IDAR: Fast Supergraph Search Using DAG Integration

Hyunjoon Kim[†] Seunghwan Min[†] Kunsoo Park[*] [†] Xuemin Lin[‡]
Seok-Hee Hong[§] Wook-Shin Han[*][¶]

[†] Seoul National University, [‡] University of New South Wales, [§] University of Sydney,
[¶] Pohang University of Science and Technology (POSTECH)

{hjkim,shmin,kpark}@theory.snu.ac.kr, lxue@cse.unsw.edu.au,
seokhee.hong@sydney.edu.au, wshan@dblab.postech.ac.kr

ABSTRACT

Supergraph search is one of fundamental graph query processing problems in many application domains. Given a query graph and a set of data graphs, supergraph search is to find all the data graphs contained in the query graph as subgraphs. In existing algorithms, index construction or filtering approaches are computationally expensive, and search methods can cause redundant computations. In this paper, 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 and scalable algorithm for supergraph search. 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.

PVLDB Reference Format:
Hyunjoon Kim, Seunghwan Min, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, Wook-Shin Han. IDAR: Fast Supergraph Search Using DAG Integration. *PVLDB*, 13(9): 1456 - 1468, 2020.
DOI: https://doi.org/10.14778/3397230.3397241

1. INTRODUCTION

For the last two decades, much research has been carried out on practical graph query processing as various public graph data attracted great attention in numerous application domains [15]. The public graph data can be categorized into two types depending on their applications: large graphs such as social networks and Resource Description Framework data, and smaller graphs such as chemical compounds and protein-protein interaction (PPI) networks. There are two fundamental graph query processing problems in the latter, i.e., *subgraph search* [11, 3, 8, 17] and *supergraph search* [5, 23, 25, 6, 12]. 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 in Q as subgraphs.

In this study, we focus on the supergraph search problem. Supergraph search is widely present in real-world applications such as chemical compound search in cheminformatics [26], PPI network analysis in bioinformatics [18, 19, 4], shape matching in image processing [14, 5], and malware detection [1]. Given a PPI network of one species and a database of structural motifs, researchers in bioinformatics find the motifs contained in the PPI network [18, 19, 4]. In cheminformatics, molecules are modeled as undirected labeled graphs where vertices represent atoms and edges represent chemical bonds. This problem is used in the process of synthesizing new compounds with well-known chemical molecules. For a set of chemical compounds with already known functionalities, a query graph is given as a new large compound, then supergraph search outputs a set of the compounds contained in the query, which help researchers predict chemical functionalities of the new compound [26]. Indeed, the National Institutes of Health (NIH) provides a web user interface for supergraph search of chemical compounds in an open chemistry database called PubChem[1].

Supergraph search is NP-hard since it includes finding subgraph isomorphism which is an NP-hard problem [7]. Therefore, solving supergraph search is a bottleneck in overall performance of the applications. Such applications give challenges to this problem: 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.

Related Work. 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 [5], GPTree [23], and PrefIndex [25]) 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., fre-

[*]contact author

Proceedings of the VLDB Endowment, Vol. 13, No. 9
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3397230.3397241

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

quent 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.

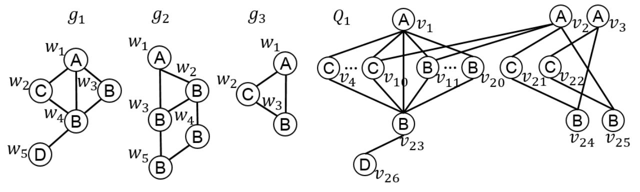
In order to address these problems, IGQuery [6] heuristically finds some answer graphs by using its index, and then runs a filtering-and-verification method in a set of remaining data graphs to reduce the cost of both filtering and verification. First, IGQuery merges a set of data graphs into an *integrated 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 [12] 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) 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 [10, 2, 9, 13] 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 [16, 22]. Recently, probabilistic supergraph search has been studied to handle uncertain data graphs with probability on edges or vertices [20, 24].

Challenges. 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 [21, 5]; 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 index, which takes exponential time in the worst case.



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

Figure 1: Data graphs and a query graph

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 1, 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.

Contributions. To address the limitations above, we introduce the following new ideas, which lead to a faster and scalable algorithm 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 [9] 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 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.

Organization. The remainder of the paper is organized as follows. Section 2 provides useful definitions of the concepts in our techniques and the supergraph search problem. Section 3 presents a brief overview of our algorithm. Section 4 describes DAG integration by which we merge input data graphs. Section 5 presents three new techniques used in query processing. Section 6 discusses the results of performance evaluation. Finally, Section 7 concludes the paper.

2. PRELIMINARIES

For simplicity of presentation, 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 $l_g : V(g) \cup E(g) \rightarrow \Sigma$ that assigns a label to each vertex or edge, where Σ is a set of labels. For a subset Y of $V(g)$, the *induced subgraph* $g[Y]$ denotes the subgraph of g whose vertex set is Y and whose edge set consists of all the edges in $E(g)$ that have both endpoints in Y.

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 are DAGs of g_1 in Figure 1). 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 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) and Parent(u) denote the children and parents of u, respectively.

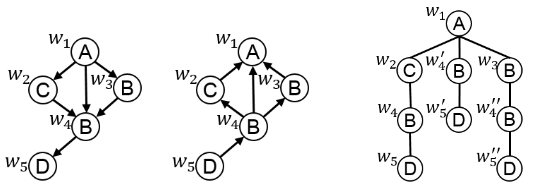
Definition 2.1. Given a graph $Q = (V(Q), E(Q), l_Q)$ and a graph $g = (V(g), E(g), l_g)$, an *embedding* of g in Q is a mapping $M : V(g) \rightarrow V(Q)$ such that (1) M is injective (i.e., $M(u) \neq M(u')$ for $u \neq u'$ in $V(g)$), (2) $l_g(u) = l_Q(M(u))$ for every $u \in V(g)$, and (3) $(M(u), M(u')) \in E(Q)$ and $l_g(u, u') = l_Q(M(u), M(u'))$ for every $(u, u') \in E(g)$.

We say that g is *subgraph-isomorphic* to Q, denoted by $g \subseteq Q$, if there exist an embedding of g in Q. An embedding of an induced subgraph of g in Q is called a *partial embedding.* A mapping that satisfies (2) and (3) is called a *homomorphism.*

Definition 2.2. 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 2c is the path tree of g in Figure 2a).

Definition 2.3. 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 $M'(u) = v$.

Definition 2.4. 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.



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

Figure 2: DAGs and path tree

Problem Statement. 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 1, 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 (i.e.,"Does Q contain a subgraph isomorphic to g?") is NP-complete [7], the supergraph search problem is NP-hard.

Table 1 lists the notations frequently used in the paper.

Table 1: Frequently used notations

Symbol	Description
D	Set of data graphs
D'	Set of data DAGs
Q	Query graph
I	Integrated DAG
A_Q	Answer set for Q
$h_i(w)$	Mapping of w in partial merging $h_i : V(g_i) \rightarrow V(I)$ where $g_i \in D$
$f(u)$	Mapping of u in a partial feasible mapping $f : V(I) \rightarrow V(Q)$
$C(u)$	Set of candidate vertices for $u \in V(I)$
$G(u, v)$	Set of candidate graphs for $u \in V(I)$, $v \in V(Q)$

3. OVERVIEW OF OUR ALGORITHM

We first describe our index construction algorithm over a set D of data graphs to build a set of integrated DAGs (IDAGs) in Algorithm 1, which follows the procedures below.

Algorithm 1: BuildIndex

```
Input: a set $D$ of data graphs
Output: a set $D^*$ of integrated DAGs
1 $D' \leftarrow \emptyset$;
2 foreach data graph $g \in D$ do
3     $g' \leftarrow$ BuildDAG($g$); $D' \leftarrow D' \cup \{g'\}$;
4 $\Pi \leftarrow$ Partition($D'$);
5 foreach for each set in $\Pi$ do
6     $I \leftarrow$ empty integrated DAG;
7     foreach data DAG $g'$ in the set do
8         $B \leftarrow$ 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, $\text{NLPF}(u)$ is the frequency of u's distinct neighbor label-pairs, e.g., in Figure 3a $\text{NLPF}(w_1)$ 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 $\text{freq}(l, x)$ 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.
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).

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.

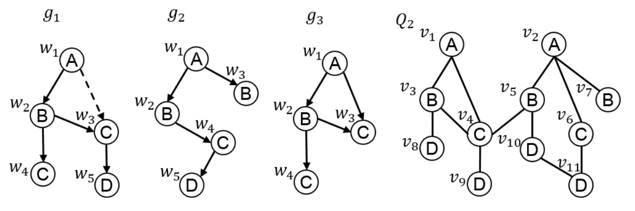
Algorithm 2: SUPERGRAPHSEARCH

```
  Input: query graph $Q$, a set $D^*$ of integrated DAGs
  Output: answer set $A_Q$
1 $A_Q \leftarrow \emptyset$;
2 foreach $I \in D^*$ do
3     ICS ← BUILDICS($I, Q$);
4     $f \leftarrow \emptyset$;
5     BACKTRACK($I$, ICS, $f, A_Q$);
```

The overall framework of query processing is shown in Algorithm 2, which takes a query graph Q and a set D^* of IDAGs, and finds an answer set A_Q for Q.

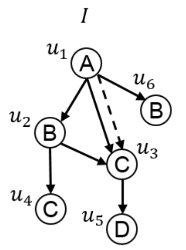
For every IDAG I, we go through two steps as follows.

1. First, BUILDICS is invoked to build an ICS by using dynamic programming between an IDAG and a graph (Section 5.1). 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.



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

Figure 3: 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: The integrated DAG for D' in Figure 3a

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 5.2) and relevance-size order (Section 5.3).

4. 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 shows the IDAG constructed from a set D' of data DAGs in Figure 3a (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. 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 4.1. Given a DAG g and an IDAG I, a *merging of g to I* is defined as an embedding $h : V(g) \rightarrow V(I) \cup V_{new}$ 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 3a 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.

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*.

Algorithm 3: FindMerging

```
   Input: a DAG $g$, an IDAG $I$, bottom-up similarity $B$
1  Update the root $u_r$ of $I$; $h \leftarrow \{(w_r, u_r)\}$;
2  while there is an extendable vertex in $V(g)$ do
3      $(w, u) \leftarrow \arg\max_{(w', u')} sim(w', u')$ for every extendable vertex $w'$ and $u' \in C_h(w')$
4      if $u$ is matched in $h$ then
5          Remove $u$ from $C_h(w)$;
6          if $C_h(w) = \emptyset$ then
7              Create a new vertex $\hat{u}$ of $I$; $h \leftarrow h \cup \{(w, \hat{u})\}$;
8      else
9          Update $u$; $h \leftarrow h \cup \{(w, u)\}$;
```

Definition 4.2. 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 $C_h(w) = \bigcup_{w_p \in \text{Parent}(w)} \hat{N}_w^{w_p}(h(w_p))$, 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.

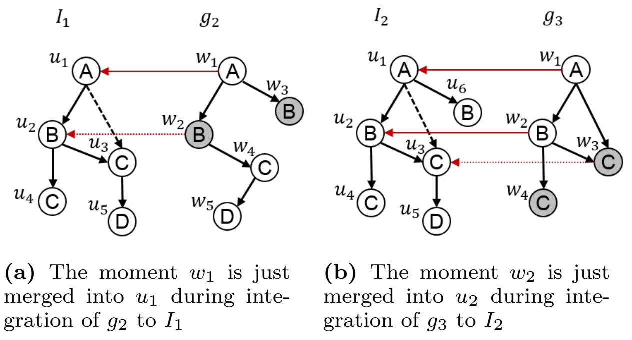
Two questions arise: (1) Among all extendable vertices regarding h, which vertex should be extended first? (2) To which unmapped mergeable candidate in $C_h(w)$ should we merge w? In our integration method, we select an extendable vertex w and a mergeable candidate u at once such that the *similarity* 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 4.1. Figure 5 displays the integration of DAGs in Figure 3a. Assume that we just merged w_1 into u_1 during integration of g_2 to I_1 in Figure 5a. 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 5b, 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.

Definition 4.3. For each $w \in V(g)$ and $u \in V(I)$, the *bottom-up similarity* of w and u is defined as $B(w, u) = score(w, u) + \sum_{w_c \in \text{Child}(w)} \max_{u_c \in \text{Child}(u)} \{B(w_c, u_c)\}$, where $score(w, u)$ is 1 if $l_g(w) = l_I(u)$; 0 otherwise.

Definition 4.4. 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 $T_h(w, u) = score(w, u) + \sum_{w_p \in \text{Parent}(w)} \{T_h(w_p, h(w_p))\}$. 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.



(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 5: Example of DAG integration

Bottom-up similarity B is computed in BottomUpSim in Algorithm 1. 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.

Top-down similarities are computed and a DAG g is integrated into an IDAG I at the same time in Algorithm 3. 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 5a of Example 4.1, 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 5b of Example 4.1, 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 $T_h(w_3, u_3) > T_h(w_4, u_3)$.

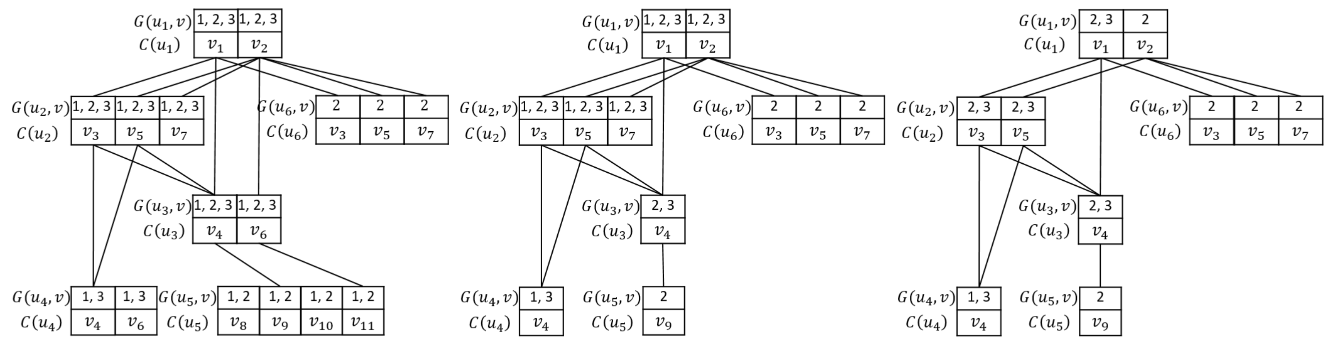
Lemma 4.1. Given a DAG g and an IDAG I, the time and space complexities of DAG integration of g to I are $O(|E(g)||E(I)| + |V(g)||V(I)| \log(|V(g)||V(I)|))$ and $O(|V(g)||V(I)|)$, respectively.

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.

5. QUERY PROCESSING

5.1 Dynamic Programming between IDAG and Graph

DAF [9] 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



(a) Initial ICS **(b)** ICS after 1st refinement **(c)** ICS after 2nd refinement

Figure 6: Refinements of ICS using IDAG-graph DP

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 6a shows the initial ICS on I in Figure 4 and Q_2 in Figure 3b (every number i in $G(u,v)$ represents g_i in Figure 6). 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., $N^{u_1}_{u_6}(v_2) = \{v_5, v_7\}$.

Definition 5.1. 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 5.2. 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 5.3. 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.*

Theorem 5.1. 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:

$$g \in G'(u,v) \text{ iff } g \in G(u,v) \text{ and there is a weak embedding of } g^*_w \text{ at } v \text{ in the ICS, where } u = h(w). \qquad (1)$$

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 6b over the IDAG I in Figure 4, we obtain the refined ICS in Figure 6c. 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 6b, 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 (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 5.4. 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 5.1. Given an ICS on I and Q, the time and space complexities of ICS construction are $O(|E(I)||E(Q)||S(I)|/w)$ and $O(|E(I)||E(Q)|+|V(I)||V(Q)||S(I)|/w)$, respectively.

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 6 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.

5.2 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 [10, 2, 9], 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 7 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 5.1. Figure 8 shows an IDAG I to which DAGs for data graphs in D of Figure 1a are integrated, and Figure 9 illustrates the ICS on I and Q_1 of Figure 1b. There are

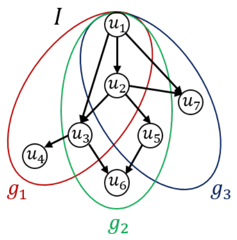


Figure 7: An illustrating example of an IDAG

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 $\{(w_1,v_1),(w_2,v_4),(w_3,v_{23})\}$ 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 5.5. 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 $\text{PFG}^*_f - A_Q$.

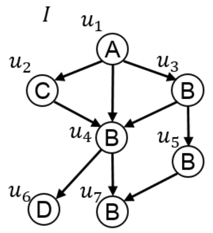
Example 5.2. Figure 10 is a search tree for IDAG I in Figure 8 and query graph Q_1 in Figure 1b. A node in the search tree corresponds to a partial feasible mapping, e.g., the root of the search tree corresponds to partial feasible mapping $f_1 = \{(u_1,v_1)\}$, its left child labeled with (u_2,v_4) 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 10, $\text{PFG}_{f_1} = \{g_1,g_2,g_3\}$ 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 5.6. Let G denote a set of data graphs. An integrated vertex u is *irrelevant* to G if G and Z_u have no common graph (i.e., $G \cap Z_u = \emptyset$); *relevant* to G otherwise.

Definition 5.7. 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 $U_{f,k} = \text{PFG}_f - \cup^k_{i=1} Z_{u_k}$ 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 5.2, 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.



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 8: The IDAG built from D of Figure 1a

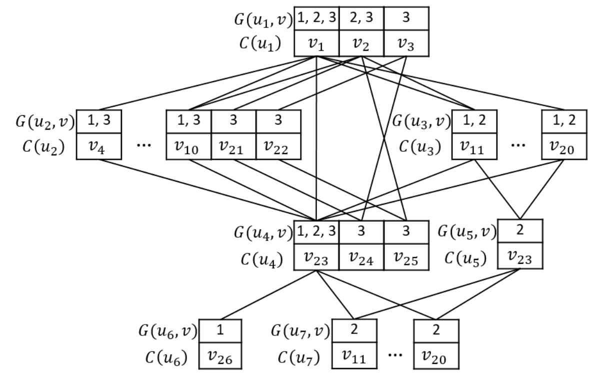


Figure 9: ICS on I in Figure 8 and Q_1 in Figure 1b

We define a failure that a partial feasible graph regarding f becomes no 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 5.8. 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 $C_f(u) = \{v \in \cup_{u_p \in \text{Parent}_f(u)} N_u^{u_p}(f(u_p)) \mid G_f(u, v) \neq \emptyset\}$ where $\text{Parent}_f(u)$ is u's parents relevant to PFG_f.

Example 5.3. Given a partial feasible mapping $f = \{(u_1,v_1), (u_2, v_4), (u_3, v_{11})\}$ in Figure 9, 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 5.2. 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 $\text{PFG}_{f'} = U_{f,k} \cap G_f(u, v)$.

In Examples 5.2 and 5.3, 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., $U_{f,0} \cap Z_{u_5} = \emptyset$.

Backtracking Framework. Based on Lemma 5.2, 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).

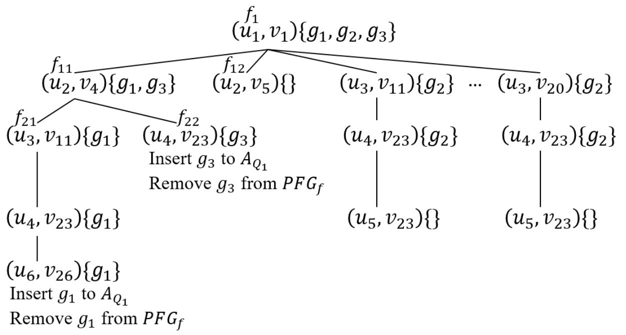


Figure 10: Search tree of backtracking for new running example of I in Figure 8 and Q_1 in Figure 1b. Each node shows the latest mapping (u, v) of f and $\text{PFG}_f = \{g_1, ..., g_k\}$ when $A_{Q_1} = \emptyset$.

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 10 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 9 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'.

5.3 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 $|\text{PFG}_{f'}|$ so that f' is shared by as many partial feasible graphs as possible in order to reduce the search space. In Figure 7, 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 $|\text{PFG}_{f'}|$ instead of $|\text{PFG}_{f'}|$ for every vertex that can be matched as follows.

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

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 [2] 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 4 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. The relevance and matchable candidates of a vertex is computed immediately when it becomes active due to an extension of f.

Algorithm 4: BACKTRACK(I, ICS, f, A_Q)

```
1  if $|f| = 0$ then
2      foreach $v \in C(r)$ do
3          $f \leftarrow \{(r, v)\}$; PFG$_f \leftarrow G(r, v)$;
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
8      return
9  else
10     $U_f \leftarrow$ PFG$_f$;
11     while $U_f \neq \emptyset$ do
12         $u \leftarrow$ next vertex based on relevance-size order;
13         if $C_f(u) = \emptyset$ then
14             $f' \leftarrow f \cup \{(u, v^*)\}$; PFG$_{f'} \leftarrow U_f - Z_u$;
15             BACKTRACK($I$, ICS, $f'$, $A_Q$);
16             $U_f \leftarrow U_f -$ PFG$_{f'}$;
17         else
18             foreach unvisited $v \in C_f(u)$ do
19                 $f' \leftarrow f \cup \{(u, v)\}$;
20                 Mark $v$ as visited;
21                 PFG$_{f'} \leftarrow U_f \cap G_f(u, v)$;
22                 Extend partial embedding of
                     $g \in$ PFG$_{f'}$ in $Q$, and if an embedding
                     of $g$ is found, insert $g$ to $A_Q$;
23                 BACKTRACK($I$, ICS, $f'$, $A_Q$);
24                 Mark $v$ as unvisited;
25                 if $Z_u \subseteq A_Q$ then break;
26             $U_f \leftarrow U_f - Z_u$;
```

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 [6] and DGTree [12] 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.

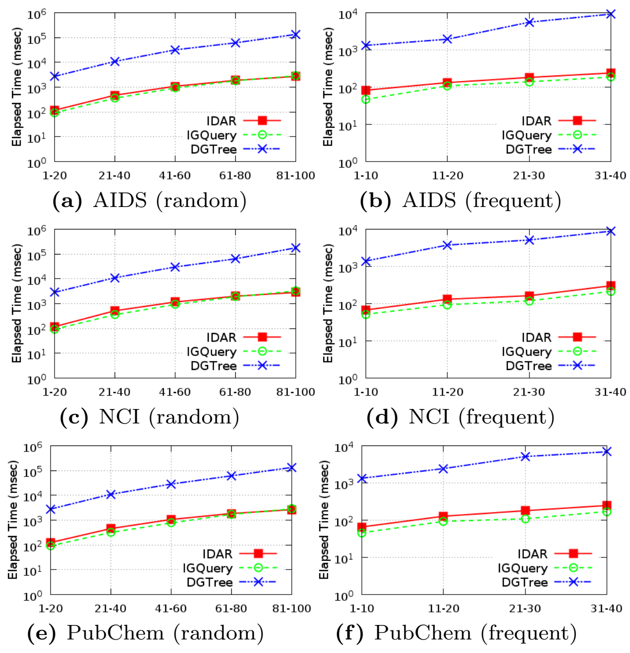
Experiments are conducted on a Windows machine with an Intel i5-7500 3.40GHz CPU and 16GB memory. The executable file of IGQuery was obtained from the authors in [6]. We couldn't get the code of DGTree from its authors. Nevertheless, we have communicated with the authors to get implementation details, and implemented the algorithm in [12]. Its performance is as good as (actually slightly better than) the one in [12] when compared against IGQuery.

Table 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[2]. NCI consists of 265,242 graphs with $1 \leq |V| \leq 342$ obtained from the National Cancer Institute database[3]. From PubChem, the open chemistry database at the NIH[4],

[2]http://dtp.nci.nih.gov/
[3]http://cactus.nci.nih.gov/download/nci/index.html
[4]https://pubchem.ncbi.nlm.nih.gov/

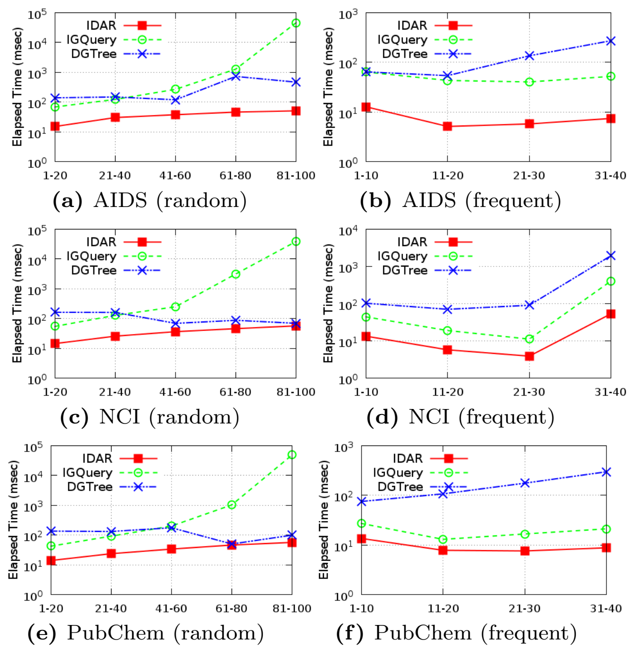


(a) AIDS (random) (b) AIDS (frequent)

(c) NCI (random) (d) NCI (frequent)

(e) PubChem (random) (f) PubChem (frequent)

Figure 11: 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 12: Query processing time for varying number of vertices in data graphs.

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 [6] and [12] 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 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 [6]. 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 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 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 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.

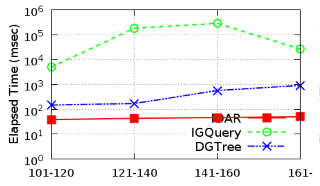
Table 3: Time complexities of the algorithms (exp = exponential, w = word size, $S(I)$ = a set of data graphs integrated to I)

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

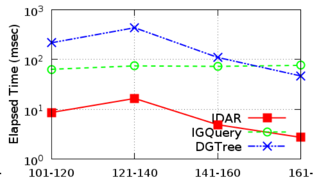
Figure 11 presents the time for index construction. IDAR and IGQuery are comparable, whereas DGTree has the worst index construction time, e.g., IDAR is faster than DGTree by at least one order of magnitude. The performance gap between DGTree and IDAR increases as the number of vertices in data graphs 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 3 are quadratic functions in the number of edges. Indeed, IDAR and IGQuery take similar time in indexing as shown in Figure 11.

In query processing performance, IDAR always outperforms the competitors as shown in Figure 12.

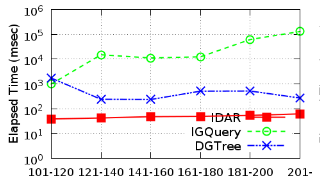
- On the one hand, IDAR is faster than IGQuery by up to two orders of magnitude (81-100 in the random data graphs of AIDS, NCI, and PubChem); furthermore, the query processing time of IDAR generally remains steady, while that of IGQuery exponentially grows. We attribute this phenomenon to the different heuristics implemented by the algorithms, i.e., IGQuery suffers from heavy cost



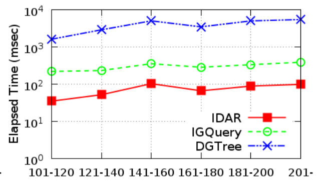
(a) AIDS (random)



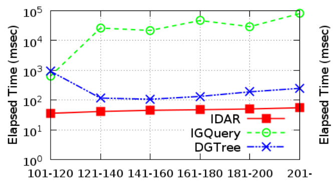
(b) AIDS (frequent)



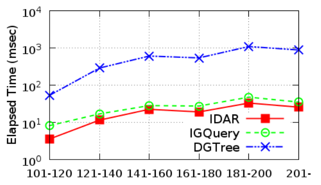
(c) NCI (random)



(d) NCI (frequent)

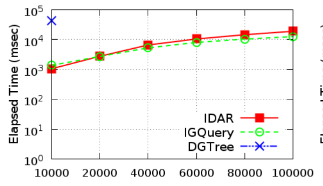


(e) PubChem (random)

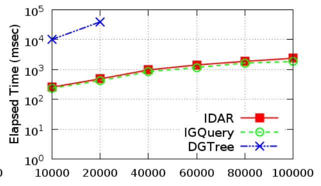


(f) PubChem (frequent)

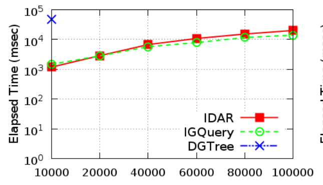
Figure 13: Query processing time for varying number of vertices in a query graph.



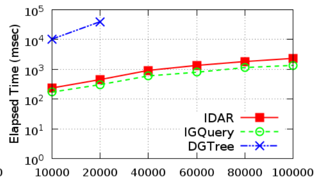
(a) AIDS (random)



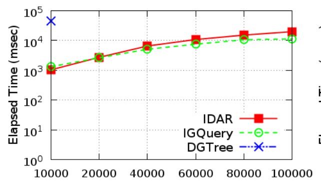
(b) AIDS (frequent)



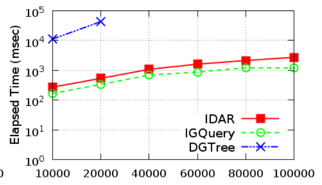
(c) NCI (random)



(d) NCI (frequent)



(e) PubChem (random)



(f) PubChem (frequent)

Figure 14: Indexing time for varying number of data graphs.

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 13: 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 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. 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.

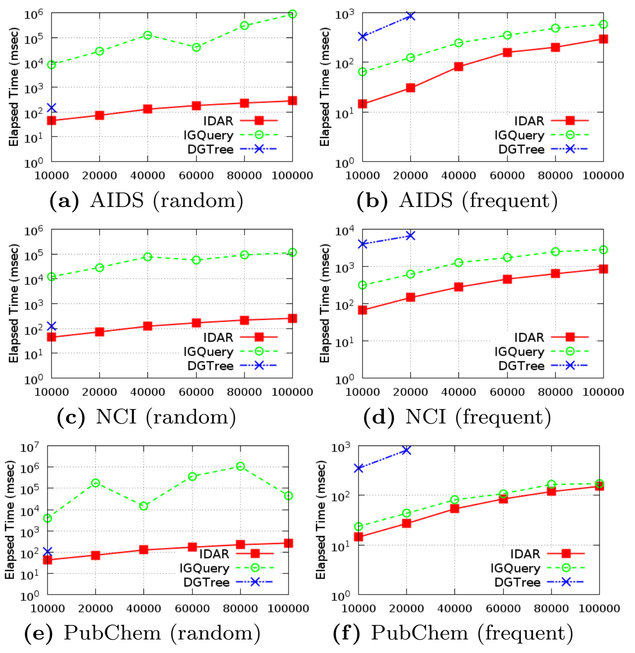
Table 4: Characteristics of data graphs and size of IG (index of IGQuery) in the experiment of Figure 13

Data graph	Avg $\lvert E(g)\rvert$	Var $\lvert E(g)\rvert$	$\lvert\Sigma\rvert$	$\lvert E(\mathsf{IG})\rvert$
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

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 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 13. According to Table 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 13.

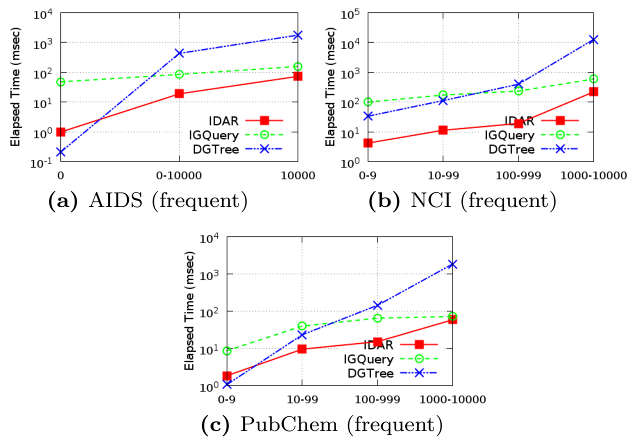
Number of Data Graphs. To test the effect of the number of data graphs on indexing and query processing, we use the sets of 10000, 20000, 40000, 60000, 80000, and 100000 data graphs.

The indexing time for each algorithm is presented in Figure 14. The indexing performance of IDAR is on par with that of IGQuery in most cases; however, DGTree doesn't 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.



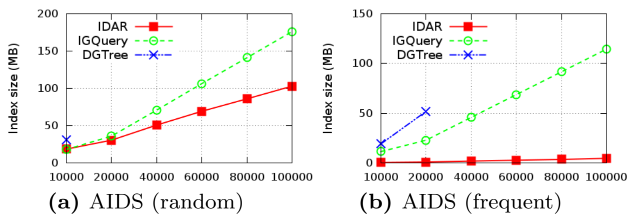
(a) AIDS (random) **(b)** AIDS (frequent) **(c)** NCI (random) **(d)** NCI (frequent) **(e)** PubChem (random) **(f)** PubChem (frequent)

Figure 15: Query processing time for varying number of data graphs.



(a) AIDS (frequent) **(b)** NCI (frequent) **(c)** PubChem (frequent)

Figure 16: Query processing time for varying number of answers.



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

Figure 17: Index size for varying number of data graphs.

Figure 15 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 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 16 shows the results. Between IGQuery and DGTree, a better performer changes as the number of answers grows. For the small number of answers, 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 17 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.

7. CONCLUSION

In this paper we have proposed a new supergraph search algorithm using DAG integration. 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. More details and related experiments will be described in the future work.

Extensive experiments on real datasets show that our approach outperforms the state-of-the-art algorithms by up to several orders of magnitude. Applying our techniques to some other graph query processing problems would be an interesting future work.

8. ACKNOWLEDGMENTS

H. Kim, S. Min, and K. Park were supported by Institute for Information communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2018-0-00551, Framework of Practical Algorithms for NP-hard Graph Problems). W.-S. Han was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIT) (No. NRF-2017R1A2B3007116).

9. REFERENCES

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

[2] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of SIGMOD*, pages 1199–1214, 2016.

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

[4] M. Cannataro and P. H. Guzzi. *Data Management of Protein Interaction Networks*, volume 17. John Wiley & Sons, 2012.

[5] C. Chen, X. Yan, P. S. Yu, J. Han, D.-Q. Zhang, and X. Gu. Towards graph containment search and indexing. In *Proceedings of VLDB*, pages 926–937, 2007.

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

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

[8] 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.

[9] 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 SIGMOD*, pages 1429–1446, 2019.

[10] 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 SIGMOD*, pages 337–348, 2013.

[11] K. Klein, N. Kriege, and P. Mutzel. Ct-index: Fingerprint-based graph indexing combining cycles and trees. In *Proceedings of IEEE ICDE*, pages 1115–1126. IEEE, 2011.

[12] B. Lyu, L. Qin, X. Lin, L. Chang, and J. X. Yu. Scalable supergraph search in large graph databases. In *Proceedings of IEEE ICDE*, pages 157–168. IEEE, 2016.

[13] A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *PVLDB*, 12(11):1692–1704, 2019.

[14] 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 (CVPR)*, pages 1–8. IEEE, 2007.

[15] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *PVLDB*, 11(4):420–431, 2017.

[16] H. Shang, K. Zhu, X. Lin, Y. Zhang, and R. Ichise. Similarity search on supergraph containment. In *Proceedings of IEEE ICDE*, pages 637–648. IEEE, 2010.

[17] S. Sun and Q. Luo. Scaling up subgraph query processing with efficient subgraph matching. In *Proceedings of IEEE ICDE*, pages 220–231. IEEE, 2019.

[18] 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.

[19] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *Proceedings of IEEE ICDE*, pages 963–972. IEEE, 2008.

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

[21] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proceedings of IEEE ICDM*, pages 721–724. IEEE, 2002.

[22] D. Yuan, P. Mitra, and C. L. Giles. Mining and indexing graphs for supergraph search. *PVLDB*, 6(10):829–840, 2013.

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

[24] 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.

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

[26] 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.