- For each $u \in V(I)$ and $v \in C(u)$, a set $G(u, v)$ of candidate graphs is a set of data graphs $g \in D_I(u)$ which can be answer graphs when u is mapped to v. $G(u, v)$ is initially $D_I(u)$. Let Z_u denote $\cup_{\forall v \in C(u)} G(u, v)$.

- There is an edge between $v \in C(u)$ and $v_c \in C(u_c)$ if and only if $(u, u_c, x) \in E(I)$ such that $x = l_Q(v, v_c)$, $(v, v_c) \in E(Q)$, and $G(u, v) \cap G(u_c, v_c) \neq \emptyset$. The edges are stored as an adjacency list $N^u_{u_c}(v)$ for each $v \in C(u)$ and each edge between u and u_c in I, where $N^u_{u_c}(v)$ represents the list of vertices v_c adjacent to v in Q such that $v_c \in C(u_c)$.

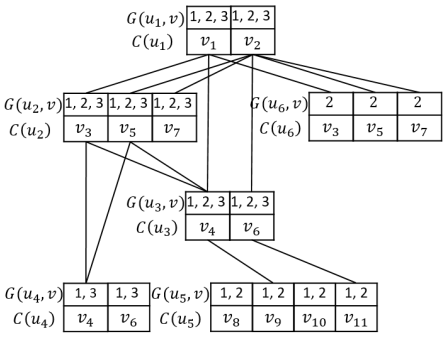
Figure 4.7a shows the initial ICS on I in Figure 4.5 and Q_2 in Figure 4.4b (every number i in $G(u, v)$ represents g_i in Figure 4.7). In the initial ICS, $C_{\text{ini}}(u_3) = \{v_4, v_6\}$ because v_4 and v_6 have the same label as u_3, $G(u_3, v_4) = G(u_3, v_6) = \{g_1, g_2, g_3\}$ since $D_I(u_3) = \{g_1, g_2, g_3\}$, and $Z_{u_3} = G(u_3, v_4) \cup G(u_3, v_6) = \{g_1, g_2, g_3\}$. There are edges between $v_2 \in C(u_1)$ and two candidate vertices in $C(u_6)$, i.e., $N^{u_1}_{u6}(v_2) = \{v_5, v_7\}$.

Definition 4.3.1. *An embedding of a data graph g_i in an ICS on I and Q is defined as an injective mapping $M : V(g_i) \to V(Q)$ such that (1) $M(w) \in C(h_i(w))$ and $g_i \in G(h_i(w), M(w))$ for every $w \in V(g_i)$ where h_i is the merging of g_i to I, and (2) there is an edge $(M(w), M(w'))$ with label $l_{g_i}(w, w')$ in the ICS for every $(w, w') \in E(g_i)$.*

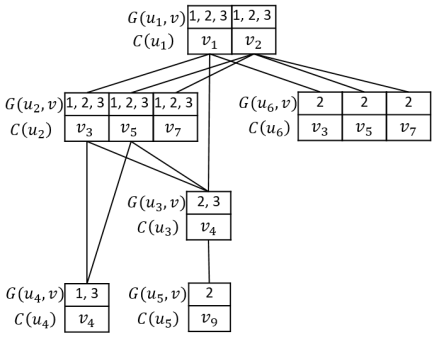
Definition 4.3.2. *An ICS on I and Q is sound if it satisfies the following statement: if there is an embedding $M : V(g) \to V(Q)$ of $g \in S(I)$ in Q such that $M(w) = v$, then v and g must exist in $C(h(w))$ and $G(h(w), v)$, respectively. (Recall that $S(I)$ is the set of data graphs integrated to I.)*

Definition 4.3.3. *An ICS on I and Q is equivalent to Q with respect to I if the set of all embeddings of $g \in S(I)$ in Q is the same as the set of all embeddings of $g \in S(I)$ in the ICS.*

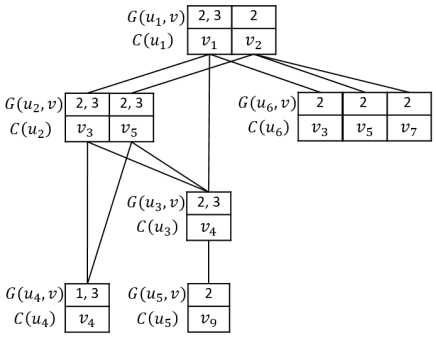
Then we have the following *equivalence property*.



(a) Initial ICS.



(b) ICS after the 1st refinement.



(c) ICS after the 2nd refinement.

Figure 4.7: Refinements of ICS using IDAG-graph DP.

Theorem 4.3.1. *If an ICS on I and Q is sound, it is equivalent to Q with respect to I.*

Once we compute a compact sound ICS, Q is no longer necessary afterward by the equivalence property.

IDAG-Graph DP. For each IDAG I, we find weak embeddings of all data graphs in $S(I)$ in our new technique called *dynamic programming between IDAG and graph* (for short, *IDAG-graph DP*). Given a sound ICS, we have the following observation.

- If there is an embedding M of a data DAG g^* (g^* can be g or g^{-1}) in ICS such that $M(w) = v$ for a vertex $w \in V(g^*)$, there must be a weak embedding of g^*_w at v in the ICS where g^*_w is the sub-DAG of g^* rooted at

w.

Based on this observation, we propose an ICS refinement algorithm by using dynamic programming in order to remove unpromising candidates from $C(u)$ and $G(u, v)$ if such a weak embedding does not exist. First, for each set $G(u, v)$ of candidate graphs, we define the *refined set of candidate graphs* $G'(u, v)$ as follows:

$$g \in G'(u, v) \text{ iff } g \in G(u, v) \text{ and there is a weak embedding of } g^*_w \text{ at } v \text{ in the ICS, where } u = h(w). \tag{4.1}$$

Second, for each set $C(u)$ of candidate vertices, we define the *refined set of candidate vertices* $C'(u)$ as follows:

$$v \in C'(u) \text{ iff } v \in C(u) \text{ and } G'(u, v) \neq \emptyset.$$

For example, if we apply this refinement to the ICS in Figure 4.7b over the IDAG I in Figure 4.5, we obtain the refined ICS in Figure 4.7c. Note that v_7 is removed from $C(u_2)$ since there are no weak embeddings of $g_{1,w_2}, g_{2,w_2}, g_{3,w_2}$ at v_7 in the ICS of Figure 4.7b, so $G(u_2, v_7) = \emptyset$. On the other hand, v_2 stays in $C(u_1)$ even though there are no weak embeddings of g_{1,w_1} and g_{3,w_1} at v_2 since there is a weak embedding of g_{2,w_1} at v_2 in the ICS.

To compute $G'(u, v)$, we remove the data graphs that do not satisfy Condition (4.1) from $G(u, v)$. A data DAG g^* (with merging h) such that there is no weak embedding of a sub-DAG g^*_w at v (where $u = h(w)$) has at least one outgoing edge (w, w_c) which has no corresponding edge (v, v_c) such that a weak embedding of $g^*_{w_c}$ exists at v_c. We define and compute a set $F^u_{u_c}(v)$ of such graphs in a bottom-up fashion.

Definition 4.3.4. *For each $u \in V(I)$, $v \in C(u)$, and a child u_c of u, the set $F^u_{u_c}(v)$ of filtered graphs is the set of $g \in D_I(u, u_c, x)$ that has the following property: there is no $v_c \in C'(u_c)$ adjacent to v with $l_Q(v, v_c) = x$ such that $g \in G'(u_c, v_c)$.*

To compute $G'(u, v)$, we use the following recurrence:

$$G'(u, v) = G(u, v) - \cup_{u_c \in \text{Child}(u)} F^u_{u_c}(v),$$

where

$$F^u_{u_c}(v) = \cup_{(u,u_c,x) \in E(I)} \{D_I(u, u_c, x) - \cup_{v_c \in N^u_{u_c}(v,x)} G'(u_c, v_c)\},$$

and $N^u_{u_c}(v, x) = \{v_c \in N^u_{u_c}(v) \mid l_Q(v, v_c) = x\}$. According to the recurrence above, we compute $G'(u, v)$ and $C'(u)$ for all $u \in V(I)$ by dynamic programming in a reverse topological order of I^*(i.e, I or I^{-1}), in which u is processed after all children of u are processed. All the sets of data graphs above are implemented as bit-arrays ($|S(I)|$ bits per set) so that union, intersection, and difference operations can be efficiently done in $O(|S(I)|/w)$ time, where w is a word size.

Lemma 4.3.1. *Given an ICS on I and Q, the time complexity of ICS construction is $O(|E(I)||E(Q)||S(I)|/w)$, and and space complexity of ICS construction is $O(|E(I)||E(Q)| + |V(I)||V(Q)||S(I)|/w)$.*

Building a Compact ICS. By using IDAG-Graph DP multiple times in different orders, we can filter as many data graphs that have no weak embeddings as possible, and get a small search space for remaining data graphs.

First, for every $u \in V(I)$, $C(u)$ is initialized to $C_{\text{ini}}(u)$, and $G(u, v)$ is initialized to $D_I(u)$ for every $v \in C_{\text{ini}}(u)$. Moreover, a set A^c_Q of filtered graphs, i.e., a set of data graphs not contained in the query, is initialized to $\emptyset$.

Next, IDAG-Graph DP refines the ICS over the rooted IDAG I and its reverse I^{-1} alternately. We perform IDAG-Graph DP using I^{-1} to the initial CS. We then further refine the ICS over I. The refined candidate sets in the current step may help further refine the candidate sets using I^{-1} again, and so on. Our empirical study showed that three steps are enough for optimization, so we set this number to 3 in our experiments. For every $u \in V(I)$, we clear $C(u)$ if $Z_u \subseteq A^c_Q$. Otherwise, we refine $C(u)$ and $G(u, v)$, calculate $Z_u = \cup_{\forall v \in C(u)} G(u, v)$, and update A^c_Q as follows: $A^c_Q \leftarrow A^c_Q \cup (D_I(u) - Z_u)$.

After computing the final candidate sets, we materialize the edges as an adjacency list $N^u_{u_c}(v)$ for each $v \in C(u)$ to obtain the complete ICS. During the refinements, we only maintain $C(u)$, $G(u, v)$, and Z_u. The edges in Figure 4.7 are illustrated only for presentation.

After ICS construction, we do not need to consider a vertex $u \in V(I)$ that has no candidate (i.e., $C(u) = \emptyset$). Let V' be a set of integrated vertices with nonempty $C(u)$, then we assume that an IDAG is $I[V']$ from the next section.

4.4 Active-First Search

From this subsection we present our new matching algorithm to find embeddings of candidate graphs (i.e, the remaining data graphs in $S(I)$ after the filtering of ICS) in the ICS. Compared to the backtracking frameworks with a single given graph in existing subgraph matching algorithms [31, 4, 30], our technique is specifically designed to search for a common partial embedding shared by candidate graphs via a mapping from $V(I)$ to $V(Q)$ with as small search space as possible by taking advantage of the overlap between the candidate graphs in I.

In our search method, only *active* vertices in $V(I)$ are allowed to be matched in a current partial mapping $f : V(I) \to V(Q)$. Figure 4.8 shows an illustration of an IDAG to which g_1, g_2 and g_3 are integrated. Suppose we just matched u_1 and u_2 in current partial mapping f. Since f covers all data graphs, all children of u_1 or u_2 (i.e., u_3, u_5 and u_7) are active. However, if we extend f to f' by matching u_3, f' covers only g_1 and g_2. Thus, u_4 and u_5 are active, but u_7 is no longer active regarding f'.

Example 4.4.1. Figure 4.9 shows an IDAG I to which DAGs for data graphs in D of Figure 4.2a are integrated, and Figure 4.10 illustrates the ICS on I and Q_1 of Figure 4.2b. There are embeddings of g_1 and g_3 in the ICS, e.g., $\{(w_1, v_1), (w_2, v_4), (w_3, v_{11}), (w_4, v_{23}), (w_5, v_{26})\}$ for g_1, and $\{(w_1, v_1), (w_2, v_4), (w_3, v_{23})\}$

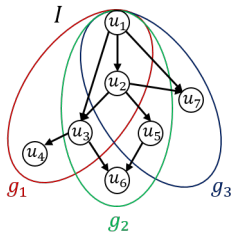


Figure 4.8: An illustrating example of an IDAG.

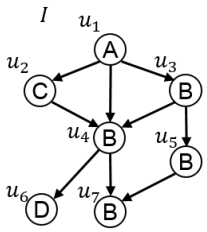
for g_3; however, there is no embedding of g_2.

We define a partial mapping $f : V(I) \to V(Q)$ and a set of data graphs covered by f as follows.

Definition 4.4.1. *A (partial) feasible mapping of an IDAG I in the ICS is defined as a mapping $f : V(I) \to V(Q)$ such that there exists at least one data graph $g_i \in S(I)$ that has its merging $h_i : V(g_i) \to V(I)$ where $f \circ h_i : V(g_i) \to V(Q)$ is a (partial) embedding of g_i in ICS, and for every $(u, v) \in f$, a vertex of g_i is merged to u, i.e., $h_i^{-1}(u)$ exists. Such data graphs are a set of (partial) feasible graphs regarding f, denoted as FG_f^* (PFG_f^*). Let PFG_f denote $PFG_f^* - A_Q$.*

Example 4.4.2. Figure 4.11 is a search tree for IDAG I in Figure 4.9 and query graph Q_1 in Figure 4.2b. A node in the search tree corresponds to a partial feasible mapping, e.g., the root of the search tree corresponds to partial feasible mapping $f_1 = \{(u_1, v_1)\}$, its left child labeled with (u_2, v_4) corresponds to $f_{11} = \{(u_1, v_1), (u_2, v_4)\}$, and so on. Therefore we use f to represent a node as well as a partial feasible mapping. Each node shows the latest mapping (u, v) of f and $\text{PFG}_f = \{g_1, ..., g_k\}$ (when $A_{Q_1} = \emptyset$). In the search tree of Figure 4.11, $\text{PFG}_{f_1} = \{g_1, g_2, g_3\}$ for node $f_1 = \{(u_1, v_1)\}$. Furthermore, $\text{PFG}_{f_{11}} = \{g_1, g_3\}$ for node $f_{11} = \{(u_1, v_1), (u_2, v_4)\}$ since g_2 is not merged to u_2, and $\text{PFG}_{f_{21}} = \{g_1\}$ for $f_{21} = \{(u_1, v_1), (u_2, v_4), (u_3, v_{11})\}$ since g_3 is not merged to u_3.

Definition 4.4.2. *Let G denote a set of data graphs. An integrated vertex u is*



u	$D_I(u)$
u_1	g_1, g_2, g_3
u_2	g_1, g_3
u_3	g_1, g_2
u_4	g_1, g_2, g_3
u_5	g_2
u_6	g_1
u_7	g_2

(u, u', x)	$D_I(u, u', x)$
$(u_1, u_2, 1)$	g_1, g_3
$(u_1, u_3, 1)$	g_1, g_2
$(u_1, u_4, 1)$	g_1, g_2, g_3
$(u_2, u_4, 1)$	g_1, g_3
$(u_3, u_4, 1)$	g_1, g_2
$(u_3, u_5, 1)$	g_2
$(u_4, u_6, 1)$	g_1
$(u_4, u_7, 1)$	g_2
$(u_5, u_7, 1)$	g_2

Figure 4.9: The IDAG built from D of Figure 4.2 .

irrelevant to G if G and Z_u have no common graph (i.e., $G \cap Z_u = \emptyset$); relevant to G otherwise.

Definition 4.4.3. *Suppose that we are given a partial feasible mapping f and a set PFG_f of partial feasible graphs. An unvisited (i.e., unmapped) vertex $u \in V(I)$ is called active regarding f if u is relevant to PFG_f and all the u's parents relevant to PFG_f are matched in f.*

Now we describe our search method. Given a partial feasible mapping f with PFG_f, let $u_1, ..., u_n$ be the integrated vertices extended from f in the search tree. Assume that we have explored the subtrees rooted at $f \cup \{(u_k, v)\}$ (where $1 \leq k \leq n$) for every v that u_k can be matched to. An *active-first search* always selects an active vertex relevant to $U_{f,k}$ as the next vertex to map, where $U_{f,0} = \text{PFG}_f$ and $U_{f,k} = \text{PFG}_f - \cup_{i=1}^{k} Z_{u_k}$ for $k \geq 1$.

We consider $U_{f,k}$, a subset of graphs in PFG_f that have been untried as partial feasible graphs in the extensions of f, in order not to find duplicate embeddings of $g \in \text{PFG}_f$. In Example 4.4.2, when we first visited f_1, $U_{f_1,0} = \text{PFG}_{f_1} = \{g_1, g_2, g_3\}$, thus u_2 and u_3 can be matched. Now, suppose that we just came back to node f_1 after the exploration of the subtrees rooted at f_{11} and f_{12}. During the exploration, we have tried all possible extensions to map u_2 with partial feasible graphs g_1 and g_3. Then $U_{f_1,1} = \text{PFG}_{f_1} - \{g_1, g_3\} = \{g_2\}$, so only u_3 can be matched.

We define a failure that a partial feasible graph regarding f becomes no

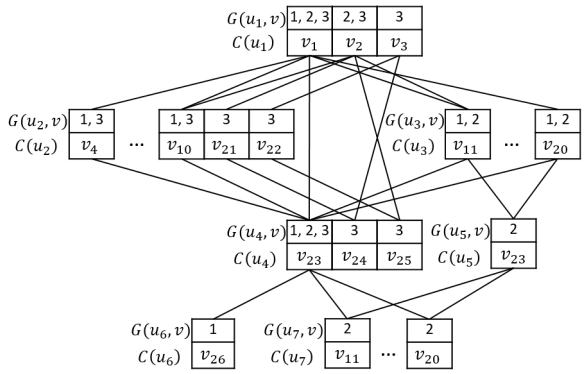


Figure 4.10: ICS on I in Figure 4.9 and Q_1 in Figure 4.2 .

longer partially feasible regarding $f' = f \cup \{(u, v)\}$. A candidate graph g_i belongs to a set $emp_f(u, v)$ of *empty-set graphs* if $w = h_i^{-1}(u)$ cannot be matched to v regarding $M' = f' \circ h_i$.

Definition 4.4.4. *Suppose that we are given a partial feasible mapping f and an active vertex u. Given a candidate vertex $v \in C(u)$, the set of matchable graphs of u and v regarding f is defined as $G_f(u, v) = G(u, v) - emp_f(u, v)$. The set of matchable candidates of u regarding f is defined as $C_f(u) = \{v \in \cup_{u_p \in Parent_f(u)} N_u^{u_p}(f(u_p)) \mid G_f(u, v) \neq \emptyset\}$ where $Parent_f(u)$ is u's parents relevant to PFG_f.*

Example 4.4.3. Given a partial feasible mapping $f = \{(u_1, v_1), (u_2, v_4), (u_3, v_{11})\}$ in Figure 4.10, u_4 is the only active vertex. Since $N_{u_4}^{u_1}(v_1) \cup N_{u_4}^{u_2}(v_4) \cup N_{u_4}^{u_3}(v_{11}) = \{v_{23}\}$ and $G_f(u_4, v_{23}) = \{g_1, g_2, g_3\}$, v_{23} is the only matchable candidate of u_4.

Lemma 4.4.1. *Suppose that we are given a partial feasible mapping f and an active vertex u_{k+1} relevant to $U_{f,k}$. For every candidate $v \in C_f(u_{k+1})$, $f' = f \cup \{(u_{k+1}, v)\}$ is a partial feasible mapping with $PFG_{f'} = U_{f,k} \cap G_f(u, v)$.*

In Examples 4.4.2 and 4.4.3, suppose that we just extended to $f = \{(u_1, v_1), (u_2, v_4), (u_3, v_{11})\}$ with $U_{f,0} = \text{PFG}_f = \{g_1\}$. Then $f' = f \cup \{(u_4, v_{23})\}$ is a partial feasible mapping with $\text{PFG}_{f'} = U_{f,0} \cap G_f(u_4, v_{23}) = \{g_1\}$, but we cannot extend f to u_5 which is not relevant to $U_{f,0}$, i.e., $U_{f,0} \cap Z_{u_5} = \emptyset$.

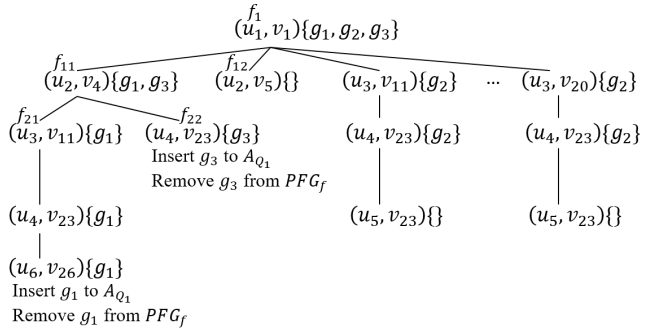


Figure 4.11: Search tree of backtracking for new running example of I in Figure 4.9 and Q_1 in Figure 4.2. Each node shows the latest mapping (u, v) of f and $\text{PFG}_f = \{g_1, ..., g_k\}$ when $A_{Q_1} = \emptyset$.

Backtracking Framework. Based on Lemma 4.4.1, our backtracking framework finds an embedding of each data graph in the ICS as follows.

1. Select an active vertex u relevant to $U_{f,k}$ regarding the current partial feasible mapping f.

2. Extend f to f' by mapping u to each unvisited $v \in C_f(u)$ if $C_f(u) \neq \emptyset$; a dummy vertex v^* otherwise (a vertex u with $C_f(u) = \emptyset$ should be matched to v^* to make its children active).

3. Compute $\text{PFG}_{f'}$ regarding the extended partial feasible mapping f'.

4. Extend $M_i : V(g_i) \to V(Q)$ by mapping $h_i^{-1}(u)$ to v for each $g_i \in \text{PFG}_{f'}$ (if all vertices in $V(g_i)$ are matched, insert g_i to A_Q and remove g_i from PFGs), and recurse.

Figure 4.11 shows the search tree of the backtracking framework above. Suppose that we just came back to node $f_{21} = \{(u_1, v_1), (u_2, v_4), (u_3, v_{11})\}$ with $\text{PFG}_{f_{21}} = \{g_1\}$ after the exploration of the subtree rooted at f_{21}. During the exploration, we have narrowed down search space to extend partial feasible mappings regarding only $\text{PFG}_{f_{21}}$, and tried all possible extensions to map u_3,

i.e., the first integrated vertex extended from f_{11}, to every $v \in C_{f_{21}}(u_3)$. After the exploration, we have $U_{f_{11},1} = \text{PFG}_{f_{11}} - Z_{u_3} = \{g_3\}$. Then we next match the active vertex u_4 relevant to $U_{f_{11},1}$. We have $f_{22} = \{(u_1, v_1), (u_2, v_4), (u_4, v_{23})\}$ and $\text{PFG}_{f_{22}} = \{g_3\}$, and focus on finding an embedding of g_3. Similarly, suppose that we just came back to node f_1 after we have explored the subtrees with active vertex u_2 to search for $\text{PFG}_{f_{11}} = \{g_1, g_3\}$. Now we have $U_{f_1,1} = \text{PFG}_{f_1} - Z_{u_2} = \{g_2\}$. Next, we choose u_3 as the next vertex to match.

Computing Empty-set Graphs. We describe how to compute empty-set graphs. Suppose that we try to extend f to $f' = f \cup \{(u, v)\}$ by matching active vertex u to its matchable candidate v. The set $emp_f(u, v)$ of empty-set graphs is the set of $g_i \in D_I(u)$ (where $u = h_i(w)$) such that for $w_p \in \text{Parent}(w)$, there is at least one edge (w_p, w) that does not correspond to edge $(M(w_p), v) \in E(Q)$, where $M = f \circ h_i$. Now we compute $emp_f(u, v)$ as follows.

$$emp_f(u, v) = \cup_{u_p \in \text{Parent}_f(u)} \{\cup_{(u_p,u,x) \in E(I)} D_{I,f}(u_p, u, v, x)\},$$

where $D_{I,f}(u_p, u, v, x)$ is $\emptyset$ if $v \in N_u^{u_p}(f(u_p), x)$; $D_I(u_p, u, x)$ otherwise. Recall that $N_u^{u_p}(v_p, x) = \{v \in N_u^{u_p}(v_p) \mid l_Q(v_p, v) = x\}$.

Consider the running example in Figure 4.10 in the backtracking framework above. Note that all graphs have single edge labels (i.e., 1). Suppose that we just visit a partial feasible mapping $f = \{(u_1, v_3), (u_2, v_{22})\}$ with $U_{f,0} = \text{PFG}_f = \{g_3\}$. Since u_4 is active regarding f, we compute $emp_f(u_4, v_{24})$ to obtain the set $G_f(u_4, v_{24})$ of matchable graphs. The parents of u_4 relevant to PFG_f (i.e., u_1 and u_2) are matched to query vertices, and $v_{24} \in N_{u_4}^{u_1}(f(u_1))$. However, $v_{24} \notin N_{u_4}^{u_2}(f(u_2))$, so for every $(u_2, u_4, x) \in E(I)$, all graphs in $D_I(u_2, u_4, x)$ have no partial embeddings if we extend f to $f' = f \cup \{(u_4, v_{24})\}$. Thus, $emp_f(u_4, v_{24}) = D_{I,f}(u_1, u_4, v_{24}, 1) \cup D_{I,f}(u_2, u_4, v_{24}, 1) = \emptyset \cup D_I(u_2, u_4, 1) = \{g_1, g_3\}$. Finally, we can compute $G_f(u_4, v_{24}) = G(u_4, v_{24}) - emp_f(u_4, v_{24}) = \emptyset$. Hence, v_{24} is not a matchable candidate of u_4, so we do not extend f to f'.

4.5 Relevance-Size Order

When we extend a partial feasible mapping f, there may be more than one vertex that can be matched. Which vertex should be extended first among them? To answer this question, we describe an adaptive matching order suitable for the backtracking framework based on active-first search.

Given a partial feasible mapping f, we prefer $f' = f \cup \{(u, v)\}$ to have large $|\mathrm{PFG}_{f'}|$ so that f' is shared by as many partial feasible graphs as possible in order to reduce the search space. In Figure 4.8, suppose that we matched u_1 and u_2 in a partial feasible mapping f, and u_3, u_5, u_7 are active. We prefer u_3 or u_5 to u_7 as the next vertex to match since u_3, u_5 are shared by more partial feasible graphs than u_7 regarding f. However, since a matchable candidate v is undecided when we try to choose the next vertex u, we consider an upper bound of $|\mathrm{PFG}_{f'}|$ instead of $|\mathrm{PFG}_{f'}|$ for every vertex that can be matched as follows.

$$\mathrm{PFG}_{f'} = U_{f,k} \cap G_f(u, v) \subseteq U_{f,k} \cap Z_u$$

where k is the number of integrated vertices that have been extended from f. We call the size of the upper bound $U_{f,k} \cap Z_u$ the *relevance* $r_f(u)$ of u regarding f.

Relevance-Size order. We make use of the estimation above to choose the next vertex in our matching order.

1. Select an active vertex u relevant to $U_{f,k}$ such that $r_f(u)$ is the maximum.

2. If there is more than one vertex with the maximum $r_f(u)$, select u with the minimum $|C_f(u)|$ among them.

Since we primarily consider the number of common graphs of $U_{f,k}$ and Z_u, this matching order is called the *relevance-size order*, where $r_f(u)$ and $C_f(u)$ are computed regarding the partial feasible mapping f. Thus, the next vertex

selected may be different for different partial feasible mappings; that is, the relevance-size order is an adaptive matching order.

We also adopt the leaf decomposition strategy of [4] where vertices in I are decomposed into the set of degree-one vertices and the set V' of the remaining vertices so that we first match $I[V']$, and then try the degree-one vertices.

Search Process. Algorithm 6 shows the backtracking process in which we find an embedding of every $g_i \in S(I)$ in Q in the ICS by extending a partial feasible mapping f. For simplicity, $U_{f,k}$ is represented as U_f for every $0 \leq k \leq n$, where n is the number of extended integrated vertices from f. If $|f| = 0$, we map the root $r \in V(I)$ to one of its candidates $v \in C(r)$ and recursively invoke BACKTRACK (lines 1-6). If $|f| \geq 1$, we backtrack if there is no active vertex relevant to U_f (line 7); otherwise, we let U_f initially be PFG_f and repeat the following. While there is a graph in U_f (line 10), u is selected based on the relevance-size order (line 11), f is updated to f', and $\text{PFG}_{f'}$ (and U_f) is computed according to the following cases before (and after) BACKTRACK is invoked (lines 12-24). Specifically, if there is no matchable candidate of u, we extend f to $f' = f \cup \{(u, v^*)\}$, let $\text{PFG}_{f'}$ be $U_f - Z_u$, and update U_f to $U_f - PFG_{f'}$ (lines 13-15). Otherwise, for each unvisited $v \in C_f(u)$, we extend f to $f' = f \cup \{(u, v)\}$, set $\text{PFG}_{f'}$ to $U_f \cap G_f(u, v)$, extend a partial embedding of each $g \in \text{PFG}_{f'}$ in Q. If an embedding of g is found, we remove g from PFGs and insert g to the answer set A_Q. If all graphs in Z_u are already in A_Q, we break (lines 17-23). We remove Z_u from U_f after mapping u to all matchable candidates of u (line 24). The relevance and matchable candidates of a vertex is computed immediately when it becomes active due to an extension of f.

Algorithm 6: BACKTRACK(I, ICS, f, A_Q)

```
1  if |f| = 0 then
2      foreach v ∈ C(r) do
3          f ← {(r, v)}; PFG_f ← G(r, v);
4          Mark v as visited;
5          BACKTRACK(I, ICS, f, A_Q);
6          Mark v as unvisited;
7  else if there is no active vertex relevant to U_f then return;
8  else
9      U_f ← PFG_f;
10     while U_f ≠ ∅ do
11         u ← next vertex based on relevance-size order;
12         if C_f(u) = ∅ then
13             f' ← f ∪ {(u, v*)}; PFG_f' ← U_f − Z_u;
14             BACKTRACK(I, ICS, f', A_Q);
15             U_f ← U_f − PFG_f';
16         else
17             foreach unvisited v ∈ C_f(u) do
18                 f' ← f ∪ {(u, v)}; PFG_f' ← U_f ∩ G_f(u, v);
19                 Mark v as visited;
20                 Extend partial embedding of g ∈ PFG_f' in Q, and if an
                    embedding of g is found, insert g to A_Q;
21                 BACKTRACK(I, ICS, f', A_Q);
22                 Mark v as unvisited;
23                 if Z_u ⊆ A_Q then break;
24             U_f ← U_f − Z_u;
```

4.6 Performance Evaluation

In this section, we present experimental results to show the effectiveness of our algorithm, referred to as IDAR. Since two state-of-the-art supergraph search algorithms IGQuery [11] and DGTree [51] significantly outperformed other existing algorithms for graphs with small size and large size, respectively, we mainly compare our approach against these two algorithms. These methods are evaluated in several aspects: (1) the effect of the number of vertices in data graphs, (2) the effect of the number of vertices in a query graph, (3) the effect of the number of data graphs, and (4) the effect of the number of answer graphs.

Experiments are conducted on a Windows machine with an Intel i5-7500 3.40GHz CPU and 16GB memory. The executable file of IGQuery was obtained from the authors in [11]. We couldn't get the code of DGTree from its authors. Nevertheless, we have communicated with the authors to get implementation details, and implemented the algorithm in [51]. Its performance is as good as (actually slightly better than) the one in [51] when compared against IGQuery.

Table 4.2: Experiment settings (k = thousand).

Parameter	Range	Default
$\|V(g)\|$ (rand)	1-20, 21-40, 41-60, 61-80, 81-100	1-100
$\|V(g)\|$ (freq)	1-10, 11-20, 21-30, 31-40	1-40
$\|V(Q)\|$	101-120, 121-140, 141-160, 161-180, 181-200, 201-	101-
$\|D\|$	10k, 20k, 40k, 60k, 80k, 100k	10k

Datasets. Experiments were performed on real datasets: AIDS, NCI, and PubChem. AIDS contains 42,687 compounds with $1 \leq |V| \leq 438$ in the AIDS antiviral screen dataset[76]. NCI consists of 265,242 graphs with $1 \leq |V| \leq 342$ obtained from the National Cancer Institute database[77]. From PubChem, the

open chemistry database at the NIH[78], 499,963 chemical compound structures with $1 \leq |V| \leq 801$ are downloaded.

Query and Data Graphs. Since IGQuery and DGTree are the state-of-the-art algorithms to compare, we prepare the query and data graphs in a way similar to [11] and [51] for fair comparison. For each dataset, we use the graphs with $|V| > 100$ as the basic query set. To evaluate the performance of query processing, we randomly select 100 query graphs from the basic query set, and take the average processing time. We extract data graphs from the basic query set in two different manners as in Table 4.2: (1) random walk on a randomly selected graph, from which we obtain a subgraph containing all the visited vertices and some edges between these vertices (denoted by "random"), and (2) frequent subgraph mining (denoted by "frequent") at minimum support threshold 0.1 [11]. One experiment consists of a set D of data graphs and 100 query graphs. Default values for the number $V(g)$ of vertices in a data graph, the number $V(Q)$ of vertices in a query graph, and the number $|D|$ of data graphs are shown in Table 4.2. If not specified, the parameters are set to their default values.

Time Complexity. Given a query graph Q and a set D of data graphs, the time complexity of each algorithm is shown in Table 4.3. While DGTree takes exponential time in the worst case for indexing, the others take polynomial time. All the algorithms take exponential time in the worst case for query processing, resulting from the nature of a NP-hard problem. Since they are designed to reduce query processing time experimentally, they cannot be compared by the worst case time complexities, but they should be compared by experiments. IGQuery and IDAR may reduce query processing time in practice by using polynomial-time heuristic techniques called direct inclusion and IDAG-graph DP, respectively, before exponential-time subsequent steps.

Number of Vertices in Data Graphs. First, we vary the number of vertices in data graphs. Specifically, we consider the sets of random data graphs with

Table 4.3: Time complexities of algorithms (exp = exponential, w = word size, $S(I)$ = graphs integrated to I).

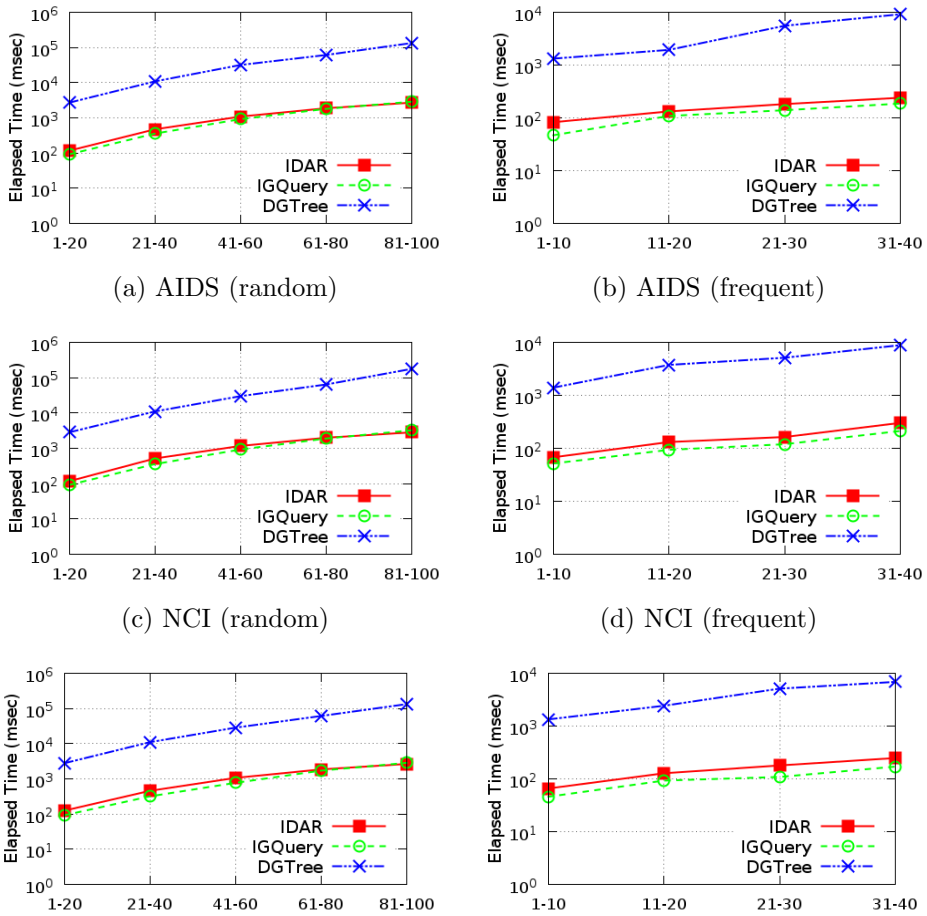
	Indexing	Query processing
IGQuery	$\Sigma_{g \in D}\|E(g)\|^2$	$\|E(Q)\|^2$ (direct inclusion) + exp (filtering) + exp (verification)
DGTree	exp	exp
IDAR	$\Sigma_{g \in D}\{\|E(g)\|\|E(I)\|+ \|V(g)\|\|V(I)\|\log(\|V(g)\|\|V(I)\|)\}$	$\Sigma_{\forall I}\|E(I)\|\|E(Q)\|\|S(I)\|/w$ (IDAG-graph DP) + exp (search)

1-20, 21-40, 41-60, 61-80, 81-100 vertices, and frequent data graphs with 1-10, 11-20, 21-30, 31-40 vertices.

Figure 4.12 presents the time for index construction. IDAR and IGQuery are comparable, whereas DGTree has the worst index construction time, e.g., IDAR is faster than DGTree by at least one order of magnitude. The performance gap between DGTree and IDAR increases as the number of vertices in data graphs grows, because DAG integration in IDAR takes polynomial time unlike DGTree construction that may take exponential time (to find embeddings of feature graphs in the data graphs). If we assume that $|E(g)|$ is significantly larger than $|V(g)|$ for $g \in D$, the time complexities of IDAR and IGQuery in Table 4.3 are quadratic functions in the number of edges. Indeed, IDAR and IGQuery take similar time in indexing as shown in Figure 4.12.

In query processing performance, IDAR always outperforms the competitors as shown in Figure 4.13.

- On the one hand, IDAR is faster than IGQuery by up to two orders of magnitude (81-100 in the random data graphs of AIDS, NCI, and Pub-



(a) AIDS (random)

(b) AIDS (frequent)

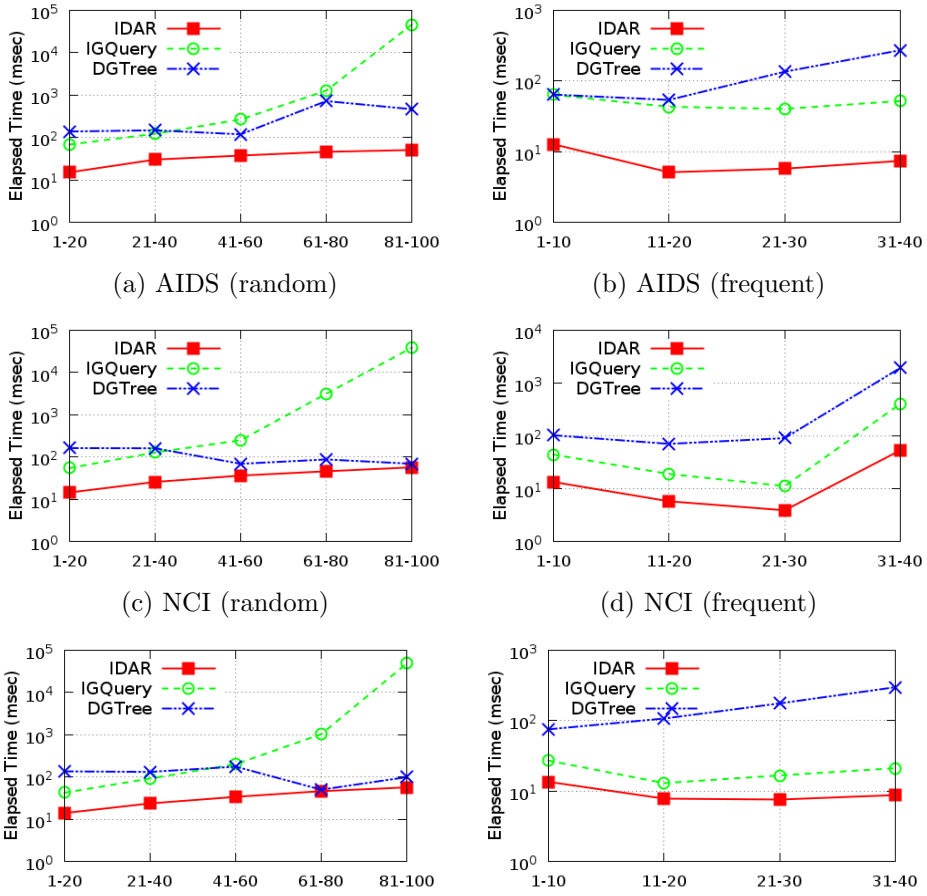
(c) NCI (random)

(d) NCI (frequent)

(e) PubChem (random)

(f) PubChem (frequent)

Figure 4.12: Indexing time for varying number of vertices in data graphs.



(a) AIDS (random)

(b) AIDS (frequent)

(c) NCI (random)

(d) NCI (frequent)

(e) PubChem (random)

(f) PubChem (frequent)

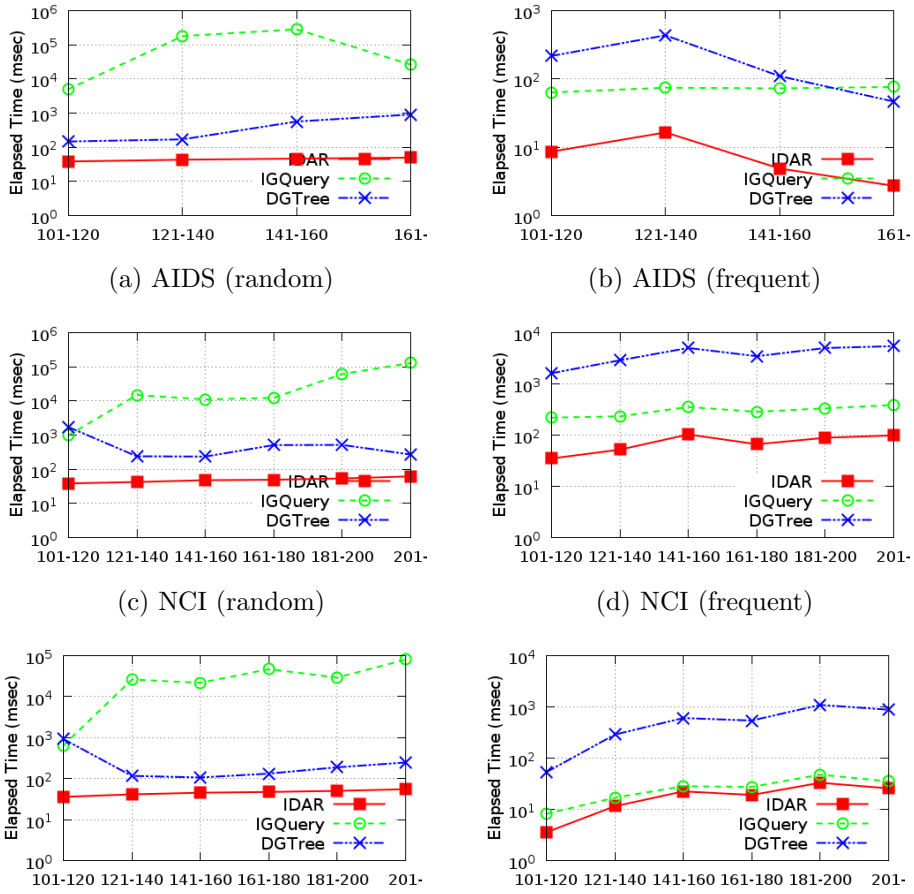
Figure 4.13: Query processing time for varying number of vertices in data graphs.

Chem); furthermore, the query processing time of IDAR generally remains steady, while that of IGQuery exponentially grows. We attribute this phenomenon to the different heuristics implemented by the algorithms, i.e., IGQuery suffers from heavy cost of subgraph isomorphism tests in the filtering and verification phases as data graph sizes increase, which takes exponential time in the worst case. However, the effective filtering (IDAG-graph DP) and efficient search strategy of IDAR result in good scalability with respect to the number of vertices in data graphs.

- On the other hand, IDAR outperforms DGTree by up to one order of magnitude for large-sized frequent data graphs. The reason for this is that IDAR searches for only one embedding of each answer graph, whereas DGTree finds all (partial) embeddings of the answer graphs, which can be costly for the data graphs that have a lot of embeddings in a query graph.

Number of Vertices in a Query Graph. Next, we vary the number of vertices in a query graph as shown in Figure 4.14: 101-120, 121-140, 141-160, 161-180, 181-200, 201- for NCI and PubChem; and 101-120, 121-140, 141-160, 161- for AIDS (query graphs with $|V| > 160$ are not enough to be divided into separate groups in AIDS, so we regard 161- as a single group). IDAR remains steadier and runs consistently faster than the others by taking advantage of the filtering power of IDAG-graph DP and the efficient search strategy. For the random data graphs, IDAR outperforms IGQuery by up to three orders of magnitude: 121-140, 141-160 in AIDS; 181-200, 201- in NCI; and 201- in PubChem. For the frequent data graphs, IDAR is faster than DGTree by at least one order of magnitude.

Between two existing algorithms, DGTree is generally faster in the random data graphs, but IGQuery shows better performances in the frequent data graphs. This phenomenon may stem from the fact that DGTree and IGQuery are designed to run efficiently for queries with the small and large number of



(a) AIDS (random)

(b) AIDS (frequent)

(c) NCI (random)

(d) NCI (frequent)

(e) PubChem (random)

(f) PubChem (frequent)

Figure 4.14: Query processing time for varying number of vertices in a query graph.

answers, respectively (see **Number of Answer Graphs**). In fact, some queries for the frequent data graphs has far more answers than most queries for the random data graphs since the frequent data graphs are generally smaller and have fewer labels as shown in Table 4.4. The large gap in performances of IGQuery between the random and frequent data graphs may be due to the big difference of their IG sizes in Table 4.4.

Among the frequent data graphs, the gap between IGQuery and IDAR in PubChem is less than those in AIDS and NCI. These gap differences may originate

Table 4.4: Characteristics of data graphs and size of IG (index of IGQuery) in the experiment of Figure 3.11.

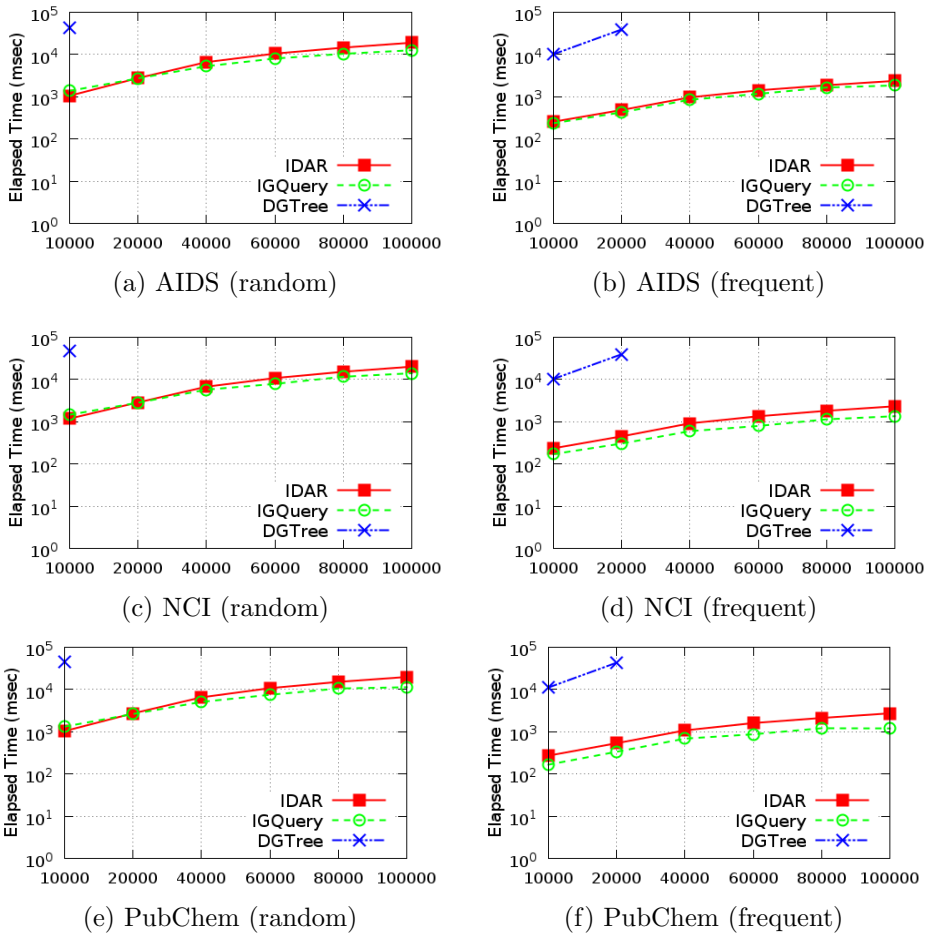
Data graph	Avg $\|E(g)\|$	Var $\|E(g)\|$	$\|\Sigma\|$	$\|E(\mathsf{IG})\|$
AIDS (rand)	52.13	906.82	35	18,690
NCI (rand)	52.25	891.24	46	23,777
PubChem (rand)	52.11	875.61	19	15,406
AIDS (freq)	31.43	14.77	6	238
NCI (freq)	22.86	10.52	6	441
PubChem (freq)	29.88	7.70	6	182

from IGQuery because the average query processing time of IGQuery varies a lot for different datasets (8-284980 msec) whereas that of IDAR is relatively stable (3-134 msec) in Figure 4.14. According to Table 4.4, IGQuery may benefit from the characteristics of PubChem: the frequent data graphs in PubChem have small sizes (i.e., Avg $|E(g)|$) and the least variance of sizes (i.e., Var $|E(g)|$), which leads to IG with the smallest size (i.e., $|E(\mathsf{IG})|$). Indeed, the smaller the size of IG is, the faster is IGQuery in query processing generally in Figure 4.14.

Number of Data Graphs. To test the effect of the number of data graphs on indexing and query processing, we use the sets of 10000, 20000, 40000, 60000, 80000, and 100000 data graphs.

The indexing time for each algorithm is presented in Figure 4.15. The indexing performance of IDAR is on par with that of IGQuery in most cases; however, DGTree cannot manage to construct the index for more than 10000 random data graphs, and more than 20000 frequent data graphs because of its high memory usage to store all (or some) embeddings of feature graphs in each data graph.

Figure 4.16 shows the average query processing time. Since IGQuery cannot solve some queries in a reasonable time, we set a time limit of 24 hours for a



(a) AIDS (random)

(b) AIDS (frequent)

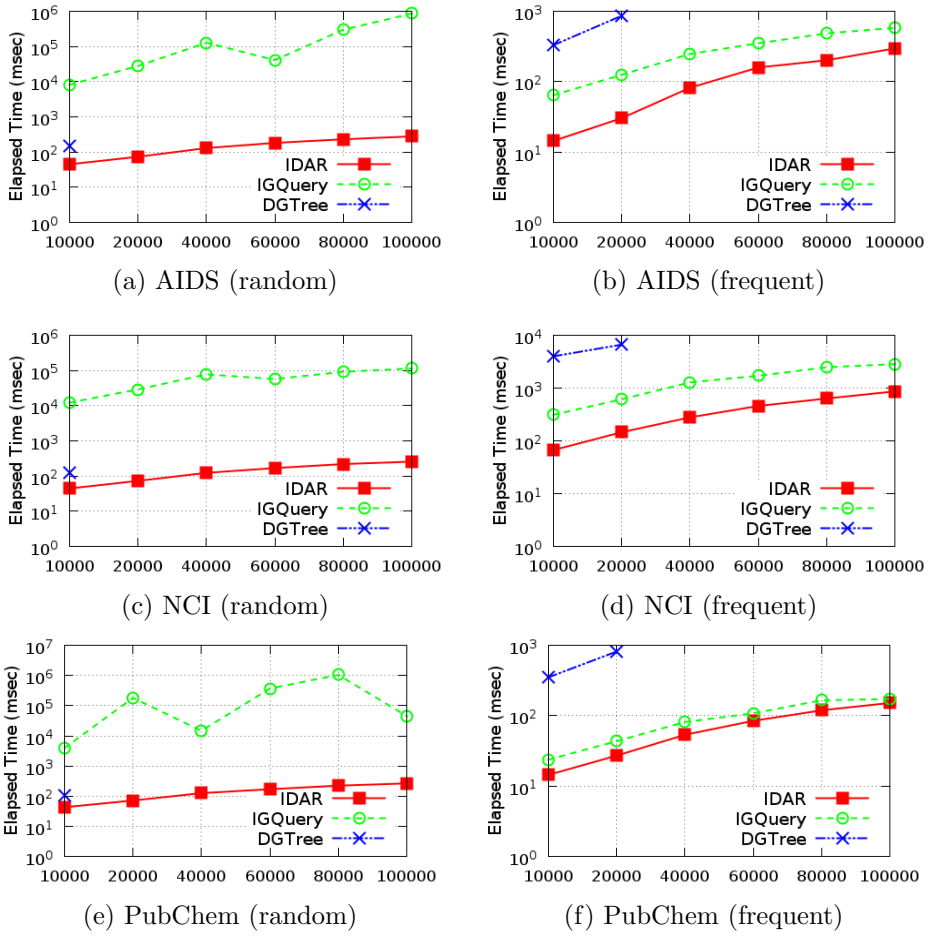
(c) NCI (random)

(d) NCI (frequent)

(e) PubChem (random)

(f) PubChem (frequent)

Figure 4.15: Indexing time for varying number of data graphs.



(a) AIDS (random)

(b) AIDS (frequent)

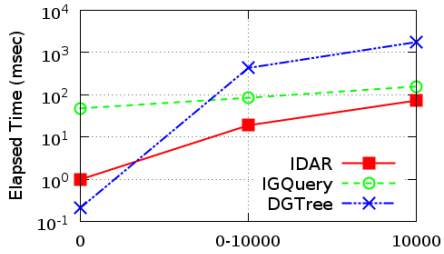
(c) NCI (random)

(d) NCI (frequent)

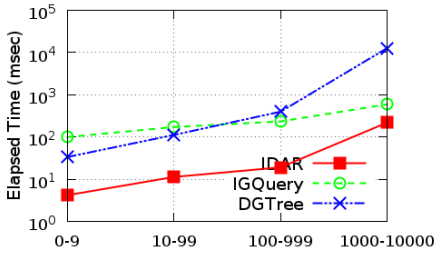
(e) PubChem (random)

(f) PubChem (frequent)

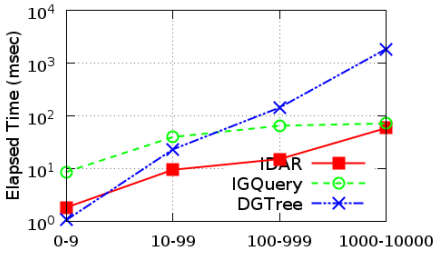
Figure 4.16: Query processing time for varying number of data graphs.



(a) AIDS (frequent)



(b) NCI (frequent)



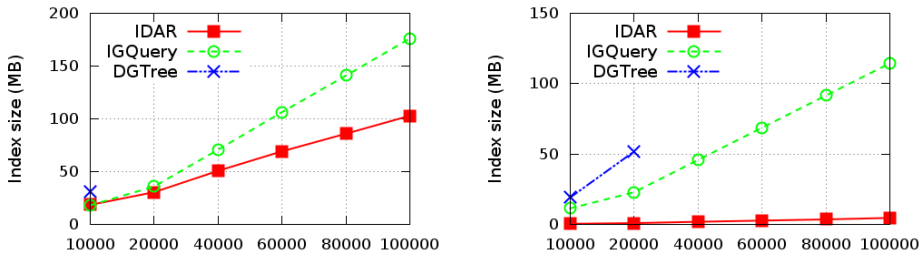
(c) PubChem (frequent)

Figure 4.17: Query processing time for varying number of answers.

query graph, and record the processing time of the query that does not finish within the time limit as 24 hours for comparison. IDAR outperforms IGQuery by up to three orders of magnitude in random graphs. For the frequent data graphs, IDAR outperforms DGTree by up to one order of magnitude. Moreover, IDAR is faster than IGQuery in all cases.

Number of Answer Graphs. We measure the query processing time for different numbers of answer graphs: 0-9, 10-99, 100-999, 1000-10000 for frequent data graphs in NCI and PubChem; and 0, 0-10000, 10000 for frequent data graphs in AIDS (we set three ranges because the number of answers is not evenly distributed). Since the number of answers for the random data graphs is less diverse, the query processing time remains relatively constant, so we mainly consider the frequent data graphs.

Figure 4.17 shows the results. Between IGQuery and DGTree, a better performer changes as the number of answers grows. For the small number of an-



(a) AIDS (random) (b) AIDS (frequent)

Figure 4.18: Index size for varying number of data graphs.

swers, DGTree outperforms IGQuery because DGTree can efficiently filter many false answers by utilizing diverse small features. For the large number of answers, IGQuery performs better because it outputs many answers by using direct inclusion, which can save the cost of subsequent filtering and verification.

IDAR consistently outperforms the others except for 0 in AIDS and 0-9 in PubChem where most query graphs have no answers. These queries are easy instances answered within average 10 msec for IDAR and DGTree (DGTree performs well especially for a query graph with no answers).

Index Size. Figure 4.18 demonstrates the size of each index for varying the number of data graphs in AIDS (the results for the other datasets are similar). In general IDAR is a better performer than others. The gap between IDAR and IGQuery grows as the number of data graphs increases, which means that IDAR is more effective in integrating numerous data graphs thanks to DAG integration. Especially in frequent data graphs, IDAR benefits from a small number of root properties in data DAGs and a large commonality among data graphs, which leads to fewer IDAGs than those for random data graphs.

Chapter 5

Conclusion

5.1 Summary

We have discussed the efficient algorithms for graph query processing.

In Chapter 2, we present the definitions of three fundamental graph query processing problems, i.e., subgraph search, subgraph matching and supergraph search. We also survey previous work for these problems.

In Chapter 3, we studied a new subgraph search algorithm $\mathsf{ELP}_{\mathsf{SS}}$ that combines three techniques, which are extended DAG-graph DP using neighbor-safety, matching degree-one query vertices adaptively, and pruning repetitive search space by using equivalence sets. Furthermore, the three techniques 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.

In Chapter 4, we propose a new supergraph search algorithm IDAR using DAG integration. In existing algorithms, index construction of filtering approaches are computationally expensive, and search methods can cause re-

dundant computations. We introduce four new concepts to address these limitations: (1) DAG integration, (2) dynamic programming between integrated DAG and graph, (3) active-first search, and (4) relevance-size order, which together lead to a much faster ans scalable algorithm for supergraph search. Extensive experiments on real datasets show that our approach outperforms the state-of-the-art algorithms by up to several orders of magnitude in terms of indexing time and query processing time.

5.2 Future Directions

Since subgraph matching, subgraph search and supergraph search are fundamental problems of graph query processing, our approaches can be extended to other applications, environments, or related problems. We suggest several future directions where the concepts or methods of this thesis may be utilized.
Practical Applications. Since our approaches are designed to improve average performance of many query graphs on diverse real-world data graphs, they are not targeted at specific real datasets; that is, our algorithms are very efficient in reducing the running time for difficult input graphs that degrade the overall performance. However, in practical use we may be guaranteed to take only graphs that follow specific characteristics as input.

On one hand, we can improve the performance of our algorithms on specific datasets by capturing their complex nature, such as betweenness centrality, clustering coefficients, neighbor label distribution of each label, and degree (or label) distribution. For example, given a label of a vertex, some labels are not allowed to be adjacent to the label in protein-protein interaction (PPI) networks. Most social networks and PPI networks are reported to be *scale-free*, i.e., degree distribution follows a power law, whereas the degree distribution of Delaunay triangulation networks follows the Gaussian distribution in most cases [57] (Delaunay triangulation for a given set P of discrete points in a plane is a triangulation such that no point in P is inside the circumcircle of

any triangle in the triangulation). While a PPI network exhibit *small-world property*, meaning that the mean shortest path distance between two vertices is small relative to the total number of vertices in the network [61], "six-degrees-of-separation" networks do not have this property [88]. For the subgraph search or subgraph matching problems in scale-free networks, one can match a query vertex to a *hub* (i.e., a vertex with a number of links that greatly exceeds the average) first if hubs exist among candidate vertices of the query vertex. Moreover, one can obtain a matching order that uses the label constraints or the label distributions above. In RDF query processing a global matching order may perform better than an adaptive matching order, because unlike other typical real-world datasets RDF graphs are well-structured, and thus generating a matching order for each region is ineffective [39].

On the other hand, we can modify our algorithms to efficiently process specific query graphs. First, triangle counting is a famous social network analysis application to detect communities or measure the cohesiveness of those communities, and it is also exploited to compute clustering coefficients. Therefore numerous algorithms have been suggested to find the occurrences of a triangle query in a data graph. Second, RDF represents data as a set of triple (subject, predicate, object), and multiple triples that have different predicates (or objects) may have a same subject in common; thus *star queries* are very common in RDF query processing where a star query is a query graph with a central vertex and its neighbors [110]. In addition, chain queries are also common because the subject of a triple pattern can be joined to the object of another triple pattern so their triple patterns are connected one by one like a chain [49].

Scalability. Processing graph queries in web-scale conditions like social networks (e.g., Twitter [43]) or recommendation systems has been an essential problem for many graph query processing algorithms [75, 53]. An increasing number of studies addressed the problems of graph query processing in a large-scale data by taking advantage of a parallel or distributed environment. Parallel

subgraph matching using Graphics Processing Units (GPUs) was suggested in [82, 28]. Since many real-world large-scale graphs are unlabeled, recent studies have focused on solving the *subgraph enumeration* problem (subgraph matching in unlabeled graphs) where given an unlabeled query graph q and an unlabeled data graph G subgraph enumeration is to find all subgraphs in G that are isomorphic to q. Some researchers proposed an efficient parallel algorithm for subgraph enumeration in a single machine [73]. Further, a number of studies tackled this problem in a distributed environment [66, 46, 34, 16, 1, 45, 44, 71]. Extending our techniques to parallel or distributed platforms would be an interesting future work.

Dynamic Graphs. Another challenge lies in managing dynamic graphs where vertices or edges in the graphs are inserted or deleted.

Processing queries on dynamic graphs has been extensively studied. Researchers proposed efficient algorithms for *continuous subgraph matching* where a query graph q, a data graph G, and a set of edges to be inserted into or deleted from G are given [40, 20, 12]. One can leverage our techniques for subgraph query processing and subgraph matching to deal with dynamic graphs.

In our supergraph search algorithm, DAG integration can be extended for dynamic graph databases as follows. To insert a data graph g, we build a DAG from g, select an IDAG to which the DAG will be integrated (based on the partitioning rule), and integrate the DAG into the IDAG. To delete g from IDAG I to which g is integrated, we remove g from the root of I, remove g from the outgoing edges of the root, and repeat for the children of the root.

Graph Matching and Graph Pattern Matching. Given two graphs G and H (with the same number of vertices), graph matching is to find a one-to-one mapping f from $V(G)$ to $V(H)$ such that $(u, v) \in E(G)$ iff $(f(u), f(v)) \in E(H)$. Subgraph isomorphism (i.e., "Given two graphs q and G, does G contain a subgraph isomorphic to q?") is a one of the most famous graph matching problems. However, there are a number of interesting results on exact and inexact

graph matching problems: graph isomorphism [56, 55, 14] and graph similarity [24, 9, 103, 27, 105, 104, 97].

Given a pattern graph q and a data graph G, *graph pattern matching* is to find all *matches* of q in G. There are two different well-known problems in graph pattern matching, depending on how we define a match of q in G: subgraph matching and graph simulation [52, 18, 21]. Meanwhile, some recent studies have focused on the problem of top-k diversified subgraph matching [94, 19], which asks for a set of up to k subgraphs in a data graph G isomorphic to a query graph q such that the subgraphs cover the largest number of vertices in G.

Since we have already addressed the subgraph matching problem in the thesis, one may apply our work to the problems above.

Graph Neural Networks. Several NP-hard problems on graphs such as graph matching (i.e., quadratic assignment problem) and the traveling salesman problem (TSP) are well-known combinatorial optimization problems, where the goal is to find the optimum of an objective function whose domain is a discrete space. In order to heuristically find near-optimal solutions, existing work rely on (1) traditional techniques (such as branch-and-bound and tabu search) or (2) deep learning methods like graph neural networks (GNNs) [3, 22].

Since the first model of GNN appeared [68], numerous GNN variants [41, 29, 84, 90, 59, 6, 95] recently have achieved state-of-the-art performance in common tasks (e.g., node classification and graph classification) to assess their performance. However, there are two challenges to apply GNNs into practical use. First, the design of many GNNs is based on empirical intuition and experimental trial-and-error, so they have theoretically limited representation power [10, 54, 60, 90, 36]. Second, scaling up GNNs is essential to deal with large graphs but difficult due to the computational overhead of the core steps in GNNs [106, 89].

A natural question arises. How could we develop GNN models that is highly

expressive while maintaining scalability as much as possible? In order to answer the question, one may study a theoretical framework for enhancing or analyzing the representation power of GNNs by finding the close connection between GNNs and the novel concepts of our approaches.

Bibliography

[1] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proceedings of the International Conference on Very Large Data Bases*, 11(6):691–704, 2018.

[2] D. Babić, D. Reynaud, and D. Song. Malware analysis with tree automata inference. In *Proceedings of the International Conference on Computer Aided Verification*, pages 116–131, 2011.

[3] Y. Bai, H. Ding, S. Bian, T. Chen, Y. Sun, and W. Wang. Simgnn: A neural network approach to fast graph similarity computation. In *Proceedings of the International Conference on Web Search and Data Mining*, pages 384–392, 2019.

[4] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing Cartesian products. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1199–1214, 2016.

[5] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha. Enhancing graph database indexing by suffix tree structure. In *Proceedings of the IAPR International Conference on Pattern Recognition in Bioinformatics*, pages 195–203. Springer, 2010.

[6] X. Bresson and T. Laurent. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553*, 2017.

[7] M. Cannataro and P. H. Guzzi. *Data management of protein interaction networks*, volume 17. John Wiley & Sons, 2012.

[8] C. Chen, X. Yan, P. S. Yu, J. Han, D.-Q. Zhang, and X. Gu. Towards graph containment search and indexing. In *Proceedings of the International Conference on Very Large Data Bases*, pages 926–937, 2007.

[9] X. Chen, H. Huo, J. Huan, J. S. Vitter, W. Zheng, and L. Zou. MSQ-Index: A succinct index for fast graph similarity search. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[10] Z. Chen, S. Villar, L. Chen, and J. Bruna. On the equivalence between graph isomorphism testing and function approximation with gnns. In *Advances in Neural Information Processing Systems*, pages 15868–15876, 2019.

[11] J. Cheng, Y. Ke, A. W.-C. Fu, and J. X. Yu. Fast graph query processing with a low-cost index. *The International Journal on Very Large Data Bases*, 20(4):521–539, 2011.

[12] S. Choudhury, L. Holder, G. Chin, K. Agarwal, and J. Feo. A selectivity based approach to continuous pattern detection in streaming graphs. In *Proceedings of the International Conference on Extending Database Technology*, pages 157–168, 2015.

[13] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[14] D. G. Corneil and C. C. Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the ACM (JACM)*, 17(1):51–64, 1970.

[15] R. Di Natale, A. Ferro, R. Giugno, M. Mongiovì, A. Pulvirenti, and D. Shasha. Sing: Subgraph search in non-homogeneous graphs. *BMC bioinformatics*, 11(1):96, 2010.

[16] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The LDBC social network benchmark: Interactive workload. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 619–630, 2015.

[17] W. Fan. Graph pattern matching revised for social network analysis. In *Proceedings of the International Conference on Database Theory*, pages 8–21, 2012.

[18] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: from intractable to polynomial time. *Proceedings of the International Conference on Very Large Data Bases*, 3(1-2):264–275, 2010.

[19] W. Fan, X. Wang, and Y. Wu. Diversified top-k graph pattern matching. *Proceedings of the International Conference on Very Large Data Bases*, 6(13):1510–1521, 2013.

[20] W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *ACM Transactions on Database Systems*, 38(3):18:1–18:47, 2013.

[21] W. Fan, X. Wang, Y. Wu, and D. Deng. Distributed graph simulation: Impossibility and possibility. *Proceedings of the International Conference on Very Large Data Bases*, 7(12):1083–1094, 2014.

[22] M. Fey, J. E. Lenssen, C. Morris, J. Masci, and N. M. Kriege. Deep graph matching consensus. *Proceedings of the International Conference on Learning Representations*, 2020.

[23] M. K. Ganapathiraju, M. Thahir, A. Handen, S. N. Sarkar, R. A. Sweet, V. L. Nimgaonkar, C. E. Loscher, E. M. Bauer, and S. Chaparala.

Schizophrenia interactome with 504 novel protein–protein interactions. *NPJ schizophrenia*, 2(1):1–10, 2016.

[24] X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. *Pattern Analysis and applications*, 13(1):113–129, 2010.

[25] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[26] R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, and D. Shasha. Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. *PloS one*, 8(10):e76911, 2013.

[27] K. Gouda and M. Hassaan. CSI_GED: An efficient approach for graph edit similarity computation. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 265–276. IEEE, 2016.

[28] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K.-L. Tan. Gpu-accelerated subgraph enumeration on partitioned graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1067–1082, 2020.

[29] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

[30] M. Han, H. Kim, G. Gu, K. Park, and W.-S. Han. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adpative Matching Order, and Failing Set Together. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1429–1446, 2019.

[31] W.-S. Han, J. Lee, and J.-H. Lee. Turbo iso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of*

the ACM SIGMOD International Conference on Management of Data, pages 337–348, 2013.

[32] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. iGraph: a framework for comparisons of disk-based graph indexing techniques. *Proceedings of the International Conference on Very Large Data Bases*, 3(1-2):449–459, 2010.

[33] H. He and A. K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 405–418, 2008.

[34] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. Prat-Pérez, T. Manhardto, H. Chafio, M. Capotă, N. Sundaram, M. Anderson, I. G. Tănase, Y. Xia, L. Nai, and P. Boncz. LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proceedings of the International Conference on Very Large Data Bases*, 9(13):1317–1328, 2016.

[35] F. Katsarou, N. Ntarmos, and P. Triantafillou. Performance and scalability of indexed subgraph query processing methods. *Proceedings of the International Conference on Very Large Data Bases*, 8(12):1566–1577, 2015.

[36] N. Keriven and G. Peyré. Universal invariant and equivariant graph neural networks. In *Advances in Neural Information Processing Systems*, pages 7090–7099, 2019.

[37] H. Kim, Y. Choi, K. Park, X. Lin, S.-H. Hong, and W.-S. Han. Fast subgraph query processing and subgraph matching. *work in progress*, 2020.

[38] H. Kim, S. Min, K. Park, X. Lin, S.-H. Hong, and W.-S. Han. IDAR: Fast supergraph search using DAG integration. *Proceedings of the International Conference on Very Large Data Bases*, 13(9):1456–1468, 2020.

[39] J. Kim, H. Shin, W.-S. Han, S. Hong, and H. Chafi. Taming subgraph isomorphism for rdf query processing. *Proceedings of the International Conference on Very Large Data Bases*, 8(11), 2015.

[40] K. Kim, I. Seo, W.-S. Han, J.-H. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 411–426, 2018.

[41] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *Proceedings of the International Conference on Learning Representations*, 2017.

[42] K. Klein, N. Kriege, and P. Mutzel. CT-index: Fingerprint-based graph indexing combining cycles and trees. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 1115–1126, 2011.

[43] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proceedings of the International Conference on World Wide Web*, pages 591–600, 2010.

[44] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in MapReduce. *Proceedings of the International Conference on Very Large Data Bases*, 8(10):974–985, 2015.

[45] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang. Scalable distributed subgraph enumeration. *Proceedings of the International Conference on Very Large Data Bases*, 10(3):217–228, 2016.

[46] L. Lai, Z. Qing, Z. Yang, X. Jin, Z. Lai, R. Wang, K. Hao, X. Lin, L. Qin, W. Zhang, et al. Distributed subgraph matching on timely dataflow. *Proceedings of the International Conference on Very Large Data Bases*, 12(10):1099–1112, 2019.

[47] A. Lamb. How to find communities online using social network analysis, 2013. https://econsultancy.com/how-to-find-communities-online-using-social-network-analysis/ .

[48] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proceedings of the International Conference on Very Large Data Bases*, 6(2):133–144, 2012.

[49] K. Lee and L. Liu. Scaling queries over big RDF graphs with semantic hash partitioning. *Proceedings of the International Conference on Very Large Data Bases*, 6(14):1894–1905, 2013.

[50] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection, 2014. http://snap.stanford.edu/data.

[51] B. Lyu, L. Qin, X. Lin, L. Chang, and J. X. Yu. Scalable supergraph search in large graph databases. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 157–168, 2016.

[52] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Strong simulation: Capturing topology in graph pattern matching. *ACM Transactions on Database Systems*, 39(1):4, 2014.

[53] T. Ma, S. Yu, J. Cao, Y. Tian, A. Al-Dhelaan, and M. Al-Rodhaan. A comparative study of subgraph matching isomorphic methods in social networks. *IEEE Access*, 6:66621–66631, 2018.

[54] H. Maron, H. Ben-Hamu, H. Serviansky, and Y. Lipman. Provably powerful graph networks. In *Advances in Neural Information Processing Systems*, pages 2153–2164, 2019.

[55] B. D. McKay et al. *Practical graph isomorphism*. Department of Computer Science, Vanderbilt University Tennessee, USA, 1981.

[56] B. D. McKay and A. Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60:94–112, 2014.

[57] G. Mei, N. Xu, and S. Cuomo. Degree distribution of delaunay triangulations. *arXiv preprint arXiv:1805.08063*, 2018.

[58] A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proceedings of the International Conference on Very Large Data Bases*, 12(11):1692–1704, 2019.

[59] F. Monti, D. Boscaini, J. Masci, E. Rodola, J. Svoboda, and M. M. Bronstein. Geometric deep learning on graphs and manifolds using mixture model CNNs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5115–5124, 2017.

[60] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4602–4609, 2019.

[61] M. E. Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003.

[62] S. Nowozin, K. Tsuda, T. Uno, T. Kudo, and G. BakIr. Weighted substructure mining for image analysis. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2007.

[63] H. Park and M.-S. Kim. Evograph: an effective and efficient graph upscaling method for preserving graph properties. In *Proceedings of ACM SIGKDD International Conference on Knowledge discovery and data mining*, pages 2051–2059, 2018.

[64] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the International Conference on Very Large Data Bases*, 8(5):617–628, 2015.

[65] X. Ren and J. Wang. Multi-query optimization for subgraph isomorphism search. *Proceedings of the International Conference on Very Large Data Bases*, 10(3):121–132, 2016.

[66] X. Ren, J. Wang, W.-S. Han, and J. X. Yu. Fast and robust distributed subgraph enumeration. *Proceedings of the International Conference on Very Large Data Bases*, 12(11):1344–1356, 2019.

[67] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the International Conference on Very Large Data Bases*, 11(4):420–431, 2017.

[68] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

[69] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proceedings of the International Conference on Very Large Data Bases*, 1(1):364–375, 2008.

[70] H. Shang, K. Zhu, X. Lin, Y. Zhang, and R. Ichise. Similarity search on supergraph containment. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 637–648, 2010.

[71] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 625–636, 2014.

[72] T. A. Snijders, P. E. Pattison, G. L. Robins, and M. S. Handcock. New specifications for exponential random graph models. *Sociological methodology*, 36(1):99–153, 2006.

[73] S. Sun, Y. Che, L. Wang, and Q. Luo. Efficient parallel subgraph enumeration on a single machine. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 232–243, 2019.

[74] S. Sun and Q. Luo. Scaling up subgraph query processing with efficient subgraph matching. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 220–231, 2019.

[75] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *Proceedings of the International Conference on Very Large Data Bases*, 5(9):788–799, 2012.

[76] The National Cancer Institute (NCI). The AIDS antiviral screen dataset. http://dtp.nci.nih.gov.

[77] The National Cancer Institute (NCI). The National Cancer Institute database. http://cactus.nci.nih.gov/download/nci/index.html.

[78] The National Institutes of Health (NIH). PubChem. https://pubchem.ncbi.nlm.nih.gov.

[79] Y. Tian, R. C. Mceachin, C. Santos, D. J. States, and J. M. Patel. Saga: a subgraph matching tool for biological graphs. *Bioinformatics*, 23(2):232–239, 2006.

[80] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 963–972, 2008.

[81] Y. Tong, X. Zhang, C. C. Cao, and L. Chen. Efficient probabilistic supergraph search over large uncertain graphs. In *Proceedings of the ACM International Conference on Information and Knowledge Management*, pages 809–818, 2014.

[82] H.-N. Tran, J. Kim, and B. He. Fast subgraph matching on large graphs using graphics processors. In *Proceedings of the International Conference on Database Systems for Advanced Applications*, pages 299–315, 2015.

[83] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[84] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *Proceedings of the International Conference on Learning Representations*, 2018.

[85] J. Wang, N. Ntarmos, and P. Triantafillou. Graphcache: a caching system for graph queries. In *Proceedings of the International Conference on Extending Database Technology*, pages 13–24, 2017.

[86] G. Weikum, G. Kasneci, M. Ramanath, and F. Suchanek. The Future of DB & IR. 2020.

[87] Wikipedia. Raspberry ellagitannin, 2020. https://en.wikipedia.org/wiki/Raspberry_ellagitannin.

[88] Wikipedia. Small-world network, 2020. https://en.wikipedia.org/wiki/Small-world_network.

[89] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[90] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *Proceedings of the International Conference on Learning Representations*, 2019.

[91] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the IEEE International Conference on Data Mining*, pages 721–724, 2002.

[92] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 335–346, 2004.

[93] P. Yanardag and S. Vishwanathan. Deep graph kernels. In *Proceedings of SIGKDD*, pages 1365–1374, 2015.

[94] Z. Yang, A. W.-C. Fu, and R. Liu. Diversified top-k subgraph querying in a large graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1167–1182, 2016.

[95] Z. Ying, J. You, C. Morris, X. Ren, W. Hamilton, and J. Leskovec. Hierarchical graph representation learning with differentiable pooling. In *Advances in Neural Information Processing Systems*, pages 4800–4810, 2018.

[96] D. Yuan, P. Mitra, and C. L. Giles. Mining and indexing graphs for supergraph search. *Proceedings of the International Conference on Very Large Data Bases*, 6(10):829–840, 2013.

[97] Y. Yuan, G. Wang, L. Chen, and H. Wang. Graph similarity search on large uncertain graph databases. *The International Journal on Very Large Data Bases*, 24(2):271–296, 2015.

[98] S. Zhang, J. Li, H. Gao, and Z. Zou. A novel approach for efficient supergraph query processing on graph databases. In *Proceedings of the International Conference on Extending Database Technology: Advances in Database Technology*, pages 204–215, 2009.

[99] S. Zhang, S. Li, and J. Yang. GADDI: Distance index based subgraph matching in biological networks. In *Proceedings of the International Conference on Extending Database Technology*, pages 192–203, 2009.

[100] W. Zhang, X. Lin, Y. Zhang, K. Zhu, and G. Zhu. Efficient probabilistic supergraph search. *IEEE Transactions on Knowledge and Data Engineering*, 28(4):965–978, 2015.

[101] P. Zhao and J. Han. On graph query optimization in large networks. *Proceedings of the International Conference on Very Large Data Bases*, 3(1):340–351, 2010.

[102] P. Zhao, J. X. Yu, and S. Y. Philip. Graph indexing: Tree+ delta ¿= graph. In *Proceedings of the International Conference on Very Large Data Bases*, pages 938–949, 2007.

[103] X. Zhao, C. Xiao, X. Lin, W. Wang, and Y. Ishikawa. Efficient processing of graph similarity queries with edit distance constraints. *The International Journal on Very Large Data Bases*, 22(6):727–752, 2013.

[104] W. Zheng, L. Zou, X. Lian, D. Wang, and D. Zhao. Graph similarity search with edit distance constraint in large graph databases. In *Proceedings of the ACM international conference on Information & Knowledge Management*, pages 1595–1600, 2013.

[105] W. Zheng, L. Zou, X. Lian, D. Wang, and D. Zhao. Efficient graph similarity search over large graph databases. *IEEE Transactions on Knowledge and Data Engineering*, 27(4):964–978, 2014.

[106] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[107] G. Zhu, X. Lin, W. Zhang, W. Wang, and H. Shang. Prefindex: An efficient supergraph containment search technique. In *Proceedings of International Conference on Scientific and Statistical Database Management*, pages 360–378, 2010.

[108] Q. Zhu, J. Yao, S. Yuan, F. Li, H. Chen, W. Cai, and Q. Liao. Superstructure searching algorithm for generic reaction retrieval. *Journal of Chemical Information and Modeling*, 45(5):1214–1222, 2005.

[109] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *Proceedings of the International Conference on Extending Database Technology*, pages 181–192, 2008.

[110] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: Answering SPARQL Queries via Subgraph Matching. *Proceedings of the International Conference on Very Large Data Bases*, 4(8):482–493, 2011.

요약

몇 십 년 전부터 다양한 그래프 데이터가 공개되면서, NP-hard 그래프 쿼리 프로세싱 문제들을 위한 실용적인 애플리케이션을 개발하는 데 방대한 노력이 투입되었다. 이러한 노력에도 불구하고 현존하는 알고리즘들은 대용량 혹은 대량의 그래프를 다루는 데 한계를 보여준다. 본 논문에서는 부분그래프 검색, 부분그래프 매칭, 수퍼그래프 검색 등 그래프 쿼리를 처리하는 중요한 문제들을 다룬다.

첫 번째로, 본 논문에서는 부분그래프 쿼리 처리와 부분그래프 매칭을 위한 빠른 알고리즘들을 제안한다. 이를 위해 동적 프로그래밍 (dynamic programming)을 포함하여 세 가지 고급 테크닉을 사용한다. 실제 데이터과 합성 데이터에 대한 실험을 통해 본 알고리즘들이 쿼리 처리 시간에서 현존하는 가장 빠른 알고리즘들보다 최대 수백 배에서 수십만 배 빠름을 검증하였다.

두 번째로, 본 논문에서는 수퍼그래프 검색을 위한 빠르고 확장성 있는 알고리즘을 개발한다. 이를 위해 동적 프로그래밍을 비롯하여 네 가지 새로운 테크닉을 이용한다. 본 논문에서는 실제 데이터를 대상으로 한 방대한 실험을 통해 본 알고리즘이 인덱싱 시간과 쿼리 처리 시간에서 현존하는 가장 빠른 알고리즘보다 최대 수천 배 빠르다는 사실을 보였다.

주요어: 그래프 쿼리 프로세싱; 부분그래프 검색; 부분그래프 쿼리 프로세싱; 부분그래프 매칭; 수퍼그래프 검색; 부분그래프 동형; 동적 프로그래밍; 백트래킹; 동적 매칭 순서

학번: 2013-23112