"© 2016 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works."

Scalable Supergraph Search in Large Graph Databases

Bingqing Lyu[♮], Lu Qin[‡], Xuemin Lin[§♮], Lijun Chang[§], and Jeffrey Xu Yu[♭]

[♮] East China Normal University, China

[‡]Centre for Quantum Computation & Intelligent Systems, University of Technology, Sydney, Australia

[§]The University of New South Wales, Australia

[♭]The Chinese University of Hong Kong, China

[♮]lvbingqings@gmail.com; [‡]lu.qin@uts.edu.au; [§]{lxue,ljchang}@cse.unsw.edu.au; [♭]yu@se.cuhk.edu.hk

***Abstract*—Supergraph search is a fundamental problem in graph databases that is widely applied in many application scenarios. Given a graph database and a query-graph, supergraph search retrieves all data-graphs contained in the query-graph from the graph database. Most existing solutions for supergraph search follow the pruning-and-verification framework, which prunes false answers based on features in the pruning phase and performs subgraph isomorphism testings on the remaining graphs in the verification phase. However, they are not scalable to handle large-sized data-graphs and query-graphs due to three drawbacks. First, they rely on a frequent subgraph mining algorithm to select features which is expensive and cannot generate large features. Second, they require a costly verification phase. Third, they process features in a fixed order without considering their relationship to the query-graph. In this paper, we address the three drawbacks and propose new indexing and query processing algorithms. In indexing, we select features directly from the data-graphs without expensive frequent subgraph mining. The features form a feature-tree that contains all-sized features and both the cost sharing and pruning power of the features are considered. In query processing, we propose a verification-free algorithm, where the order to process features is query-dependent by considering both the cost sharing and the pruning power. We explore two optimization strategies to further improve the algorithm efficiency. The first strategy applies a lightweight graph compression technique and the second strategy optimizes the inclusion of answers. Finally, we conduct extensive performance studies on two real large datasets to demonstrate the high scalability of our algorithms.**

I. INTRODUCTION

Graphs are widely used in numerous application domains, such as biology, chemistry, image processing, ecology, electrical circuits, computer vision, etc. A fundamental problem in graph databases is graph containment search, which consists of subgraph search and supergraph search. Given a query-graph Q, subgraph search [8, 13, 16, 23, 25, 30] finds all data-graphs that contain Q in the graph database, while supergraph search [6, 7, 24, 29, 31] retrieves all data-graphs that are contained in Q in the graph database. In this paper, we focus on the supergraph search problem.

Applications. Supergraph search is used in many application scenarios. For example:

(1) Biology. In biology, protein-protein interaction (PPI) networks play an important role in understanding the cell as a system of interacting components and finding how proteins in the cell interact with each other [5]. Containment analysis of PPI networks across species will facilitate researchers to discover some specific relationships between species.

(2) Chemistry. In chemistry, when a new molecule is discovered, researchers can predict its possible properties by comparing its topological structure with already known chemical compounds [32]. Supergraph search can be used in this application that if the new molecule contains some structures of existing molecules, it is very likely that the new molecule has similar properties to these existing molecules.

(3) Computer Vision. In image processing and computer vision, a segmented object is represented by a region adjacency graph that can be transformed into a spatial entity [18]. In recognition systems, such classical models allow users to identify specific objects in images with supergraph search techniques by taking the images as queries.

More applications can be found in [6, 7, 24, 29, 31].

Existing Solutions. Supergraph search usually involves subgraph isomorphism testing which is NP-complete [10]. Existing solutions for supergraph search are based on two logics, inclusion logic and exclusion logic, to reduce the number of subgraph isomorphism testings. Inclusion logic is used in [7], which first finds a set of data-graphs that are guaranteed to be answers and then verifies other data-graphs using subgraph isomorphism testing. Considering that most data-graphs are not answers in practice, the cost saving based on inclusion logic is usually limited. Therefore, all other works in the literature [6, 24, 29, 31] follow the exclusion logic, which first prunes false answers and then verifies other data-graphs using subgraph isomorphism testing.

Exclusion logic is first introduced in [6], which computes a set of features selected from the frequent subgraphs. Given a query-graph Q, if the feature is not contained in Q, all data-graphs containing the feature can be pruned. How to mine better features using query logs for supergraph search is further studied in [24]. In [29], an algorithm is proposed to improve [6], that divides the features and data-graphs into partitions, and considers cost sharing in each partition in the pruning phase and verification phase. The algorithm is further improved in [31] by considering the cost sharing between partitions and the cost sharing between pruning and verification phases. More details on existing solutions are outlined in Section III.

Motivation. Existing solutions based on exclusion logic face a scalability problem when handling large-sized data-graphs and

Lu Qin and Xuemin Lin are the corresponding authors.

query-graphs for three reasons: (1) The features selected in existing solutions largely rely on a frequent subgraph mining algorithm (e.g., gSpan [22]). However, such algorithms are usually expensive and can only find small-sized subgraphs which are likely to be included in a large-sized query-graph, and thus have low pruning power. (2) A verification phase is required that may involve a large number of subgraph isomorphism testings. (3) Features are processed in a fixed order without considering their pruning power regarding to the query-graph. As a result, when processing a feature with low pruning power, the algorithm may be trapped in an exponential search space without pruning any data-graphs.

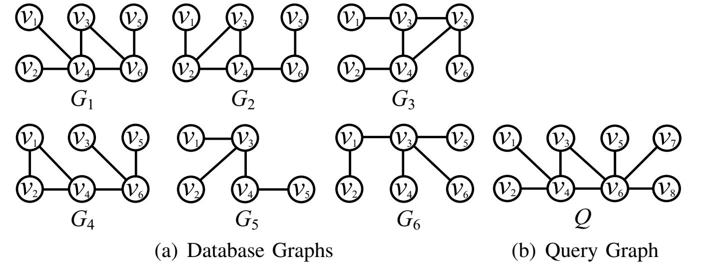
Contributions. In this paper, we follow exclusion logic and propose algorithms to address the above drawbacks of existing solutions. We make the following contributions:

(1) A full-structure index without frequent subgraph mining. In this paper, we propose an indexing algorithm that selects features directly from the data-graphs without relying on a frequent subgraph mining algorithm. We construct a feature-tree to tackle cost sharing among features. The feature-tree is constructed by iteratively growing edges on existing features even if the newly generated features are infrequent, and the leaf nodes of the feature-tree are exactly the set of data-graphs in the graph database. In other words, we construct a full-structure index that tries to consider possible structure sharing among the features. The edge-growing process is directed by a score function that considers both pruning power and cost sharing of each newly generated feature. The number of features in the feature-tree can be well bounded, and we can guarantee that no two features in the feature-tree are isomorphic without graph isomorphism checking.

(2) A verification-free query processing algorithm with dynamic feature ordering. Based on the full-structure feature-tree we constructed, we propose a new algorithm for query processing. Our algorithm does not require a verification phase and it handles both small and large features in a progressive manner. We also consider dynamic feature ordering to process features according to their scores. The score of the feature is designed based on consideration of both its cost sharing and pruning power, and when data-graphs are included or excluded from the answers, our algorithm updates the scores of the features adaptively. In this way, we avoid processing useless features and thus significantly reduce computational cost.

(3) Two optimization strategies to further improve the algorithm efficiency. We explore two novel optimization strategies to further improve the efficiency of our algorithm. The first strategy considers the redundant computational cost when processing symmetric nodes in data-graphs. We propose a lightweight graph compression technique to contract symmetric nodes in data-graphs and thus save computational cost. In the second optimization strategy, we propose an inclusion-aware query processing algorithm, that tries to reduce computational cost when processing a feature that is likely to include data-graphs in the answers. Instead of expanding all matches of the feature in the query-graph for pruning, we partition the matches so that the matches which are more likely to include answers have a higher priority to be expanded. Both optimization strategies can be easily embedded in our algorithm framework.

(4) Extensive performance studies on large real datasets. We conduct extensive performance studies by using two large real datasets CCD dataset and NCI dataset. The experimental results demonstrate that our indexing and query processing algorithms are scalable to handle large-sized data-graphs and query-graphs. According to our experimental results, our query processing algorithm is eight times faster than the state-of-the-art algorithms on average.



(a) Database Graphs (b) Query Graph

Fig. 1: An Example of Supergraph Search Problem

II. Problem Definition

A graph G is represented as a tuple $G = (V, E)$, where $V(G)$ is the set of nodes and $E(G) \subseteq V \times V$ is the set of edges in G. We denote the number of nodes and edges in G by $|V(G)|$ and $|E(G)|$ respectively. For simplicity, we only focus on *undirected, unlabelled, connected,* and *simple* graphs in this paper. Here, a simple graph is a graph with no self-loops and no parallel edges. However, our algorithms can also be extended to handle other types of graphs. For each node $u \in V(G)$, we use $\mathsf{Nbr}(u, G)$ to denote the set of neighbors of u in G, i.e., $\mathsf{Nbr}(u, G) = \{v|(u, v) \in E(G)\}$. The degree of a node $u \in V(G)$, denoted by $d(u, G)$, is the number of neighbors of u in G, i.e., $d(u, G) = |\mathsf{Nbr}(u, G)|$.

Definition 2.1: (Subgraph Isomorphism) Given two graphs G_1 and G_2, G_1 is *subgraph isomorphic* to G_2, denoted as $G_1 \subseteq G_2$, iff there is an injective function $f : V(G_1) \rightarrow V(G_2)$, such that $\forall(u, v) \in E(G_1)$, $(f(u), f(v)) \in E(G_2)$.

If G_1 is subgraph isomorphic to G_2, G_1 is called a *subgraph* of G_2, and G_2 is called a *supergraph* of G_1. If $G_1 \subseteq G_2$, we call that G_2 *contains* G_1. We use $G_1 \nsubseteq G_2$ to denote that G_1 is not subgraph isomorphic to G_2.

G_1 is *isomorphic* to G_2, denoted as $G_1 = G_2$, iff $G_1 \subseteq G_2$ and $G_2 \subseteq G_1$. We use $G_1 \neq G_2$ to denote that G_1 is not isomorphic to G_2. □

Problem Statement. Given a graph database $\mathcal{D} = \{G_1, G_2, \ldots, G_n\}$ and a graph Q, where each $G_i \in \mathcal{D}$ is called a *data-graph*, and Q is called a *query-graph*, the problem of *supergraph search* is to find all graphs in $\mathcal{D}$, that are subgraph isomorphic to Q, i.e., supergraph search aims to compute the answer set $\mathcal{A}(Q) = \{G_i|G_i \in \mathcal{D}, G_i \subseteq Q\}$. For simplicity, in this paper, we assume that no two graphs in $\mathcal{D}$ are isomorphic.

Note that the problem of supergraph search is NP-hard. This is because, given two graphs G and Q, the problem of deciding whether G is subgraph isomorphic to Q is an NP-complete problem [10].

Example 2.1: Given a graph database $\mathcal{D} = \{G_1, G_2, ..., G_6\}$ in Fig. 1(a) and a query-graph Q in Fig. 1(b), supergraph search finds the answer set $\mathcal{A}(Q) = \{G_1, G_5, G_6\}$, each of which is subgraph isomorphic to query Q. □

III. EXISTING SOLUTIONS

In the literature, existing solutions for supergraph search mainly fall into two categories, namely, feature based approach and integrated-graph based approach.

A. Feature Based Approach

Algorithm CIndex. CIndex [6] is the first feature based approach for supergraph search, that adopts the *pruning-and-verification* framework. Given a graph database $\mathcal{D} = \{G_1, G_2, \ldots, G_n\}$, a feature-based index is built on $\mathcal{D}$ for query processing. The index consists of a set of features $\mathcal{F} = \{f_1, f_2, \ldots, f_k\}$, that are selected from the (closed) frequent subgraphs on $\mathcal{D}$ mined by a (closed) frequent subgraph mining algorithm, e.g., gSpan [22]. The feature selection is based on historical queries in the query logs. For each feature $f_i \in \mathcal{F}$, a set of data-graphs $S(f_i) = \{G_j | G_j \in \mathcal{D}, f_i \subseteq G_j\}$ is also precomputed. In query processing, given a query-graph Q, the answer to Q is computed in two phases, pruning and verification. In the pruning phase, the algorithm applies an *exclusion logic* to prune data-graphs, that is, if a feature $f_i \nsubseteq Q$, any data-graph $G_j \in S(f_i)$ can be pruned. In the verification phase, the data-graphs that are not pruned by any features will be verified against the query-graph Q using subgraph isomorphism testing. How to mine better features from the query logs and maintain such features is further studied in [24]. Note that, we do not consider the optimization techniques based on query logs in this paper.

Algorithm GPTree. To improve CIndex by (1) reducing redundant computational cost to process common subgraphs of features and/or data-graphs, and (2) avoiding being largely dependent on query logs, in [29], a feature based approach GPTree is proposed under the same pruning-and-verification framework. In GPTree, after selecting a set of features from the frequent subgraphs, both the set of features and the set of data-graphs are partitioned into groups. Graphs in each feature group or data-graph group share a certain subgraph, which is denoted as the prefix of the group. The partition is performed in a way that large-sized prefixes are preferred. In query processing, for both the pruning and verification phases, the cost of subgraph isomorphism testing on the prefix part can be shared among the graphs in the same group, which reduces redundant computational cost.

Algorithm PrefIndex. To further reduce redundant computational cost, in [31], a feature based algorithm PrefIndex is proposed for supergraph search. PrefIndex improves GPTree in three ways: (1) By assuming each feature to be a prefix of a certain data-graph group, PrefIndex handles the computational cost sharing between the pruning and verification phases; (2) By considering multi-level sharing among features, PrefIndex handles the computational cost sharing among data-graph groups; and (3) In the feature selection process, instead of preferring large-sized features, PrefIndex considers both the size of the feature and the number of data-graphs containing each feature, which can further enlarge the cost sharing compared to GPTree in query processing.

B. Integrated-graph Based Approach

Algorithm IGQuery. Recall that feature based approaches apply an exclusion logic to prune false answers followed by verification. The integrated-graph based approach IGQuery, proposed by Cheng et al. [7], aims to answer supergraph search queries based on *inclusion logic*. That is, given a graph database $\mathcal{D}$ and a query Q, instead of pruning false answers, IGQuery tries to find a set of data-graphs $\mathcal{A}^*$ that each graph in $\mathcal{A}^*$ is guaranteed to be a subgraph of Q. After computing $\mathcal{A}^*$, IGQuery further applies a simple feature based algorithm to prune false answers. Finally, the remaining data-graphs are verified against Q one by one to arrive at the final answer. To compute $\mathcal{A}^*$, IGQuery precomputes an index by integrating all the data-graphs in $\mathcal{D}$ into a single graph, namely, the *integrated graph IG*. Each graph in $\mathcal{D}$ has a unique embedding in IG. For each edge e in IG, the set of data-graphs containing e in their embeddings is precomputed and denoted as $host(e)$. In query processing, given a query Q, the algorithm first computes an embedding of Q in IG. If there is a such an embedding, by intersecting $host(e)$ for all edges e in the embedding, the set $\mathcal{A}^*$ can be obtained.

IV. A NEW VERIFICATION-FREE APPROACH

A. Problem Analysis

Drawbacks of Existing Solutions. Note that in the integrated-graph based approach, the integrated-graph is used to compute a subset of answers based on the inclusion logic. However, only small-sized data-graphs tend to be found by inclusion logic. Moreover, in a supergraph search, the number of answers is usually much smaller than the number of data-graphs. Therefore, the cost saving using integrated-graph is usually limited. For example, in our experiment, using the database consisting of 3000 graphs with an average size of 50 obtained from the CCD dataset, given a query of size 100, the number of answers is only 13% of the number of data-graphs.

Due to the above inherent limitations of inclusion logic used in integrated-graph based approach, in this paper, we aim to improve the feature based approach using exclusion logic to process supergraph search. The existing feature based approaches have the following limitations, which make them unscalable when handling large-sized data-graphs.

(L_1) Expensive frequent subgraph mining algorithms are used in indexing. In existing feature based solutions, the features are selected from the set of frequent subgraphs mined by an existing frequent subgraph mining algorithm (e.g., gSpan [22]) or its variants. However, such algorithms are usually expensive and not scalable to handle large-sized data-graphs. For example, in our experiment, for the database consisting of 3000 graphs with an average size of 90 obtained from the CCD dataset, by setting the minimum threshold to be 0.2, over 14 hours are spent mining the frequent subgraphs using gSpan.

(L_2) Verification may be expensive on large-sized data-graphs. In existing feature based approaches, although the cost sharing when processing common features can be considered in the verification phase, cost savings are limited when the size of data-graphs is large, since the features are selected from frequent subgraphs. This makes the verification costly.

(L_3) Features are processed in a query-independent order in query processing. Existing feature based approaches mainly focus on how to select a good set of features. In query processing, the order to process the features is fixed and is independent of the query-graph. However, it is obvious that a feature may have different pruning power for different query-graphs. Therefore, given a query-graph, if we can estimate

the pruning power of each feature w.r.t. the query-graph, processing the features in decreasing order of their pruning power may significantly enhance the search efficiency.

Our Approach. The limitations of existing solutions motivate us to find better indexing and query processing algorithms. In this paper, we propose a new approach that addresses the above limitations in the following ways:

(A_1) Direct feature selection without frequent subgraph mining. In our approach, features are selected directly from the data-graphs to avoid generating an exponential number of subgraphs followed by selecting features from them. To generate the features, we construct a feature-tree by incrementally growing edges on each feature. We consider both the cost sharing and pruning power for each generated feature. Compared to frequent subgraph mining, our feature selection process has two advantages: (1) The total number of features generated is bounded; and (2) For each newly generated feature, we do not need to perform isomorphism checking against existing features, since we can guarantee that no two features generated are isomorphic to each other. The two advantages make our feature selection process much more efficient than frequent subgraph mining.

(A_2) Verification-free query processing based on a full-structure index. In our approach, we allow both small and large features in our feature generation process. A small feature usually has limited pruning power but it may be shared by a large number of data-graphs, while a large feature may have high pruning power but it is usually shared by a small number of data-graphs. In our algorithm, the edge-growing process terminates until the new generated feature becomes a data-graph. In other words, we consider a full-structure index that tries to consider possible structure sharing among features, even if they are not frequent. All data-graphs are preserved in the index. Based on the full-structure index, our query processing algorithm is verification-free.

(A_3) Query-dependent feature processing during supergraph search. Since the pruning power of each feature is query-dependent, during the supergraph search of our approach, we define a dynamic score function for each feature in the feature-tree. The score function considers both the pruning power and cost sharing of the feature and is adaptively adjusted during the search process when data-graphs are included or excluded from the answers. The features are processed in a top-down manner in the feature-tree and once a feature is not contained in the query-graph, all its descendants are discarded.

B. DGTree*: A Full-structure Index*

In this subsection, we introduce our indexing technique. We first define a match of a graph as follows:

Definition 4.1: (Match) Given a graph P with nodes $\{u_1, u_2, \ldots, u_{|V(P)|}\}$ and a data-graph G, a *match* f of P in G is a mapping from $V(P)$ to $V(G)$ such that the following two conditions hold:

- (Conflict-free): For any pair of nodes $u_i \in V(P)$ and $u_j \in V(P)$ $(u_i \neq u_j)$, $f(u_i) \neq f(u_j)$.
- (Structure-preserved): For any edge $(u_i, u_j) \in E(P)$, $(f(u_i), f(u_j)) \in E(G)$.

We use $f = [v_1, v_2, \ldots, v_{|V(P)|}]$ to denote the match f, i.e., $f(u_i) = v_i$ for any $1 \leq i \leq |V(P)|$. If $f(u_i) = v_i$, we have $f^{-1}(v_i) = u_i$. □

Algorithm 1: DGTreeConstruct(database $\mathcal{D} = \{G_1, \ldots, G_n\}$)

```
1  g_r ← a new tree-node;
2  g_r.graph ← a single-edge graph;
3  g_r.S ← D; g_r.S* ← D; g_r.grow-edge = ∅;
4  for G_i ∈ D and (v, v') ∈ E(G_i) do
5  └ g_r.M(G_i) ← g_r.M(G_i) ∪ {[v, v'], [v', v]};
6  TreeGrow(g_r);
7  return g_r;
8  Procedure TreeGrow(tree-node g)
9  H ← CandidateFeature(g);
10 C ← g.S*;
11 while C ≠ ∅ do
12 │  g+ ← BestFeature(H, C);
13 │  if |g+.S*| > 1 then
14 │  │  g+.graph ← a graph by adding g.grow-edge in g.graph;
15 │  └  TreeGrow(g+);
16 │  else  g+.graph ← the graph in g.S*; g+.S ← g+.S*;
17 │  g.children ← g.children ∪ {g+};
18 └  C ← C \ g+.S*;
```

As mentioned above, our index is a tree structure which can preserve all data-graphs with substructure sharing. We call the index a DGTree, which has the following structure:

DGTree **Structure**. The DGTree contains a set of tree-nodes. Each tree-node g consists of the following components:

g.children	The set of child tree-nodes of g. A leaf tree-node is a tree-node g with $\lvert g.\text{children}\rvert = 0$.
g.graph	The feature-graph of g. The set of nodes in the feature-graph is represented by $\{1, 2, \ldots, \lvert V(g.\text{graph})\rvert\}$.
g.grow-edge	The edge added to g.graph from the feature-graph of the parent tree-node of g.
g.edge-type	The type of g.grow-edge, which is CLOSE if no new node is created in g.graph and OPEN if a new node is created in g.graph after adding g.grow-edge.
$g.\mathcal{S}$	The set of data-graphs containing g.graph, i.e., $g.\mathcal{S} = \{G \mid G \in \mathcal{D}, g.\text{graph} \subseteq G\}$. If g is a leaf tree-node, $g.\mathcal{S}$ contains the data-graph equaling to g.graph, and thus the union of $g.\mathcal{S}$ for all leaf tree-nodes g is $\mathcal{D}$.
$g.\mathcal{M}(G_i)$	The set of matches of g.graph in G_i for each $G_i \in g.\mathcal{S}$.
$g.\mathcal{S}^*$	$g.\mathcal{S}^* \subseteq g.\mathcal{S}$. If g is the root tree-node, we have $g.\mathcal{S}^* = \mathcal{D}$. If g is a leaf tree-node, we have $\lvert g.\mathcal{S}^*\rvert = 1$. For a non-leaf tree-node g, we have: (1) for any $g_i \in g$.children and $g_j \in g$.children $(g_i \neq g_j)$, $g_i.\mathcal{S}^* \cap g_j.\mathcal{S}^* = \emptyset$; and (2) $\bigcup_{g_i \in g.\text{children}} g_i.\mathcal{S}^* = g.\mathcal{S}^*$. In other words, $g_i.\mathcal{S}^*$ for all $g_i \in g$.children form a disjoint cover of $g.\mathcal{S}^*$.
g.score	The score of g.graph, which is used to select the best edge to grow.

Note that the last three components $g.\mathcal{M}(G_i)$, $g.\mathcal{S}^*$, and g.score are only used in DGTree construction and can be discarded from the DGTree after it is constructed. In the following, we use a tree-node and a feature-graph interchangeably if the context is obvious.

DGTree **Construction**. Given a graph database $\mathcal{D} = \{G_1, G_2, \ldots, G_n\}$, the algorithm to construct the DGTree is shown in Algorithm 1. Generally speaking, the algorithm divides the data-graphs in $\mathcal{D}$ into partitions. Each partition (represented by a tree-node g) consists of the set of data-graphs $g.\mathcal{S}^*$ containing g.graph. A partition can be further divided recursively. During the recursive partitioning of $\mathcal{D}$, the DGTree is constructed. Specifically, the algorithm first creates the root tree-node g_r with a feature-graph of a single edge (line 1-2).

Algorithm 2: CandidateFeature(tree-node g)

```
1  H ← ∅;
2  for data-graph G ∈ g.S* and match f ∈ g.M(G) do
3      for u_i ← 1 to |f| and v ∈ Nbr(f(u_i), G) do
4          if v ∈ f then  u_j ← f^{-1}(v); t ← CLOSE;
5          else  u_j ← |f| + 1; t ← OPEN;
6          if u_j > u_i and (u_i, u_j) ∉ E(g) then
7              g+ ← H.Find((u_i, u_j));
8              if g+ = ∅ then
9                  g+ ← a new tree-node;
10                 g+.grow-edge ← (u_i, u_j);
11                 g+.S* ← {G}; g+.score ← 0;
12                 g+.edge-type ← t;
13                 H.Push(g+);
14             else  g+.S* ← g+.S* ∪ {G};
15 for data-graph G ∈ g.S and match f ∈ g.M(G) do
16     for u_i ← 1 to |f| and v ∈ Nbr(f(u_i), G) do
17         if v ∈ f then  u_j ← f^{-1}(v);
18         else  u_j ← |f| + 1;
19         if u_j > u_i and (u_i, u_j) ∉ E(g) then
20             g+ ← H.Find((u_i, u_j));
21             if g+ ≠ ∅ then
22                 g+.S ← g+.S ∪ {G};
23                 if g+.edge-type = OPEN then
24                     g+.M(G) ← g+.M(G) ∪ {[f, v]};
25                 else  g+.M(G) ← g+.M(G) ∪ {f};
26 for g+ ∈ H do
27     compute g+.score; H.Update(g+);
28 return H;
```

Other components of g_r are initialized accordingly (line 3-5). After creating g_r, the algorithm constructs the DGTree in a top-down manner starting from g_r using TreeGrow(g_r) and returns g_r as the root entry of the DGTree.

The tree-growing process TreeGrow(g) is shown in line 8-18 of Algorithm 1. We first generate a set of candidate grow feature-graphs and put them into a heap $\mathcal{H}$ (line 9). We will introduce how to compute $\mathcal{H}$ using CandidateFeature(g) later. Each entry $g^+ \in \mathcal{H}$ represents a candidate feature-graph by growing a certain edge on the feature-graph of g. *Here, we use either a feature-graph or an edge to represent an entry in $\mathcal{H}$ since each feature-graph in $\mathcal{H}$ can be uniquely identified by a growing edge.* In line 10, we use $\mathcal{C}$ to maintain the set of uncovered data-graphs in $g.\mathcal{S}^*$. In line 11-18, we iteratively generate the child tree-nodes of g until all data-graphs in $g.\mathcal{S}^*$ have been covered.

In each iteration (line 12-18), we generate a new feature-graph by adding an edge in g.graph. We find the candidate g^+ with the highest g^+.score in $\mathcal{H}$ by invoking BestFeature($\mathcal{H}, \mathcal{C}$), which will be introduced later. After obtaining g^+, we consider two cases. First, when $|g^+.\mathcal{S}^*| > 1$ (line 13), we create the feature-graph of g^+ by adding the edge g^+.grow-edge on the feature-graph of g (line 14), and we recursively invoke TreeGrow(g^+) to generate the subtree of g^+. Second, when $|g^+.\mathcal{S}^*| = 1$, g^+ is a leaf tree-node with the feature-graph being the data-graph in $g^+.\mathcal{S}^*$, and we set $g^+.\mathcal{S}$ as $g^+.\mathcal{S}^*$ (line 16). Here, if $g^+.\mathcal{S}^*$ contains a data-graph equaling to g^+.graph in the first case, we create a leaf node for this data-graph under g^+ as in line 16. Finally, we add g^+ to be a child tree-node of g (line 17) and remove the covered data-graphs $g^+.\mathcal{S}^*$ from $\mathcal{C}$ (line 18).

• **Procedure** CandidateFeature. The procedure to compute the candidate feature-graphs grown from the feature-graph of g is shown in Algorithm 2. The procedure consists of two phases. In phase 1 (line 2-14), we compute all the candidates $g^+ \in \mathcal{H}$ as well as the set $g^+.\mathcal{S}^*$, which contains the data-graphs that g^+ covers. In phase 2 (line 15-25), we compute $g^+.\mathcal{S}$ and $g^+.\mathcal{M}(G)$ for each $G \in g.\mathcal{S}$.

Algorithm 3: BestFeature(heap $\mathcal{H}$, uncovered graphs $\mathcal{C}$)

```
1 g+ ← H.Pop();
2 while g+.S* ⊈ C do
3     g+.S* ← g+.S* ∩ C;
4     if g+.S* ≠ ∅ then  {compute g+.score; H.Push(g+)};
5     g+ ← H.Pop();
6 return g+;
```

In phase 1 (line 2-14), we enumerate all matches $f \in g.\mathcal{M}(G)$ for each data-graph $G \in g.\mathcal{S}^*$ to find all possible edges that can grow on the feature-graph of g (line 2). For each match f and each node u_i in the feature-graph of g, we traverse all the neighbors v of $f(u_i)$ in the data-graph G (line 3). If v has been matched to node u_j in f, the edge (u_i, u_j) indicates a new CLOSE edge to grow (line 4); Otherwise, we set u_j to be $|f| + 1$, indicating that (u_i, u_j) is a new OPEN edge to grow (line 5). In line 6, we ensure $u_j > u_i$ to guarantee that each undirected edge is uniquely represented, and we also ensure that (u_i, u_j) does not exist in $E(g)$. If the candidate g^+ corresponding to edge (u_i, u_j) does not exist in $\mathcal{H}$ (line 8), we create a new candidate g^+ (line 9), set the corresponding components (line 10-12), and push it into $\mathcal{H}$ (line 13); Otherwise, we add G into $g^+.\mathcal{S}^*$, indicating that after adding edge (u_i, u_j) in the feature-graph of g, G still covers the new feature-graph.

In phase 2 (line 15-25), we follow a similar procedure to enumerate all possible edges (u_i, u_j) to be added by enumerating matches f of graphs G in the set $g.\mathcal{S}$ rather than $g.\mathcal{S}^*$ (line 15-20). For each such (u_i, u_j), if the corresponding candidate g^+ exists in $\mathcal{H}$ (line 21), we add G into $g^+.\mathcal{S}$ (line 22) and add a new match in $g^+.\mathcal{M}(G)$. Here, if (u_i, u_j) is an OPEN edge (line 23), the new match to be added is $[f, v]$ (line 24) where v is the newly matched node of u_j in G; Otherwise, the new match to be added is f (line 25). Note that in our implementation, to save computational cost, we do not actually compute $g^+.\mathcal{M}(G)$ for all $g^+ \in \mathcal{H}$. Instead, we only count $|g^+.\mathcal{M}(G)|$ to be used in feature score computation. However, we still need to compute $g^+.\mathcal{M}(G)$ for each selected candidate feature-graph before recursively invoking TreeGrow(g^+), i.e. after line 14 in Algorithm 1. In addition, if $G \notin g^+.\mathcal{S}^*$, we control the size of $g^+.\mathcal{M}(G)$ for efficiency consideration, which will not affect the correctness of query processing.

Finally, after computing all components of entries g^+ in $\mathcal{H}$, we compute g^+.score and update g^+ in $\mathcal{H}$ using the new score. We will introduce how to compute g^+.score later.

• **Procedure** BestFeature. The procedure to get the candidate feature-graph with the highest score is shown in Algorithm 3. Note that when a new feature-graph is generated by growing an edge on the feature-graph of g, $g^+.\mathcal{S}^*$ for some other entries g^+ in $\mathcal{H}$ may change by removing those covered data-graphs, which will decrease g^+.score. In our algorithm, instead of maintaining $g^+.\mathcal{S}^*$ for all g^+ in $\mathcal{H}$ each time a new feature-graph is generated, we compute $g^+.\mathcal{S}^*$ in a lazy manner after it is popped from $\mathcal{H}$. In order to do so, after we pop a new

candidate g^+ from $\mathcal{H}$ (line 1 and line 5), we check whether $g^+.\mathcal{S}^*$ is up-to-date, i.e., $g^+.\mathcal{S}^* \subseteq \mathcal{C}$, if so, we return g^+ (line 6) as the best candidate. Otherwise, we update $g^+.\mathcal{S}^*$ by removing those covered data-graphs (line 3). If $g^+.\mathcal{S}^*$ is not $\emptyset$, we recompute g^+.score and push it back into $\mathcal{H}$ (line 4). The process stops when $g^+.\mathcal{S}^*$ is up-to-date.

Feature Score Computation. We discuss how to compute g.score for each candidate g. Intuitively, to share computational cost in query processing, g should be selected to maximize the number of data-graphs containing g.graph, i.e., to maximize $|g.\mathcal{S}|$. However, by using $|g.\mathcal{S}|$ as the score function, we may generate redundant feature-graphs, e.g., two similar feature-graphs that cover almost the same set of data-graphs. Therefore, we consider to maximize $|g.\mathcal{S}^*|$, since $g.\mathcal{S}^*$ is the set of data-graphs covered by g after excluding the already covered data-graphs. Our first score function is defined as:

$$g.\mathsf{score}_1 = |g.\mathcal{S}^*| \quad (1)$$

Eq. 1 considers both the cost sharing and the feature diversity when selecting candidate feature-graphs. However, it does not consider the pruning power of the feature-graph. Intuitively, if the average number of matches for a feature-graph in the data-graphs is small, it is not likely to be contained in a query-graph, and thus the pruning power of such a feature-graph is high. Here the average number of matches for a feature-graph is $\sum_{G \in g.\mathcal{S}} |g.\mathcal{M}(G)|/|g.\mathcal{S}|$. Combined with Eq. 1, our second score function is defined as:

$$g.\mathsf{score}_2 = \frac{|g.\mathcal{S}^*| \times |g.\mathcal{S}|}{\sum_{G \in g.\mathcal{S}} |g.\mathcal{M}(G)|} \quad (2)$$

Finally, note that in query processing, given a query Q, we may need to compute the matches of the new feature-graph in Q. For a candidate g, if g.edge-type is OPEN, the number of matches of g.graph in Q may increase exponentially; otherwise if g.edge-type is CLOSE, the number of matches of g.graph in Q will always decrease. Therefore, adding a CLOSE edge to the feature-graph will make the feature-graph more selective without introducing much extra cost in online query processing. In addition, a CLOSE edge will not increase the number of nodes in a feature-graph, and thus can better bound the size of the feature-graph. Based on such properties, when selecting edges to grow, we give higher priority to CLOSE edges than OPEN edges. Let MAX be a large constant, e.g., $\mathsf{MAX} = |\mathcal{D}|$, our final score function is defined as:

$$g.\mathsf{score} = \begin{cases} g.\mathsf{score}_2 & \text{if } g.\mathsf{edge\text{-}type} = \mathsf{OPEN} \\ g.\mathsf{score}_2 + \mathsf{MAX} & \text{if } g.\mathsf{edge\text{-}type} = \mathsf{CLOSE} \end{cases} \quad (3)$$

Algorithm Analysis. Compared to the frequent subgraph mining algorithm (e.g., gSpan [22]) used in the indexing of existing approaches for supergraph search, our indexing method based on DGTree has the following advantages:

• **The number of feature-graphs is bounded.** While existing frequent subgraph mining algorithm may generate exponential number of subgraphs, the feature-graphs generated in our algorithm can be bounded based on the following lemma:

Lemma 4.1: *Given a graph database $\mathcal{D}$, the number of feature-graphs in the* DGTree *is no larger than* $\sum_{G \in \mathcal{D}} |E(G)|$. □

Proof: Note that each leaf tree-node of the DGTree represents a certain data-graph $G \in \mathcal{D}$ with depth no larger than $|E(G)|$, and no two leaf tree-nodes represent the same data-graph. It is easy to prove that the total number of feature-graphs in the DGTree is no larger than $\sum_{G \in \mathcal{D}} |E(G)|$. □

• **No isomorphism checking is required.** In frequent subgraph mining, when a new subgraph is generated, it needs to be checked against existing subgraphs to see whether an isomorphic subgraph has been generated. Such an operation is usually costly. In our DGTree construction algorithm, based on the following lemma, we do not need to perform such isomorphism checking operation for each new feature-graph.

Lemma 4.2: *For any two tree-nodes g_1 and g_2 in the* DGTree*, the feature-graphs of g_1 and g_2 are not isomorphic.* □

Proof: In Algorithm 1, after the first child tree-node g^+ of the root tree-node g_r is generated, $\mathcal{D}$ is divided into two disjoint parts $\mathcal{D}_1$ and $\mathcal{D}_2$, where $\mathcal{D}_1$ consists of the data-graphs containing the feature-graph of g^+ and $\mathcal{D}_2$ consists of the data-graphs not containing the feature-graph of g^+. $\mathcal{D}_1$ and $\mathcal{D}_2$ are further divided recursively. Note that for any tree-node g_1 that is directly/indirectly generated from $\mathcal{D}_1$, the feature-graph of g_1 contains the feature-graph of g^+; and for any tree-node g_2 that is directly/indirectly generated from $\mathcal{D}_2$, the feature-graph of g_2 does not contain the feature-graph of g^+. Therefore, g_1 and g_2 cannot be isomorphic. By applying this rule to all branches of the DGTree, it is easy to conclude that, for any two tree-nodes g_1 and g_2, if g_1 is not an ancestor / decedent of g_2, the feature-graphs of g_1 and g_2 are not isomorphic. Otherwise, g_1 is an ancestor / decedent of g_2. In this case, the feature-graphs of g_1 and g_2 are still not isomorphic since they contain different number of edges. □

• **Infrequent feature-graphs are allowed.** Due to the above two advantages, we can allow both frequent and infrequent subgraphs as feature-graphs. Because of the high selectivity, infrequent feature-graphs usually have a high pruning power and therefore are useful in query processing.

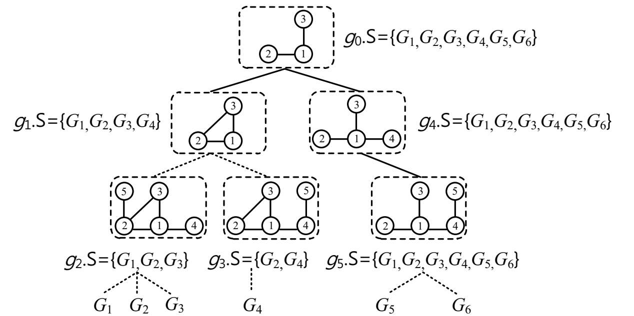


Fig. 2: An Example of DGTree

Example 4.1: Fig. 2 shows a partial DGTree of the graph database $\mathcal{D}$ in Fig. 1(a), with each tree-node g in the dashed box. Here, we omit the single-edge root node g_r and the one-edge-grow child nodes of g_1 for simplicity. We generate g_1 from g_0 first since g_1 has a CLOSE edge. Then growing from g_1, we have candidate feature-graphs g_2 and g_3. By computing $\mathsf{score}(g_2) = \frac{3}{4}$ and $\mathsf{score}(g_3) = \frac{2}{3}$, we choose g_2 and index G_1, G_2 and G_3 under g_2. After refining $g_3.\mathcal{S}^*$, we index G_4 under g_3. The remaining construction steps are similar, and we stop generating child tree-nodes when only one graph is left in current $g.\mathcal{S}^*$, which is regarded as a leaf node. Compared with PrefIndex, we consider both the cost sharing and pruning power of all feature-graphs even if it is not frequent (e.g. g_3), while PrefIndex will not take advantage of such features. □

C. Query-dependent Supergraph Search

In this subsection, we introduce our query processing algorithm, which computes the set of subgraphs $\mathcal{A}(Q)$ for a query-graph Q based on the constructed DGTree.

The Query Processing Algorithm. Given a query-graph Q, our query processing algorithm is shown in Algorithm 4. The input includes the query-graph Q and the root of the DGTree g_r. We use $\mathcal{A}(Q)$ to maintain the set of answers, and use $\mathcal{C}$ to maintain the set of candidate data-graphs that are neither in $\mathcal{A}(Q)$ nor pruned by any feature-graphs. We also use a heap $\mathcal{H}$ to maintain the set of feature-graphs to be processed. Each feature-graph entry q in $\mathcal{H}$ represents a feature-graph and it contains the following four components:

q.tree-node	The corresponding tree-node of q in the DGTree. Here, we call the feature-graph of q.tree-node the feature-graph of q for simplicity, and denote it as q.graph, i.e., q.graph $= q$.tree-node.graph
$q.\mathcal{S}^*$	The set of candidate data-graphs that contain the feature-graph of q.
$q.\mathcal{M}(Q)$	The set of matches of the feature-graph of q in Q.
q.score	the score of the feature-graph of q.

In Algorithm 4, we initialize $\mathcal{C}$ to be those data-graphs with size no larger than Q (line 1). Here we can also use some other trivial techniques, e.g., using degree sequence, to further reduce the number of data-graphs in $\mathcal{C}$. In line 2, we initialize $\mathcal{H}$ and $\mathcal{A}(Q)$ to be $\emptyset$. Then we create the first entry q in $\mathcal{H}$ (line 3). We initialize q.tree-node to be g_r, $q.\mathcal{S}^*$ to be $\mathcal{C}$, and $q.\mathcal{M}(Q)$ to be all matched edges in Q (line 4-5). In line 6, we compute the score of q and push it into $\mathcal{H}$. We will discuss how to compute the score of q later. In line 7-14, we process features iteratively in the DGTree until $\mathcal{C} = \emptyset$.

In each iteration (line 8-14), we compute the best entry in $\mathcal{H}$ by invoking BestFeature$(\mathcal{H}, \mathcal{C})$ which will be introduced later (line 8). Let g be the tree-node of q. We traverse all child nodes g^+ of g to find new feature-graphs to be processed (line 9). We consider two cases. First, if g^+ is a leaf tree-node (line 10), the feature-graph of g^+ is a data-graph. In this situation, we try to find a match f of g^+.graph in Q by directly extending the existing matches in $q.\mathcal{M}(Q)$ (line 11). If such a match exists, we add the corresponding data-graph into $\mathcal{A}(Q)$ (line 12). The data-graph is removed from $\mathcal{C}$ no matter whether f exists or not (line 13). Second, if g^+ is not a leaf tree-node, we create a corresponding new entry in $\mathcal{H}$ by invoking FeatureExpansion$(Q, q, g^+, \mathcal{H}, \mathcal{C})$ which will be introduced later (line 14). Finally, after $\mathcal{C}$ becomes $\emptyset$, we return $\mathcal{A}(Q)$ as the answer for query Q (line 15).

• **Procedure** BestFeature. The procedure to find the best feature-graph entry in $\mathcal{H}$ is shown in line 16-22 of Algorithm 4. Since after processing a feature-graph entry in $\mathcal{H}$, some graphs may be removed from $\mathcal{C}$, which may decrease the scores of some other entries in $\mathcal{H}$. Instead of updating all such scores after a new feature-graph is processed, in BestFeature, we handle the updates in a lazy manner. The process is similar to the process of finding the best candidate edge to grow in Algorithm 3, which iteratively updates the score of the top element of $\mathcal{H}$ (line 20) until it is up-to-date (line 18).

• **Procedure** FeatureExpansion. The procedure to compute the feature-graph entry for a new tree-node g^+ is shown in Algorithm 5. In line 1-2, we initialize the components of q^+. Here, $q^+.\mathcal{S}^*$ is initialized to be the set of data-graphs in both $g^+.\mathcal{S}$ and $\mathcal{C}$. Suppose the feature-graph of g^+ is obtained by growing the edge (u_i, u_j) on the feature-graph of the parent tree-node of g^+ (line 3). In line 4-9, we compute $q^+.\mathcal{M}(Q)$ by extending every match f in $q.\mathcal{M}(Q)$. Here q is the feature-graph entry for the parent of g^+. For each match f, we consider two cases. First, edge (u_i, u_j) is an OPEN edge (line 5). In this case, we traverse the neighbors v of node $f(u_i)$ in Q (line 6). If $v \notin f$, $(f(u_i), v)$ can match (u_i, u_j) and thus we add a new match $[f, v]$ into $q^+.\mathcal{M}(Q)$ (line 7). Second, edge (u_i, u_j) is a CLOSE edge. In this case, if $(f(u_i), f(u_j)) \in E(Q)$, f is a match of the feature-graph of q^+, and thus we add a new match f into $q^+.\mathcal{M}(Q)$ (line 8-9). After $q^+.\mathcal{M}(Q)$ is computed, if $q^+.\mathcal{M}(Q) \neq \emptyset$ (line 10), q^+ needs to be further expanded and thus we compute the score of q^+ and push it into $\mathcal{H}$ (line 11); Otherwise, all the data-graphs in $q^+.\mathcal{S}^*$ can be pruned (line 12).

Algorithm 4: SuperGraphSearch(query Q, DGTree with root g_r)

```
1  C ← {G | G ∈ g_r.S, |E(G)| ≤ |E(Q)|, |V(G)| ≤ |V(Q)|};
2  H ← ∅; A(Q) ← ∅;
3  q ← a new entry;
4  q.tree-node ← g_r; q.S* ← C; q.M(Q) ← ∅;
5  for (v, v') ∈ E(Q) do  q.M(Q) ← q.M(Q) ∪ {[v, v'], [v', v]};
6  compute q.score; H.Push(q);
7  while C ≠ ∅ do
8      q ← BestFeature(H, C); g ← q.tree-node;
9      for g+ ∈ g.children do
10         if |g+.children| = 0 then
11             search a match f of g+.graph by extending q.M(Q);
12             if f ≠ ∅ then  A(Q) ← A(Q) ∪ g+.S;
13             C ← C \ g+.S;
14         else  FeatureExpansion(Q, q, g+, H, C);
15 return A(Q);

16 Procedure BestFeature(heap H, candidate data-graph set C)
17 q ← H.Pop();
18 while q.S* ⊈ C do
19     q.S* ← q.S* ∩ C;
20     if q.S* ≠ ∅ then  {compute q.score; H.Push(q)};
21     q ← H.Pop();
22 return q;
```

Algorithm 5: FeatureExpansion(query-graph Q, entry q, tree-node g^+, heap $\mathcal{H}$, candidate data-graph set $\mathcal{C}$)

```
1  q+ ← a new entry;
2  q+.tree-node ← g+; q+.S* ← g+.S ∩ C; q+.M(Q) ← ∅;
3  (u_i, u_j) ← g+.grow-edge;
4  for match f ∈ q.M(Q) do
5      if g+.edge-type = OPEN then
6          for v ∈ Nbr(f(u_i), Q) do
7              if v ∉ f then  q+.M(Q) ← q+.M(Q) ∪ {[f, v]};
8      else if (f(u_i), f(u_j)) ∈ E(Q) then
9          q+.M(Q) ← q+.M(Q) ∪ {f};
10 if q+.M(Q) ≠ ∅ then
11     compute q+.score; H.Push(q+);
12 else  C ← C \ q+.S*;
```

Query-dependent Feature Score. We discuss how to compute the score of a feature-graph entry q. First, if the feature-graph of q is contained in a large number of candidate data-graphs, the cost sharing to process q is maximized. Therefore, the first

score of q can be defined as:

$$q.\mathsf{score}_1 = |q.\mathcal{S}^*| \quad (4)$$

Second, the feature-graph that has a small number of matches in Q will have a high pruning power. Therefore, the second score of q can be defined as:

$$q.\mathsf{score}_2 = 1/|q.\mathcal{M}(Q)| \quad (5)$$

Finally, by considering both computational cost sharing and pruning power, our final score for q is defined as:

$$q.\mathsf{score} = q.\mathsf{score}_1 \times q.\mathsf{score}_2 \quad (6)$$

Algorithm Analysis. Compared to existing feature based approaches for supergraph search, our query processing algorithm has the following advantages:

• **Verification-free query processing.** In existing feature based approaches, although cost sharing can be considered in the verification phase, the sharing is based on frequent subgraphs and thus the cost-saving is usually limited. In our approach, we create a full-structure index, and no verification is needed. The following lemma guarantees the correctness of our algorithm.

Lemma 4.3: *Algorithm 4 correctly computes $\mathcal{A}(Q)$.* □

Proof: First, each leaf tree-node in the DGTree represents a data-graph. Therefore, no false-positive exists in $\mathcal{A}(Q)$. Second, each data-graph is represented by a leaf tree-node in the DGTree. Therefore, no false-negative exists in $\mathcal{A}(Q)$. □

• **Adaptive online feature-graph selection.** Existing feature based approaches process features in a fixed order. As a result, when processing a feature with low pruning power, it is possible for the algorithm to be trapped in an exponential search space until all matches of the feature in the query-graph are found. In our algorithm, we consider dynamic feature processing, in which the score function for each feature-graph is query-dependent and it considers both the computational cost sharing and the pruning power of the feature. In addition, when data-graphs are included or excluded from answers in the process of query processing, our algorithm can update the scores for each feature-graph adaptively. In this way, features with low pruning power have a low opportunity to be processed and therefore the computational cost can be largely saved.

Example 4.2: Suppose we process supergraph search of query-graph Q shown in Fig. 1(b) based on the DGTree in Fig. 2. After matching g_0, we compute matches for its children g_1 and g_4 and then push them into $\mathcal{H}$. Since $\mathsf{score}(g_1) > \mathsf{score}(g_4)$, we pop g_1 first and process the descendants of g_1 in a similar way. By following such a dynamical selection, we include G_1 as an answer-graph and prune G_2, G_3, and G_4 via the descendants of g_1, since the dynamically computed scores of them are larger than g_4. Thus, only two candidate graphs are left in $\mathcal{C}$ when we deal with g_4, i.e., the adaptive online feature-graph selection technique can filter candidates as early as possible. □

V. Optimization Strategies

In this section, we introduce two optimization strategies to further improve our algorithm. The first strategy aims to save computational cost by compressing the data-graphs while preserving the structural information. The second strategy aims to optimize the query processing algorithm by considering both inclusion and exclusion of answers instead of only exclusion of answers using an adaptive node splitting technique.

A. *Cost-saving by Graph Compression*

Given a graph database $\mathcal{D}$, for a certain data-graph $G \in \mathcal{D}$, there may exist several nodes in $V(G)$ that are symmetrical. Given a query-graph Q, by applying Algorithm 4, finding all matches for the symmetric nodes in the query-graph Q may result in large redundant computations. This is because given any match of G in Q, each permutation of the symmetric nodes will result in a new match. In this subsection, we aim to compress each data-graph to reduce the number of its symmetric nodes and therefore reduce redundant computational cost. The following two conditions should be satisfied after graph compression:

(C_1): The structural information of each graph is preserved in the corresponding compressed graph. Here, by structure preservation, we mean that the original graph can be easily recovered from the compressed graph.

(C_2): After compression, Algorithm 1 and Algorithm 4 can be easily adapted for indexing and query processing.

Considering the above two conditions, in this paper, we adopt a lightweight graph-compression scheme as follows:

Graph Compression. We consider a simple case of node symmetry, namely, node equivalence, which is defined as:

Definition 5.1: (Node Equivalence) Given any graph G and a pair of nodes u and v in G, u and v are *equivalent* iff $\mathsf{Nbr}(u, G) = \mathsf{Nbr}(v, G)$. □

For each data-graph $G \in \mathcal{D}$, we use G^c to denote the compressed graph of G. G^c is constructed by contracting each set of equivalent nodes in G into a node C in G^c. For a node $v \in G$, we use C_v to denote the corresponding contracted node in G^c. For any pair of nodes C_v and $C_{v'}$ in $V(G^c)$, $(C_v, C_{v'}) \in E(G^c)$ iff $(v, v') \in E(G)$. We use $v \in C$ to denote that C is a contracted node for v, and we use $|C|$ to denote the number of equivalent nodes in G represented by C.

Indexing. Given a graph database $\mathcal{D}$, with the graph compression technique, we modify our DGTree index as follows: First, we compress each graph $G \in \mathcal{D}$ into G^c and create a compressed graph database $\mathcal{D}^c$. Second, we construct the DGTree on $\mathcal{D}^c$ as our index using a simple modification of Algorithm 1: We allow each feature-graph in the DGTree to contain contracted nodes, and we modify the match computation in Algorithm 1 by adding a size constraint. Specifically, given a tree-node g, for each contracted node C_u in the feature-graph of g, when we compute $g.\mathcal{M}(G^c)$ for a certain $G^c \in \mathcal{D}^c$, C_u can match a node $C_v \in V(G^c)$ only if $|C_u| \le |C_v|$.

Query Processing. Given a query-graph Q and the modified DGTree constructed on $\mathcal{D}^c$, in our modified query processing algorithm, for each entry q in the heap $\mathcal{H}$, each node C in the feature-graph of q should be matched to $|C|$ nodes in Q. This can be achieved by modifying Algorithm 5 as follows: In line 6 of Algorithm 5, instead of enumerating nodes in $\mathsf{Nbr}(f(u_i), Q)$, we enumerate all permutations of $|u_j|$ common neighbors of $f(u_i)$ in Q. Here, u_i, $f(u_i)$, and u_j may contain multiple nodes. In line 7, if a permutation does not overlap f, we add a corresponding new match into $q^+.\mathcal{M}(Q)$. Each such new match corresponds to $|u_j|!$ matches of the original feature-graph in Q. In line 8, since $f(u_i)$ and $f(u_j)$ may contain multiple nodes, we need to check whether every node in $f(u_i)$

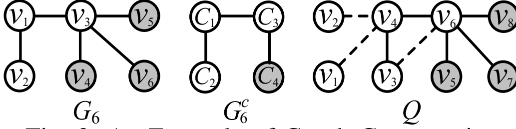


Fig. 3: An Example of Graph Compression

has an edge with every node in $f(u_j)$. Finally, we modify Eq. 5 for each entry q to compute a new q.score$_2$ as follows:

$$q.\mathsf{score}_2 = \frac{1}{|q.\mathcal{M}(Q)| \times \prod_{C \in V(q.\mathsf{graph})} |C|!} \tag{7}$$

Here, q.graph is the feature-graph of q.

The new query processing algorithm using graph compression improves Algorithm 4 by utilizing the symmetry information of the nodes in the data-graphs. Specifically, for each set of k symmetric nodes in the feature-graph (extracted from the data-graphs), we can use a match of k nodes in the query-graph to represent k! matches. This saves redundant computational cost when matching the k symmetric nodes.

Example 5.1: Fig. 3 shows an example of graph compression. We compress G_6 (Fig. 1(a)) into G_6^c by contracting v_4, v_5, and v_6 into C_4. During query processing, in a certain match of G_6^c in Q, C_4 can be matched to the set $\{v_5, v_7, v_8\}$ in Q . This match can represent 3! matches of G_6 in Q. □

B. Inclusion-aware Query Processing

Motivation. Given a query-graph Q, Algorithm 4 uses exclusion logic to prune graphs that are not subgraphs of Q using the DGTree. The algorithm iteratively selects the entry with the highest score from the heap $\mathcal{H}$ of candidate feature-graphs for pruning. However, after processing all entries with high pruning power in $\mathcal{H}$, the unpruned data-graphs in $q.\mathcal{S}^*$ for the entry q in heap $\mathcal{H}$ tend to be included in the answer set rather than be pruned by the entry. Here, the pruning power of an entry q in $\mathcal{H}$ is specified in Eq. 5 which is used to compute q.score (Eq. 6). In this case, we may generate a large number of useless matches for an entry q in the heap.

Solution Overview. In this subsection, to reduce the number of useless matches when processing an entry q with low pruning power, we introduce an inclusion based method to handle q. Specifically, we allow two types of entries in the heap, namely, *inclusion-entry* and *exclusion-entry*. We use q.type, which is either INCLUSION or EXCLUSION, to differentiate the two types of entries. Initially, all entries in the heap are exclusion-entries. When q.score (Eq. 6) for a certain entry q is smaller than a certain threshold δ, we switch to process q according to an inclusion based method using an adaptive node-splitting algorithm. The algorithm splits matches in $q.\mathcal{M}(Q)$ and handles these matches with different priorities. Intuitively, a match that is more likely to be expanded to include answers in $\mathcal{A}(Q)$ has a higher priority to be processed. For each match $f \in q.\mathcal{M}(Q)$, we use f.score to specify the score of f. A match with a higher score has a higher priority to be processed. Next we will introduce how to split a node and how to compute the score of a match.

Adaptive Node-splitting. The algorithm to split an entry q in $\mathcal{H}$ is shown in Algorithm 6. In line 1-2, we sort all matches in $q.\mathcal{M}(Q)$ in a non-increasing order of their scores and put them in $\mathcal{M}$. In line 3-9, we iteratively split q by dividing matches in $\mathcal{M}$ into different partitions, each of which forms a new entry.

Algorithm 6: NodeSplit(query-graph Q, entry q, heap $\mathcal{H}$)

```
1 M ← q.M(Q);
2 sort matches f ∈ M in non-increasing order of f.score;
3 while |M| > 0 do
4     q' ← a new entry;
5     q'.M(Q) ← the last ⌈|M|/2⌉ matches in M;
6     q'.type ← INCLUSION;
7     q'.tree-node ← q.tree-node; q'.S* ← q.S*;
8     compute q'.score; H.Push(q');
9     M ← M \ q'.M(Q);
```

For each iteration (line 4-9), we create a new entry q' (line 4) and put the last $\lceil |\mathcal{M}|/2 \rceil$ matches with lowest scores into $q'.\mathcal{M}(Q)$. By iteratively applying this on $\mathcal{M}$, we guarantee that matches with higher scores are put into an entry q' with lower $|q'.\mathcal{M}(Q)|$ and thus the entry q' has a higher priority to be processed according to Eq. 6. For instance, if $|q.\mathcal{M}(Q)| = 14$, $q.\mathcal{M}(Q)$ is divided into 4 partitions with 7, 4, 2, and 1 matches, and the match f with the highest f.score belongs to the last partition. In line 6, we assign the type of q' to be INCLUSION. In line 7, we set q'.tree-node and $q'.\mathcal{S}^*$ to be those in q. Then, we compute q'.score using Eq. 6 based on the new $q'.\mathcal{M}(Q)$ and push it into heap $\mathcal{H}$ (line 8). Finally, in line 9, we update $\mathcal{M}$ by removing $q'.\mathcal{M}(Q)$ since $\mathcal{M}$ will be used in the next iteration.

Based on the above node-splitting algorithm, we modify our query processing algorithm as follows: First, the root entry is an exclusion-entry, and each newly expanded entry inherits the type of its parent entry when it is created in line 1 of Algorithm 5. Second, after line 8 of Algorithm 4, for the entry q, we first check whether $q.\mathsf{score} < \delta$. If not, we process line 9-14; Otherwise, we invoke Algorithm 6 to split q. Finally, in line 12 of Algorithm 5 and line 13 of Algorithm 4, if q^+ is an inclusion-entry, we will not remove $q^+.\mathcal{S}^*$ from $\mathcal{C}$ only if the data-graphs are included in $\mathcal{A}(Q)$, since an inclusion-entry q^+ cannot be used to prune data-graphs in $q^+.\mathcal{S}^*$. In this case, it is possible that some data-graphs in $\mathcal{C}$, which are not answers, will not be pruned by any entry, so we modify the condition of the while loop in line 7 of Algorithm 4 to $\mathcal{H} \neq \emptyset$. Such data-graphs will be kept in $\mathcal{C}$ and be discarded eventually.

Inclusion-aware Match Score. Given a match $f \in q.\mathcal{M}(Q)$ for an entry q, we discuss how to compute f.score. Intuitively, if the match f appears in a denser region of Q, it is more likely to be expanded to include a certain data-graph in $q.\mathcal{S}^*$. We can use the degrees of nodes in f to model the density of the corresponding matched region. Based on this intuition, we define f.score as follows:

$$f.\mathsf{score} = \sum_{v \in f} d(v, Q) \tag{8}$$

Here, $d(v, Q)$ is the degree of node v in Q.

VI. Performance Studies

In this section, we present our experimental results by comparing our algorithm, denoted as DGTree, with two state-of-the-art supergraph search algorithms PrefIndex [31] and IGQuery [7]. The codes for PrefIndex and IGQuery are obtained from the authors. We evaluate our algorithms in several aspects: (1) the effectiveness of feature score functions and optimization strategies used in DGTree, (2) the efficiency of query processing by comparing DGTree with PrefIndex and IGQuery, and (3) the efficiency of indexing by comparing DGTree with PrefIndex (IGQuery is not compared since its indexing time can be omitted). All of our experiments are

TABLE I: PARAMETERS

Parameter	Range	Default
graph database size	1000, 3000, 5000, 7000, 9000	3000
avg. node # of data-graph	10, 30, 50, 70, 90	50
avg. node # of query-graph	60, 80, 100, 120, 140	100
data-graph density group	$DG_1, DG_2, DG_3, DG_4, DG_5$	–
query-graph density group	$DQ_1, DQ_2, DQ_3, DQ_4, DQ_5$	–

conducted on a machine with an Intel Core i5 3.1GHz CPU and 8GB main memory running Windows 7.

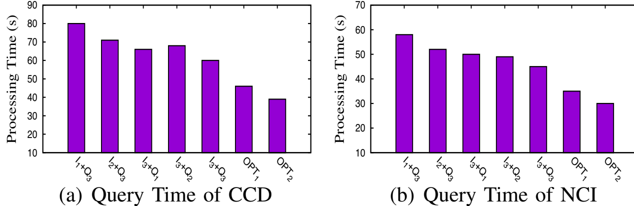
Datasets. We evaluate the performance of the algorithms on two real datasets: CCD and NCI.

• **CCD dataset** contains chemical compound structures, which is downloaded from the PubChem website. The average node number of the graphs in CCD is 40.75.

• **NCI dataset** is downloaded from the National Cancer Institute Open Database website. The average node number of the graphs in NCI is 40.48.

We show the parameters and default values used in our experiments in Table I. If not specified, the values of all parameters are set to their default values. For each testing in query processing, we randomly select 20 queries and take the average processing time. To evaluate the performance by varying data-graph density, we sort the CCD and NCI datasets in an increasing order of graph density and divide them into five groups, and then extract 3000 data-graphs with average node number of 50 from each group, denoted as DG_1, DG_2, DG_3, DG_4 and DG_5 respectively. Similarly, to test the effect of query-graph density, we extract five sets of query-graphs with average node number of 100, donated as DQ_1, DQ_2, DQ_3, DQ_4 and DQ_5 respectively.

A. *Score Functions and Optimization Strategies*



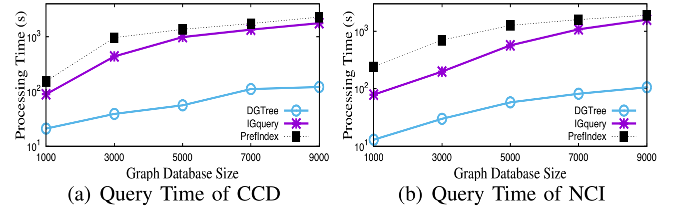
(a) Query Time of CCD (b) Query Time of NCI

Fig. 4: Score Functions and Optimization Strategies

We first test the effect of the proposed score functions on feature-graph selection during indexing and query processing. We also compare the basic algorithms with the optimized versions to show the effectiveness of the graph compression strategy and inclusion-aware query processing strategy. The results are shown in Fig. 4. We use I_1, I_2, and I_3 to denote the three score functions Eq. 1, Eq. 2 and Eq. 3 in DGTree construction respectively, and use Q_1, Q_2, and Q_3 to denote the three score functions Eq. 4, Eq. 5, and Eq. 6 in query processing respectively. For example, $I_1 + Q_3$ denotes the algorithm using Eq. 1 and Eq. 6 as the score functions for DGTree construction and query processing respectively. We use OPT_1 to denote the algorithm with the graph compression strategy based on $I_3 + Q_3$, and use OPT_2 to denote the algorithm with the inclusion-aware query processing strategy based on OPT_1. In OPT_2, we set $\delta = 0.001$. For score functions in indexing, Fig. 4 shows that $I_3 + Q_3$ improves $I_1 + Q_3$ and $I_2 + Q_3$ by 30% and 15% respectively. It is worth mentioning that the index construction of I_3 is about three times faster than I_2 and seven times faster than I_1. For score functions in query processing, Fig. 4 shows that $I_3 + Q_3$ is about 10% faster than both $I_3 + Q_1$ and $I_3 + Q_2$. Comparing $I_3 + Q_3$ with the optimized algorithms, we see that graph compression (OPT_1) accelerates query processing by around 30%, and inclusion-aware query processing (OPT_2) further improves the efficiency by 17%. When comparing with other algorithms in the following experiments, we apply the best score functions with all optimization techniques in DGTree.

B. *Query Processing:* DGTree, IGQuery, *and* PrefIndex

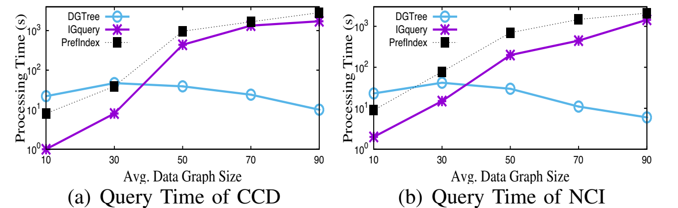
In this subsection, we evaluate the performance of query processing algorithms, by comparing DGTree with IGQuery and PrefIndex. We conduct extensive experiments to evaluate the query processing time by varying database size, data-graph size, data-graph density, query-graph size and query-graph density, so as to show the efficiency and scalability of DGTree.



(a) Query Time of CCD (b) Query Time of NCI

Fig. 5: Varying Database Size on Query Processing

Varying Database Size. In this experiment, we evaluate the scalability of query processing by varying database size from 1k to 9k. Fig. 5 reports the query processing time on CCD and NCI datasets. It shows that when the database is scaled up, the query processing time of all algorithms increases. We can see that DGTree is scalable when the database becomes larger. When the database size is larger than 5000, DGTree is over an order of magnitude faster than PrefIndex and IGQuery.



(a) Query Time of CCD (b) Query Time of NCI

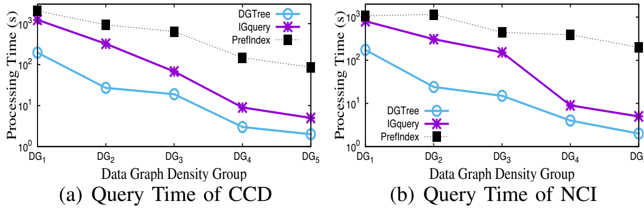
Fig. 6: Varying Data Graph Size on Query Processing

Varying Data Graph Size. We vary the average data-graph size to study the performance of query processing. The results are shown in Fig. 6. When data-graph is small-sized, e.g., no larger than 30, PrefIndex and IGQuery outperforms DGTree by several seconds. However, when the data-graph size increases, the computational cost of PrefIndex and IGQuery increases rapidly due to their defects in scalability. The reason is that, for PrefIndex, the cost saving by common prefix is limited for large-sized data-graphs, and for IGQuery, the number of directly included answers is small for large-sized data-graphs. DGTree, however, costs most when the data-graph size is 30, since the answer set is relatively large. But when the data-graph size increases, the computational cost is saved by the

https://pubchem.ncbi.nlm.nih.gov/

http://cactus.nci.nih.gov/download/nci/index.html

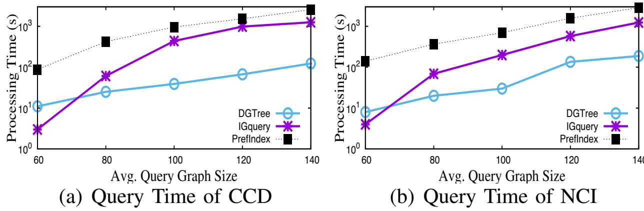
high pruning power of dynamically selected feature-graphs. DGTree is over an order of magnitude faster than the other two algorithms when the data-graph size is larger than 70.



(a) Query Time of CCD (b) Query Time of NCI

Fig. 7: Varying Data Graph Density on Query Processing

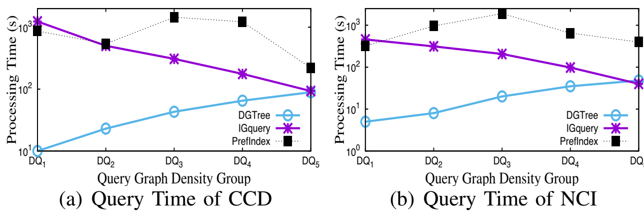
Varying Data Graph Density. We vary data-graph density to show the efficiency of the algorithms and report the results in Fig. 7. It shows that when the data-graphs become denser, the query processing cost decreases. The reason is that denser data-graphs result in denser feature-graphs, that are effective for pruning in query processing for all three algorithms. However, in PrefIndex, the problem of heavy verification costs still exists, and in IGQuery, the answers obtained by direct inclusion is rather limited, so the performance of PrefIndex and IGQuery is degraded, as shown in Fig. 7.



(a) Query Time of CCD (b) Query Time of NCI

Fig. 8: Varying Query Graph Size on Query Processing

Varying Query Graph Size. To evaluate the effect of query-graph size on query processing, we obtain five sets of query-graphs, with average node numbers of 60, 80, 100, 120 and 140 respectively. We conduct experiments on databases of 3000 data-graphs with average size of 50 in the CCD and NCI datasets. The results are shown in Fig. 8. We can see that when the query-graph grows larger, the processing time of all algorithms increases since it costs more for graph matching during query processing. By taking advantage of graph compression and inclusion-aware strategies, DGTree is scalable for large-sized query-graphs. Fig. 8 shows that DGTree is slower than IGQuery by several seconds when the query-graph is small, but outperforms IGQuery by several times when the query-graph size is larger than 60. Besides, DGTree is faster than PrefIndex by over an order of magnitude in most of the testings.



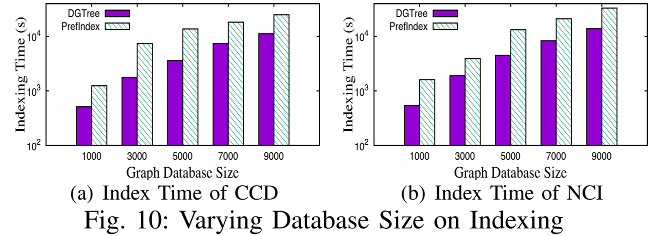
(a) Query Time of CCD (b) Query Time of NCI

Fig. 9: Varying Query Graph Density on Query Processing

Varying Query Graph Density. We vary the query-graph density to evaluate the performance of algorithms on query processing. The results are presented in Fig. 9. DGTree performs better for query-graphs with lower density. This is because DGTree is based on exclusion logic, and sparse query-graphs tend to have a smaller answer set. In contrast, IGQuery prefers denser queries since IGQuery may include more answers directly when processing a denser query . IGQuery on DQ_5 in the NCI dataset slightly outperforms DGTree, but is slower than DGTree in all other cases. The results of PrefIndex are unstable when the query-graph density varies, and DGTree is over an order of magnitude faster than PrefIndex in most of the testings.

C. *Indexing:* DGTree *and* PrefIndex

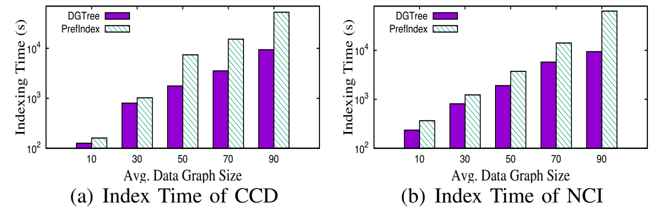
In this subsection, we compare the indexing performance of DGTree and PrefIndex by varying database size, data-graph size, and data-graph density. We conduct experiments on both the CCD and NCI datasets. Note that the index size of DGTree is only 22% of the index size of IGQuery, and 18% of the index size of PrefIndex on average, which benefits from the bounded number of generated features in DGTree. For example, on the setting of 3000 data-graphs with average size of 50 in the CCD dataset, the index sizes of DGTree, IGQuery and PrefIndex are $0.87MB$, $3.71MB$ and $4.53MB$ respectively.



(a) Index Time of CCD (b) Index Time of NCI

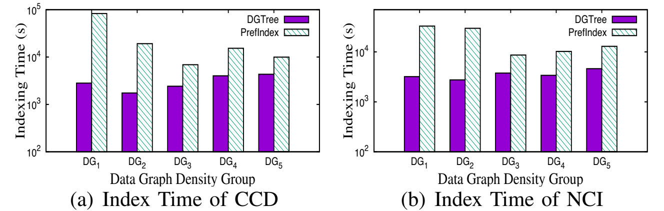
Fig. 10: Varying Database Size on Indexing

Varying Database Size. We test the effect of database size on the performance of graph indexing. Fig. 10 shows the indexing time of DGTree and PrefIndex when database size varies. When the number of data-graphs increases, the processing time for both DGTree and PrefIndex increases. In all testings, PrefIndex is much slower than DGTree as shown in Fig. 10.



(a) Index Time of CCD (b) Index Time of NCI

Fig. 11: Varying Data Graph Size on Indexing



(a) Index Time of CCD (b) Index Time of NCI

Fig. 12: Varying Data Graph Density on Indexing

Varying Data Graph Size. We vary data-graph size and report the indexing time of DGTree and PrefIndex in Fig. 11. From Fig. 11, we can see that both DGTree and PrefIndex are efficient when data-graphs are small. But with the growing of data-graph size, the cost of PrefIndex exponentially increases (e.g. over 14 hours for 90-sized graphs in CCD), while the cost

of DGTree is more stable. This is due to the expensive mining cost in PrefIndex, which is not scalable for large data-graphs.

Varying Data Graph Density. In this experiment, we evaluate the performance of indexing by varying graph density. The results are presented in Fig. 12. In all testings, DGTree performs stably and it is from 1.5 to 30 times faster than PrefIndex.

VII. Related Work

We review the related work of supergraph search problem from three categories, namely, subgraph search, supergraph search, and similar graph search.

Subgraph Search. Subgraph search is the most common graph search type and has been extensively studied in the literature. Existing algorithms include Ullmann [19], VF2 [9], gIndex [23], TreePi [28], Tree+δ [30], Fg-index [8], GString [13], QuickSI [16], GraphQL [12], TurboIso [11], etc. Most of them follow a backtracking framework, first proposed by Ullmann [19]. Other works accelerate the matching process using pruning-verification and matching-order optimizations.

Supergraph Search. Supergraph search has become increasingly popular in recent years. Existing solutions mainly include CIndex [6], GPTree [29], PrefIndex [31], IGQuery [7], and LWIndex [24]. More details can be found in Section III.

Similar Graph Search. Similarity graph search aims to find graphs that are similar to the query-graph. Existing techniques mainly fall into three categories, the propagation-based paradigm [2, 15, 17], whose intuition is that two nodes are similar if their neighbors are similar; the spectral-based paradigm [1, 3, 4, 14, 20, 21], due to the fact that two graphs are isomorphic if their adjacency matrices have the same eigenvalues and eigenvectors; and optimization-based paradigm [26, 27], which transfers graph matching to an optimization problem.

VIII. Conclusions

In this paper, we study the supergraph search problem which is a fundamental problem in graph databases. Existing solutions are not scalable when processing large-sized data-graphs and query-graphs. In our approach, we build a full-structure feature-tree as our index, that does not rely on an expensive frequent subgraph mining algorithm, and can generate better features. We propose a verification-free algorithm that processes features in a query-dependent order for query processing. We further explore two optimization strategies to improve the efficiency of our algorithm. We conduct extensive experiments on two large real datasets to demonstrate the high scalability of our approach.

Acknowledgements

Lu Qin is supported by ARC DE140100999 and ARC DP160101513. Xuemin Lin is supported by NSFC61232006, ARC DP140103578, and ARC DP150102728. Lijun Chang is supported by ARC DE150100563 and ARC DP160101513. Jeffrey Xu Yu is supported by Research Grants Council of the Hong Kong SAR, China No. 14209314 and 418512.

References

[1] M. Belkin and P. Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation*, 15(6), 2003.

[2] V. D. Blondel, A. Gajardo, M. Heymans, P. Senellart, and P. Van Dooren. A measure of similarity between graph vertices: Applications to synonym extraction and web searching. *SIAM review*, 46(4), 2004.

[3] T. Caelli and S. Kosinov. An eigenspace projection clustering method for inexact graph matching. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(4), 2004.

[4] T. Caelli and S. Kosinov. Inexact graph matching using eigen-subspace projection clustering. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(03), 2004.

[5] M. Cannataro, P. H. Guzzi, and P. Veltri. Protein-to-protein interactions: Technologies, databases, and algorithms. *ACM Comput. Surv.*, 43(1), 2010.

[6] C. Chen, X. Yan, P. S. Yu, J. Han, D. Zhang, and X. Gu. Towards graph containment search and indexing. In *Proc. of VLDB'07*, 2007.

[7] J. Cheng, Y. Ke, A. W. Fu, and J. X. Yu. Fast graph query processing with a low-cost index. *VLDB J.*, 20(4), 2011.

[8] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *Proc. of SIGMOD'07*, 2007.

[9] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(10), 2004.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[11] W.-S. Han, J. Lee, and J.-H. Lee. Turbo iso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proc. of SIGMOD'13*, 2013.

[12] H. He and A. K. Singh. Query language and access methods for graph databases. In *Managing and Mining Graph Data*. Springer, 2010.

[13] H. Jiang, H. Wang, P. S. Yu, and S. Zhou. Gstring: A novel approach for efficient search in graph databases. In *Proc. of ICDE'07*, 2007.

[14] D. Knossow, A. Sharma, D. Mateus, and R. Horaud. Inexact matching of large and sparse graphs using laplacian eigenvectors. In *Graph-Based Representations in Pattern Recognition*. Springer, 2009.

[15] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proc. of ICDE'02*, 2002.

[16] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1), 2008.

[17] R. Singh, J. Xu, and B. Berger. Pairwise global alignment of protein interaction networks by matching neighborhood topology. In *Research in computational molecular biology*, 2007.

[18] A. Trémeau and P. Colantoni. Regions adjacency graph applied to color image segmentation. *IEEE Transactions on Image Processing*, 9(4), 2000.

[19] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1), 1976.

[20] S. Umeyama. An eigendecomposition approach to weighted graph matching problems. *PAMI*, 10(5), 1988.

[21] L. Xu and I. King. A pca approach for fast retrieval of structural patterns in attributed graphs. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 31(5), 2001.

[22] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proc. of ICDM'02*, 2002.

[23] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *Proc. of SIGMOD'04*, 2004.

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

[25] D. Yuan, P. Mitra, H. Yu, and C. L. Giles. Iterative graph feature mining for graph indexing. In *Proc. of ICDE'12*, 2012.

[26] M. Zaslavskiy, F. Bach, and J.-P. Vert. Global alignment of protein–protein interaction networks by graph matching methods. *Bioinformatics*, 25(12), 2009.

[27] M. Zaslavskiy, F. Bach, and J.-P. Vert. A path following algorithm for the graph matching problem. *PAMI*, 31(12), 2009.

[28] S. Zhang, M. Hu, and J. Yang. Treepi: A novel graph indexing method. In *Proc. of ICDE'07*, 2007.

[29] S. Zhang, J. Li, H. Gao, and Z. Zou. A novel approach for efficient supergraph query processing on graph databases. In *Proc. of EDBT'09*, 2009.

[30] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: Tree + delta >= graph. In *Proc. of VLDB'07*, 2007.

[31] G. Zhu, X. Lin, W. Zhang, W. Wang, and H. Shang. Prefindex: An efficient supergraph containment search technique. In *Proc. of SSDBM'10*, 2010.

[32] 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), 2005.