Chapter 5

GRAPH INDEXING

Xifeng Yan
Department of Computer Science
University of California at Santa Barbara
xyan@cs.ucsb.edu

Jiawei Han
Department of Computer Science
University of Illinois at Urbana-Champaign
hanj@cs.uiuc.edu

Abstract Advanced database systems face a great challenge arising from the emergence of massive, complex structural data in bioinformatics, chem-informatics, business processes, etc. One of the most important functions needed in these areas is efficient search of complex graph data. Given a graph query, it is desirable to retrieve relevant graphs quickly from a large database via efficient graph indices. This chapter gives an introduction to graph substructure search, approximate substructure search and their related graph indexing techniques, particularly feature-based graph indexing.

Keywords: Frequent pattern, graph index, graph query, similarity search

1. Introduction

Development of scalable methods for analyzing large graph data sets, including graphs built from chemical structures and biological networks, poses great challenges. At the core of many graph analysis applications, lies a common and critical problem: how to efficiently search graphs.

Given a graph database $D = \{G_1, G_2, \ldots, G_n\}$ and a graph query Q, *graph search* returns a query answer set $D_Q = \{G|M(Q,G) = 1, G \in D\}$, where M is a boolean function. M could be a function testing graph isomorphism (full structure search), subgraph isomorphism (substructure search), approximate match (full structure similarity search), and subgraph approximate match (substructure similarity search). It is inefficient to perform a sequential scan on a graph database and check each graph to find answers to a query graph. Sequential scan is costly because one has to not only access the whole graph database but also check (sub)graph isomorphism. It is known that subgraph isomorphism is an NP-complete problem [8]. Therefore, high performance graph indexing is needed to quickly prune graphs that obviously violate the query requirement.

The problem of graph search has been addressed in different domains since it is a critical problem for many applications. In content-based image retrieval, Petrakis and Faloutsos [25] represented each graph as a vector of features and indexed graphs in a high dimensional space using R-trees. Shokoufandeh et al. [29] indexed graphs by a signature computed from the eigenvalues of adjacency matrices. Instead of casting a graph to a vector form, Berretti et al. [2] proposed a metric indexing scheme which organizes graphs hierarchically according to their mutual distances. The SUBDUE system developed by Holder et al. [17] uses minimum description length to discover substructures that compress graph data and represent structural concepts in the data. In 3D protein structure search, algorithms using hierarchical alignments on secondary structure elements [21], or geometric hashing [35], have already been developed. There are other literatures related to graph retrieval that we are not going to enumerate here.

In semistructured/XML databases, query languages built on path expressions become popular. Efficient indexing techniques for path expression were initially introduced in DataGuide [13] and 1-index [23]. A(k)-index [20] proposes k-bisimilarity to exploit local similarity existing in semistructured databases. APEX [7] and D(k)-index [5] consider the adaptivity of index structure to fit the query load. Index Fabric [9] represents every path in a tree as a string and stores it in a Patricia trie. For more complicated graph queries, Shasha et al. [28] extended the path-based technique to do full scale graph retrieval, which is also used in the Daylight system [18]. Srinivasa et al. [30] built indices based on multiple vector spaces with different abstract levels of graphs.

This chapter introduces feature-based graph indexing techniques that facilitate graph substructure search in graph databases with thousands of instances. Nevertheless, similar techniques can also be applied to indexing single massive graphs.

2. Feature-Based Graph Index

Definition 5.1 (Substructure Search). *Given a graph database $D = \{G_1, G_2, \ldots, G_n\}$ and a query graph Q,* substructure search *is to find all the graphs that contain Q.*

Substructure search is one kind of basic graph queries, observed in many graph-related applications. Feature-based graph indexing is designed to answer substructure search queries, which consists of the following two major steps:

Index construction: It precomputes features from a graph database and builds indices based on these features. There are various kinds of features that could be used, including node/edge labels, paths, trees, and subgraphs. Let F be a feature set for a given graph database D. For any feature $f \in F$, D_f is the set of graphs containing f, $D_f = \{G|f \subseteq G, G \in D\}$. We define a null feature, $f_\varnothing$, which is contained by any graph. An inverted index is built between F and D: D_f could be the ids of graphs containing f, which is similar to inverted index in document retrieval [1].

Query processing: It has three substeps: (1) *Search*, which enumerates all the features in a query graph, Q, to compute the *candidate query answer set*, $C_Q = \bigcap_f D_f$ ($f \subseteq Q$ and $f \in F$); each graph in C_Q contains all of Q's features. Therefore, D_Q is a subset of C_Q. (2) *Fetching*, which retrieves the graphs in the candidate answer set from disks. (3) *Verification*, which checks the graphs in the candidate answer set to verify if they really satisfy the query. The candidate answer set is verified to prune false positives.

The *Query Response Time* of the above search framework is formulated as follows,

$$T_{search} + |C_Q| * (T_{io} + T_{iso\_test}), \tag{5.1}$$

where T_{search} is the time spent in the search step, T_{io} is the average I/O time of fetching a candidate graph from the disk, and T_{iso_test} is the average time of checking a subgraph isomorphism, which is conducted over query Q and graphs in the candidate answer set.

The candidate graphs are usually scattered around the entire disk. Thus, T_{io} is the I/O time of fetching a block on a disk (assume a graph can be accommodated in one disk block). The value of T_{iso_test} does not change much for a given query. Therefore, the key to improve the query response time is to minimize the size of the candidate answer set as much as possible. When a database is so large that the index cannot be held in main memory, T_{search} will affect the query response time.

Since all the features in the index contained by a query are enumerated, it is important to maintain a compact feature set in the memory. Otherwise, the cost of accessing the index may be even greater than that of accessing the database itself.

2.1 Paths

One solution to substructure search is to take paths as features to index graphs: Enumerate all the existing paths in a database up to a $maxL$ length and use them as features to index, where a path is a vertex sequence, $v_1, v_2, \ldots, v_k$, s.t., $\forall 1 \leq i \leq k-1$, (v_i, v_{i+1}) is an edge. It uses the index to identify graphs that contain all the paths (up to the $maxL$ length) in the query graph.

This approach has been widely adopted in XML query processing. XML query is one kind of graph query, which is usually built around path expressions. Various indexing methods [13; 23; 9; 20; 7; 28; 5] have been developed to process XML queries. These methods are optimized for path expressions and tree-structured data. In order to answer arbitrary graph queries, GraphGrep and Daylight systems were proposed in [28; 18]. All of these methods take *path* as the basic indexing unit; we categorize them as *path-based indexing*. The path-based approach has two advantages: (1) Paths are easier to manipulate than trees and graphs, and (2) The index space is predefined: All the paths up to the $maxL$ length are selected. In order to answer tree- or graph-structured queries, a path-based approach has to break query graphs into paths, search each path separately for the graphs containing the path, and join the results. Since the structural information could be lost when query graphs are decomposed to paths, likely many false positive candidates will be returned. In addition, a graph database may contain millions of different paths if it is large and diverse. These disadvantages motivate the search of new indexing features.

2.2 Frequent Structures

A straightforward approach of extending paths is to involve more complicated features, e.g., all of substructures extracted from a graph database. Unfortunately, the number of substructures could be even more than the number of paths, leaving an exponential index structure in practice. One solution is to set a threshold of substructures' frequency and only index those frequent ones.

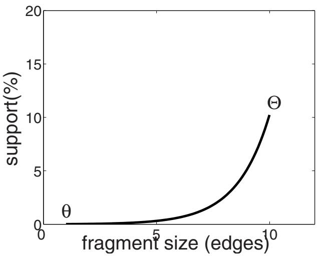
Definition 5.2 (Frequent Structures). *Given a graph database* $D = \{G_1, G_2, \ldots, G_n\}$ *and a graph structure* f*, the* support *of* f *is defined as* $sup(f) = |D_f|$*, whereas* D_f *is referred as* f*'s* supporting graphs. *With a predefined threshold* min_sup*,* f *is said to be* frequent *if* $sup(f) \geq min\_sup$*.*

Frequent structures could be used as features to index graphs. Given a query graph Q, if Q is frequent, the graphs containing Q can be retrieved directly since Q is indexed. Otherwise, we sort all Q's subgraphs in the support decreasing order: $f_1, f_2, \ldots, f_n$. There must exist a boundary between f_i and f_{i+1} where $|D_{f_i}| \geq$ *min_sup* and $|D_{f_{i+1}}| <$ *min_sup*. Since all the frequent structures with minimum support *min_sup* are indexed, one can compute the candidate answer set C_Q by $\bigcap_{1 \leq j \leq i} D_{f_j}$, whose size is at most $|D_{f_i}|$. For many queries, $|D_{f_i}|$ is close to *min_sup*. Therefore, the cost of verifying C_Q is minimal when *min_sup* is low.

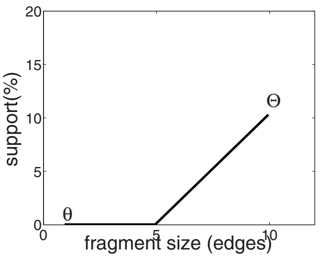
Unfortunately, for low support queries (i.e., queries whose answer set is small), the size of candidate answer set C_Q is related to the setting of *min_sup*. If *min_sup* is set too high, C_Q might be very large. If *min_sup* is set too low, it could be difficult to generate all the frequent structures due to the exponential pattern space.

Should a uniform *min_sup* be enforced for all the frequent structures? In order to reduce the overall index size, it is appropriate to have a *low* minimum support on *small* structures (for effectiveness) and a *high* minimum support on *large* structures (for compactness). This criterion of selecting frequent structures for effective indexing is called *size-increasing support constraint*.

Definition 5.3 (Size-increasing Support). *Given a monotonically nondecreasing function, $\psi(l)$, structure f is frequent under the* size-increasing support constraint *if and only if $|D_f| \geq \psi(size(f))$, and $\psi(l)$ is a* size-increasing support function.



(a) Exponential



(b) Piecewise-linear

Figure 5.1. Size-increasing Support Functions

Figure 5.1 shows two size-increasing support functions: *exponential* and *piecewise-linear*. One could select size-1 structures with a minimum support θ and larger structures with a higher support until we exhaust structures up to the size of $maxL$ with a minimum support Θ.

The size-increasing support constraint will select and index small structures with low minimum supports and large structures with high minimum supports.

This method has two advantages: (1) the number of frequent structures so obtained is much smaller than that using a low uniform support, and (2) low-support large structures could be well indexed by their smaller subgraphs. The first advantage also shortens the mining process when graphs have big structures in common.

2.3 Discriminative Structures

Among similar structures with the same support, it is often sufficient to index only the *smallest common substructures* since more query graphs may contain these structures (higher coverage). That is to say, if f', a supergraph of f, has the same support as f, it will not be able to provide more information than f if both are selected as indexing features. That is, f' is not more *discriminative* than f. This concept can be extended to a collection of subgraphs.

Definition 5.4 (Redundant Structure). *Structure x is redundant with respect to a feature set F if D_x is close to $\bigcap_{f \in F \wedge f \subseteq x} D_f$.*

Each graph in $\bigcap_{f \in F \wedge f \subseteq x} D_f$ contains all x's subgraphs in the feature set F. If D_x is close to $\bigcap_{f \in F \wedge f \subseteq x} D_f$, it implies that the presence of structure x in a graph can be predicted well by the presence of its subgraphs. Thus, x should not be used as an indexing feature since it does not provide new benefits to pruning if its subgraphs are being indexed. In such case, x is a redundant structure. In contrast, there are structures that are not redundant, called *discriminative structures*.

Let $f_1, f_2, \ldots$, and f_n be the indexing structures. Given a new structure x, the discriminative power of x can be measured by

$$Pr(x|f_{\varphi_1}, \ldots, f_{\varphi_m}), f_{\varphi_i} \subseteq x, 1 \leq \varphi_i \leq n. \tag{5.2}$$

Eq. (5.2) shows the probability of observing x in a graph given the presence of $f_{\varphi_1}, \ldots$, and f_{φ_m}. *Discriminative ratio*, γ, is defined as $1/Pr(x|f_{\varphi_1}, \ldots, f_{\varphi_m})$, which could be calculated by the following formula:

$$\gamma = \frac{|\bigcap_i D_{f_{\varphi_i}}|}{|D_x|}, \tag{5.3}$$

where D_x is the set of graphs containing x and $\bigcap_i D_{f_{\varphi_i}}$ is the set of graphs containing the features belonging to x. In order to mine discriminative structures, a minimum discriminative ratio γ_{min} is selected; those structures whose discriminative ratio is at least γ_{min} are retained as indexing features. The structures are mined in a level-wise manner, from small size to large size. The concept of indexing discriminative frequent structures, called gIndex, was first introduced by Yan et al. [36]. gIndex is able to achieve better performance in comparison with path-based methods.

For a feature $x \subseteq Q$, the operation, $C_Q = C_Q \cap D_x$ could reduce the candidate answer set by intersecting the id lists of C_Q and D_x. One interesting question is how to reduce the number of intersection operations. Intuitively, if a query Q has two structures, $f_x \subset f_y$, then $C_Q \bigcap D_{f_x} \bigcap D_{f_y} = C_Q \bigcap D_{f_y}$. Thus, it is not necessary to intersect C_Q with D_{f_x}. Let $F(Q)$ be the set of discriminative structures contained in the query graph Q, i.e., $F(Q) = \{f_x | f_x \subseteq Q \wedge f_x \in F\}$. Let $F_m(Q)$ be the set of structures in $F(Q)$ that are not contained by other structures in $F(Q)$, i.e., $F_m(Q) = \{f_x | f_x \in F(Q), \nexists f_y, s.t., f_x \subset f_y \wedge f_y \in F(Q)\}$. The structures in $F_m(Q)$ are called *maximal discriminative structures*. In order to calculate C_Q, one only needs to perform intersection operations on the id lists of maximal discriminative structures.

2.4 Closed Frequent Structures

Graph query processing that applies feature-based graph indices often requires a post verification step that finds true answers from a candidate answer set. If the candidate answer set is large, the verification step might take a long time to finish. Fortunately, a query graph having a large answer set is likely a frequent graph, which can be very efficiently processed using the frequent structure based index without any post verification. If the query graph is not a frequent structure, the candidate answer set obtained from the frequent structure based index is likely small; hence the number of candidate verifications should be minimal. Based on this observation, Cheng et al. [6] investigated the issue arising from frequent structure based indexing. As discussed before, the number of frequent structures could be exponential, indicating a huge index, which might not fit into main memory. In this case, the query performance will be degraded, since graph query processing has to access disks frequently. Cheng et al. [6] proposed using δ-Tolerance Closed Frequent Subgraphs (δ-TCFGs) to compress the set of frequent structures. Each δ-TCFG can be regarded as a representative supergraph of a set of frequent structures. An outer inverted-index is built on the set of δ-TCFGs, which is resident in main memory. Then, an inner inverted-index is built on the cluster of frequent structures of each δ-TCFG, which is resident in disk. Using this two-level index structure, many graph queries could be processed directly without verification.

2.5 Trees

Zhao et al. [38] analyzed the effectiveness and efficiency of paths, trees, and graphs as indexing features from three aspects: feature size, feature selection cost, and pruning power. Like paths and graphs, tree features can be effectively and efficiently used as indexing features for graph databases. It was observed that the majority of frequent graph patterns discovered in many applications are tree structures. Furthermore, if the distribution of frequent trees and graphs is similar, likely they will share similar pruning power.

Since tree mining can be performed much more efficiently than graph mining, Zhao et al. [38] proposed a new graph indexing mechanism, called Tree+Δ, which first mines and indexes frequent trees, and then on-demand selects a small number of discriminative graph structures from a query, which might prune graphs more effectively than tree features. The selection of discriminative graph structures is done on-the-fly for a given query. In order to do so, the pruning power of a graph structure is estimated approximately by its subtree features with upper/lower bounds. Given a query, Tree+Δ enumerates all the frequent subtrees of Q up to the maximum size $maxL$. Based on the obtained frequent subtree feature set of Q, $T(Q)$, it computes the candidate answer set, C_Q, by intersecting the supporting graph set of t, for all $t \in T(Q)$. If Q is a non-tree cyclic graph, it obtains a set of discriminative non-tree features, F. These non-tree features, f, may be cached already in previous search. If not, Tree+Δ will scan the graph database and build an inverted index between f and graphs in D. Then it intersects C_Q with the supporting graph set D_f.

GCoding [39] is another tree-based graph indexing approach. For each node u, it extracts a level-n path tree, which consists of all n-step simple pathes from u in a graph. The node is then encoded with eigenvalues derived from this local tree structure. If a query graph Q is a subgraph of a graph G, for each vertex u in Q, there must exist a corresponding vertex u' in G such that the local structure around u in Q should be preserved around u' in G. There is a partial order relationship between the eigenvalues of these two local structures. Based on this property, GCoding could quickly prune graphs that violate the order.

GString [19] combines three basic structures together: path, star, and cycle for graph search. It first extracts all of cycles in a graph database and then finds the star and path structures in the remaining dataset. The indexing methodology of GString is different from the feature-based approach. It transforms graphs into string representations and treats the substructure search problem as a substring match problem. GString relies on suffix tree to perform indexing and search.

2.6 Hierarchical Indexing

Besides the feature-based indexing methodology, it is also possible to organize graphs in a hierarchical structure to facilitate graph search. Close-tree [15] and GDIndex [34] are two examples of hierarchical graph indexing.

Closure-tree organizes graphs hierarchically where each node in the hierarchical structure contains summary information about its descendants. Given two graphs and an isomorphism mapping between them, one can take an elementwise union of the two graphs and obtain a new graph where the attribute of vertices and edges is a union of their corresponding attribute values in the two graphs. This union graph summarizes the structural information of both graphs, and serves as their bounding box [15], akin to a Minimum Bounding Rectangle (MBR) in traditional index structures. There are two steps to process a graph query Q using the closure-tree index: (1) Traverse the closure tree and prune nodes (graphs) based on a pseudo subgraph isomorphism; (2) Verify the remaining graphs to find the real answers. The pseudo subgraph isomorphism performs approximate subgraph isomorphism testing with high accuracy and low cost.

GDIndex [34] proposes indexing the complete set of the induced subgraphs in a graph database. It organizes the induced subgraphs in a DAG structure and builds a hash table to cross-index the nodes in the DAG structure. Given a query graph, GDIndex first identifies the nodes in the DAG structure that share the same hash code with the query graph, and then their canonical codes are compared to find the right answers. Unfortunately, the index size of GDIndex could be exponential due to a large number of induced subgraphs. It was suggested to place a limit on the size of indexed subgraphs.

3. Structure Similarity Search

A common problem in graph search is: what if there is no match or very few matches for a given query graph? In this situation, a subsequent query refinement process has to be taken in order to find the structures of interest. Unfortunately, it is often too time-consuming for a user to manually refine the query. One solution is to ask the system to find graphs that approximately contain the query graph. This structure similarity search problem has been studied in various fields. Willett et al. [33] summarized the techniques of fingerprint-based and graph-based similarity search in chemical compound databases. Raymond et al. [27] proposed a three tier algorithm for full structure similarity search. Nilsson[24] presented an algorithm for the pairwise approximate substructure matching. The matching is greedily performed to minimize a distance function for two graphs. Hagadone [14] recognized the importance of substructure similarity search in a large set of graphs. He used atom and edge labels to do screening. Messmer and Bunke [22] studied the reverse substructure similarity search problem in computer vision and pattern recognition. In [28], Shasha et al. also extended their substructure search algorithm to support queries with wildcards, i.e. don't care nodes and edges. In the following discussion, we will introduce feature-based graph indexing for substructure similarity search.

Definition 5.5 (Substructure Similarity Search). *Given a graph database* $D = \{G_1, G_2, \ldots, G_n\}$ *and a query graph* Q*,* substructure similarity search *is to discover all the graphs that approximately contain* Q*.*

Definition 5.6 (Substructure Similarity). *Given two graphs G and Q, if P is the maximum common subgraph of G and Q, then the substructure similarity between G and Q is defined by* $\frac{|E(P)|}{|E(Q)|}$*, and* $\theta = 1 - \frac{|E(P)|}{|E(Q)|}$ *is called relaxation ratio.*

Besides the common subgraph similarity measure, graph edit distance could also be used to measure the similarity between two graphs. It calculates the minimum number of edit operations (insertion, deletion, and substitution) needed to transform one graph into another [3].

3.1 Feature-Based Structural Filtering

Given a relaxed query graph, there is a connection between structure-based similarity and feature-based similarity, which could be used to leverage feature-based graph indexing techniques for similarity search.

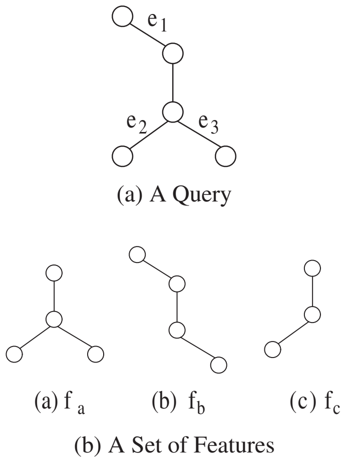


Figure 5.2. Query and Features

Figure 5.2(a) shows a query graph and Figure 5.2(b) depicts three structural fragments. Assume that these fragments are indexed as features in a graph database. Suppose there is no match for this query graph in a graph database. Then a user may relax one edge, e.g., e_1, e_2, or e_3, through a deletion operation. No matter which edge is relaxed, the relaxed query graph should have at least three embeddings of these features. That is, the relaxed query graph may *miss* at most four embeddings of these features in comparison with the seven embeddings in the original query graph: one f_a, two f_b's, and four f_c's. According to this constraint, graphs that do not contain at least three embeddings of these features could be safely pruned. This filtering concept is called *feature-based structural filtering*. In order to facilitate feature-based filtering, an index structure is developed, referred to *feature-graph matrix* [12; 28]. Each column of the feature-graph matrix corresponds to a target graph in the graph database, while each row corresponds to a feature being indexed. Each entry records the number of the embeddings of a specific feature in a target graph.

3.2 Feature Miss Estimation

	f_a	$f_{b(1)}$	$f_{b(2)}$	$f_{c(1)}$	$f_{c(2)}$	$f_{c(3)}$	$f_{c(4)}$
e_1	0	1	1	1	0	0	0
e_2	1	1	0	0	1	0	1
e_3	1	0	1	0	0	1	1

Figure 5.3. Edge-Feature Matrix

In order to calculate the maximum feature misses for a given relaxation ratio, we introduce *edge-feature matrix* that builds a map between edges and features for a query graph. In this matrix, each row represents an edge while each column represents an embedding of a feature. Figure 5.3 shows the matrix built for the query graph in Figure 5.2(a) and the features shown in Figure 5.2(b). All of the embeddings are recorded. For example, the second and the third columns are two embeddings of feature f_b in the query graph. The first embedding of f_b *cover*s edges e_1 and e_2 while the second covers edges e_1 and e_3. The middle edge does not appear in the edge-feature matrix if a user prefers retaining it. We say that an edge e_i *hit*s a feature f_j if f_j covers e_i.

The feature miss estimation problem is formulated as follows: *Given a query graph Q and a set of features contained in Q, if the relaxation ratio is θ, what is the maximum number of features that can be missed*? In fact, it is the maximum number of columns that can be hit by k rows in the edge-feature matrix, where $k = \lfloor \theta \cdot |G| \rfloor$. This is a classic maximum coverage (or set k-cover) problem, which has been proved NP-complete. The optimal solution that finds the maximal number of feature misses can be approximated by a greedy algorithm [16]. The greedy algorithm first selects a row that hits the largest number of columns and then removes this row and the columns covering it. This selection and deletion operation is repeated until k rows are removed. The number of columns removed by this greedy algorithm provides a way to estimate the upper bound of feature misses. Although the bound derived by the greedy algorithm cannot be improved asymptotically, it is possible to improve the greedy algorithm in practice by exhaustively searching the most selective features [37].

3.3 Frequency Difference

Once the upper bound of feature misses is obtained, it could be used to prune graphs. Let f_1, f_2, ..., f_n be the indexing features. Given a target graph G and a query graph Q, let $\mathbf{u} = [u_1, u_2, \ldots, u_n]^T$ and $\mathbf{v} = [v_1, v_2, \ldots, v_n]^T$ be their corresponding feature vectors, where u_i and v_i are the frequencies (i.e., the number of embeddings) of feature f_i in graphs G and Q. Figure 5.4 shows the two feature vectors $\mathbf{u}$ and $\mathbf{v}$. As mentioned before, for any feature set, the corresponding feature vector of a target graph can be obtained from the feature-graph matrix directly without scanning the graph database.

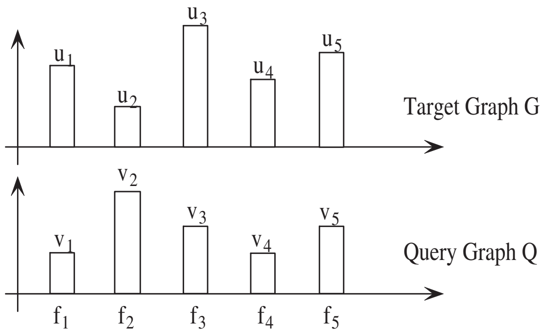


Figure 5.4. Frequency Difference

Eq. (5.4) calculates frequency difference of f_i between the query graph and the target graph,

$$r(u_i, v_i) = \begin{cases} 0, \; if \; u_i \geq v_i, \\ v_i - u_i, \; otherwise. \end{cases} \tag{5.4}$$

For the feature vectors shown in Figure 5.4, $r(u_1, v_1) = 0$; the extra embeddings from the target graph are not taken into account. The summed frequency difference of each feature in G and Q is written as $d(G, Q)$. Eq. (5.5) sums up all the frequency differences,

$$d(G, Q) = \sum_{i=1}^{n} r(u_i, v_i). \tag{5.5}$$

Suppose the query can be relaxed with k edges and the upper bound of allowed feature misses is then estimated using the greedy algorithm mentioned before. If $d(G, Q)$ is greater than that bound, it can be concluded that G does not contain Q within k edge relaxations. For this case, it is not necessary to perform any complicated structure comparison between G and Q. Since all the computations are done on the preprocessed information in the indices, the filtering process is fast.

3.4 Feature Set Selection

Though a bit counter-intuitive, using all the features together will not necessarily give the optimal solution; in some cases, it even deteriorates the performance rather than improving it. Given a query graph Q, let $F = \{f_1, f_2, \dots, f_m\}$ be the set of features included in Q, and d_F^k the maximal number of features missed in F after Q is relaxed (either relabeled or deleted) with k edges. Relabeling and deleting an edge e in Q have the same effect: the features containing e are broken. Let $\mathbf{u} = [u_1, u_2, \dots, u_m]^T$ and $\mathbf{v} = [v_1, v_2, \dots, v_m]^T$ be the feature vectors built from a target graph G in the graph database and a query graph Q based on a chosen feature set F. Let $\Gamma_F = \{G | d(G, Q) > d_F^k\}$, which is the set of graphs pruned from the database by the feature set F. It is obvious that, for any feature set F, the greater the cardinality of Γ_F, the better.

In general, a candidate graph G passing a filter should satisfy the following inequality,

$$r(u_1, v_1) + r(u_2, v_2) + \dots + r(u_n, v_n) \leq d_F^k. \tag{5.6}$$

Let P be the maximum common subgraph of G and Q. Vector $\mathbf{u}' = [u'_1, u'_2, \dots, u'_n]^T$ is its feature vector. If G contains Q within the relaxation ratio, P should contain Q within the relaxation ratio as well, i.e.,

$$r(u'_1, v_1) + r(u'_2, v_2) + \dots + r(u'_n, v_n) \leq d_F^k. \tag{5.7}$$

Since for any feature f_i, $u_i \geq u'_i$, we have

$$\begin{aligned} r(u_i, v_i) &\leq r(u'_i, v_i), \\ \sum_{i=1}^{n} r(u_i, v_i) &\leq \sum_{i=1}^{n} r(u'_i, v_i). \end{aligned}$$

Inequality (5.7) is stronger than Inequality (5.6). Assume that Inequality (5.7) does not hold for graph P, and there exists a feature f_i such that its frequency in P is too small to keep Inequality (5.7) true. However, Inequality (5.6) could still hold for graph G, if the misses of f_i is compensated by more occurrences of other features in G. This phenomenon is called *feature conjugation.* Feature conjugation likely takes place since the filtering does not distinguish the misses of individual features, but a collection of features. Due to feature conjugation, some graphs might not be pruned by the feature-based structural filtering method.

Definition 5.7 (Selectivity). *Given a graph database D, a query graph Q, and a feature f, the selectivity of f is defined by its average frequency difference within D and Q, written as $\delta_f(D, Q)$. $\delta_f(D, Q)$ is equal to the average of $r(u, v)$, where u is a variable denoting the frequency of f in a graph belonging to D, v is the frequency of f in Q, and r is defined in Eq. (5.4).*

There are three general feature set selection principles. The first principle is to select a large number of features. If only a small number of features are selected, the maximum allowed feature misses may become very close to $\sum_{i=1}^{n} v_i$. In that case, the filtering algorithm loses its pruning power. The second one is to make sure features cover the entire query graph. If most of the features cover several common edges, the relaxation of these edges will make the maximum allowed feature misses too big. The third one is to separate features with different selectivity. Low selective features deteriorate the potential filtering power from high selective ones due to frequency conjugation.

The above three criteria are not consistent with each other. For example, if all the features in a query graph are used, the second and the third principles will be violated since features often are concentrated in the center of a graph. On the other hand, one cannot use the most selective features alone because a query graph might not have enough highly selective features. The task of feature set selection is to make a trade-off among these principles. In practice, using a single filter with all the features included is not expected to perform well. Yan et al. [37] introduced a multi-filter strategy: Multiple filters are constructed and applied sequentially, where each filter uses a subset of features. This strategy was demonstrated to outperform a single filter based approach.

3.5 Structures with Gaps

The graph indexing methods introduced so far only consider connected subgraphs in a graph database. SAGA [31] proposes using fragments that do not always correspond to connected subgraphs and allows gaps in the indexing fragments.

The indexing unit in SAGA is a set of k nodes from the graphs in a database, where k is a user specified parameter, and is usually a small number. However, it could be expensive to enumerate all possible k-node sets in a large graph database. SAGA puts a limit on the diameter of each k-node set. If any pair of nodes in a k-node set are too far apart, this fragment does not correspond to a meaningful substructure, thus is not worth indexing. For a k-node set $\{v_1, v_2, \ldots, v_k\}$, if any two nodes v_i and v_j satisfy $d(v_i, v_j) \leq d_{max}$, where d_{max} is a diameter limit, SAGA connects the two nodes by a pseudo edge. Only those fragments that form a connected graph with the original edges or the newly introduced pseudo edges are indexed. Because of the pseudo edges, SAGA could index fragments with gaps.

The matching process of SAGA has three steps. The first step is to find small hits. In this step, the query graph is broken into small fragments and the graph index is probed to find database fragments that are similar to the query fragments. The second step is to assemble small hits retrieved in the first step to formulate larger matches. In this step, the small hits are first grouped by the database graph IDs and two neighbor hits are connected with each other to formulate a hit-compatible graph. This graph will tell which hits could be merged together to form a potential large match for the given query graph. The third step examines each candidate match and produces a set of real matches. SAGA allows users to specify a threshold to control the percentage of gap nodes in the subgraph match.

Different from Grafil [37] and SAGA [31], TALE [32] employs a new graph indexing method, called NH-Index (Neighborhood Index) for approximate subgraph matching of large query graphs efficiently. Instead of indexing various kinds of subgraphs in a graph database, NH-Index only considers the neighborhood structure of each node in a graph. Therefore, the number of indexing structures in NH-Index is equal to the number of nodes in the database, which is much smaller than the number of features used in many feature-based indexing methods. TALE also has an innovative matching paradigm for querying large graphs. Unlike the existing graph matching tools that treat every node in a graph equally, TALE distinguishes nodes by their importance in a graph structure. The algorithm first probes the NH-Index to match the important nodes in a query graph, and then progressively extends the matches by enclosing satisfiable nearby nodes of the matched nodes. TALE was applied to two real biological datasets and was able to produce meaningful results in both cases [32].

4. Reverse Substructure Search

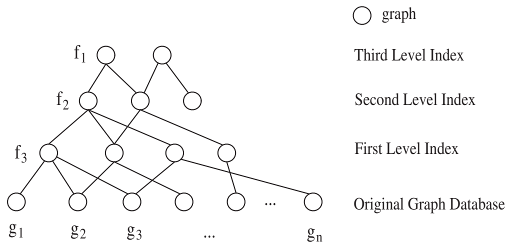
In contrast to substructure search (Definition 5.1) which finds all graphs that contain a query graph, reverse substructure search finds all graphs that are contained by a query graph. Reverse substructure search finds applications in chem-informatics, pattern recognition [11] (visual surveillance, face recognition), cyber security (virus signature detection [10]), information management (user-interest mapping [26]), etc. For example, in chemistry, a descriptor is a set of atoms with designated bonds that has certain properties of chemical reactions. Given a new molecule, identifying “descriptor" structures can help researchers to understand its possible properties. In computer vision, attributed relational graphs (ARG) [11] are used to model images by transforming them into spatial entities such as points, lines, and shapes. ARG also connects these spatial entities (nodes) together with their mutual relationships (edges) such as distances, using a graph representation. The graph models of basic objects such as humans, animals, cars, airplanes, are built first. A recognition system could then query these models to identify objects, or perform large-scale video search for specific models if the key frames of videos are represented by ARGs. Such a system can also be used to automatically recognize and classify objects in technical drawings.

Definition 5.8 (Reverse Substructure Search). *Given a graph database $\mathcal{D} = \{G_1, G_2, \ldots, G_n\}$ and a graph query Q, find all graphs G_i in $\mathcal{D}$, s.t., $Q \supseteq G_i$.*

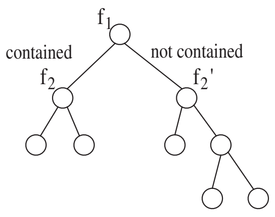
Reverse substructure search has its unique characteristics. The pruning strategy employed in substructure search has *inclusion logic*: Given a query graph Q and a database graph $G \in \mathcal{D}$, if a feature $f \subseteq Q$ and $f \nsubseteq G$, then $Q \nsubseteq G$. That is, if feature f is in Q then the graphs not having f are pruned. The inclusion logic prunes graphs using features contained in the query graph. On the contrary, reverse substructure search has an exclusion logic: If a feature $f \nsubseteq Q$ and $f \subseteq G$, then $Q \nsupseteq G$. That is, if feature f is not in Q then the graphs having f are pruned.

According to the exclusion logic, given a graph database D, the best indexing features are those subgraphs contained by lots of graphs in D, but unlikely contained by a query graph. This kind of subgraph features are called *contrast feature*s. There is a connection between contrast subgraphs and their frequency: Both infrequent and very frequent subgraphs are likely not contrastive, and thus not useful for indexing. Therefore, one can apply frequent graph pattern mining and select those contrast subgraphs. The number of contrast subgraphs could be huge; most of them are very similar to each other. Since the index performance is determined by a set of indexing features, rather than individual ones, it is important to find a set of contrast subgraphs that collectively perform well. Chen et al. [4] developed a redundancy-aware selection mechanism, cIndex, to sort out a set of distinctive contrast subgraphs that can maximize the pruning performance for a set of query graphs. cIndex has a flat index structure, where each feature is tested sequentially against queries. Based on cIndex, cIndex-BottomUp and cIndex-TopDown were developed to support hierarchical indexing models that could further improve the pruning capability.

The bottom-up hierarchical index builds indices layer by layer starting from the bottom-level original graphs in a database. Figure 5.5(a) shows a bottom-up hierarchical index where the i_{th}-level index $\mathcal{I}_i$ is built by applying cIndex to features in the $(i-1)_{th}$-level index $\mathcal{I}_{i-1}$. For example, the first-level index $\mathcal{I}_1$ is built on the original graph database by cIndex. Once this is done, the features in $\mathcal{I}_1$ can be regarded as another graph database, where cIndex can be executed again to form a second-level index $\mathcal{I}_2$. Following this manner, one can continue building higher-level indices until the pruning gain becomes zero. This method is called cIndex-BottomUp. Note that in a bottom-up index, features on the i_{th}-level must be subgraphs of features on the $(i-1)_{th}$-level. In Figure 5.5(a), subgraph relationships are shown as edges. For example, f_1 is a subgraph of f_2, which is in turn a subgraph of f_3. Given a query graph Q, if $f1 \nsubseteq Q$, then the tree covered by f_1 need not be examined due to the exclusion logic. Since the index on each level will save some isomorphism tests for the



(a) Bottom-up



(b) Top-down

Figure 5.5. cIndex

graphs it indexes, it is obvious that cIndex-BottomUp should outperform the flat index of cIndex.

The top-down hierarchical index first puts f_1, the feature with the highest pruning power, at the top of the hierarchy (Figure 5.5(b)). Given a query graph Q, if f_1 is contained by Q, f_2 is further tested against Q; if f_1 is not contained by Q, all the graphs indexed by f_1 are pruned, and then the second feature f_2' is tested for the remaining graphs. In a flat index built by cIndex, f_2 and f_2' are forced to be the same: No matter whether f_1 is contained by Q or not, the same second feature will be examined next. However, in a top-down index, they can be different. As shown in [4], cIndex-TopDown achieved the best performance due to its differentiating index structure.

5. Conclusions

Graph indexing is one of the emerging important tasks in graph database management and graph data mining. It is fundamental to many graph related applications, especially when an application involves large scale graph databases. In this chapter, we introduced the concepts of substructure search, approximate substructure search, and feature-based graph indexing methods that mine and index a compact set of discriminative and selective structure features for fast graph retrieval. These methods are going to significantly improve the performance of advanced graph applications such as graph classification and clustering.

References

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press/Addison-Wesley, 1999.

[2] S. Beretti, A. Bimbo, and E. Vicario. Efficient matching and indexing of graph models in content based retrieval. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 23:1089–1105, 2001.

[3] H. Bunke and G. Allermann. Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters*, 1(4):245–253, 1983.

[4] C. Chen, X. Yan, P. S. Yu, J. Han, D.-Q. Zhang, and X. Gu. Towards graph containment search and indexing. In *Proc. of 2007 Int. Conf. on Very Large Data Bases (VLDB'07)*, pages 926 – 937, 2007.

[5] Q. Chen, A. Lim, and K. W. Ong. D(k)-Index: An adaptive structural summary for graph-structured data. In *Proc. of 2003 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'03)*, pages 134–144, 2003.

[6] J. Cheng, Y. Ke, W. Ng, and A. Lu. FG-Index: Towards verification-free query processing on graph databases. In *Proc. of 2007 ACM Int. Conf. on Management of Data (SIGMOD'07)*, pages 857 – 872, 2007.

[7] C. Chung, J. Min, and K. Shim. APEX: An adaptive path index for xml data. In *Proc. of 2002 ACM Int. Conf. on Management of Data (SIGMOD'02)*, pages 121–132, 2002.

[8] S. Cook. The complexity of theorem-proving procedures. In *Proc. of the 3rd ACM Symp. on Theory of Computing (STOC'71)*, pages 151–158, 1971.

[9] B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proc. of 2001 Int. Conf. on Very Large Data Bases (VLDB'01)*, pages 341–350, 2001.

[10] Y. Fang, , R. Katz, and T. Lakshman. Gigabit rate packet pattern-matching using TCAM. In *Proc. of the 12th IEEE Int. Conf. on Network Protocols (ICNP'04)*, pages 174–183, 2004.

[11] K. Fu. A step towards unification of syntactic and statistical pattern recognition. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 8(3):398–404, 1986.

[12] R. Giugno and D. Shasha. GraphGrep: A fast and universal method for querying graphs. pages 112–115, 2002.

[13] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. of 1997 Int. Conf. on Very Large Data Bases (VLDB'97)*, pages 436–445, 1997.

[14] T. Hagadone. Molecular substructure similarity searching: Efficient retrieval in two-dimensional structure databases. *J. Chem. Inf. Comput. Sci.*, 32:515–521, 1992.

[15] H. He and A. Singh. Closure-Tree: An index structure for graph queries. In *Proc. of 2006 Int. Conf. on Data Engineering (ICDE'06)*, 2006.

[16] D. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing, MA, 1997.

[17] L. Holder, D. Cook, and S. Djoko. Substructure discovery in the subdue system. In *Proc. of AAAI'94 Workshop on Knowledge Discovery in Databases (KDD'94)*, pages 169–180, 1994.

[18] C. James, D. Weininger, and J. Delany. *Daylight Theory Manual Version 4.82*. Daylight Chemical Information Systems, Inc, 2003.

[19] H. Jiang, H. Wang, P. Yu, and S. Zhou. GString: A novel approach for efficient search in graph databases. In *Proc. of 2007 Int. Conf. on Data Engineering (ICDE'07)*, pages 566–575, 2007.

[20] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *Proc. of 2002 Int. Conf. on Data Engineering (ICDE'02)*, pages 129–140, 2002.

[21] T. Madej, J. Gibrat, and S. Bryant. Threading a database of protein cores. *Proteins*, 3-2:289–306, 1995.

[22] B. Messmer and H. Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 20:493–504, 1998.

[23] T. Milo and D. Suciu. Index structures for path expressions. *Lecture Notes in Computer Science*, 1540:277–295, 1999.

[24] N. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, Palo Alto, CA, 1980.

[25] E. Petrakis and C. Faloutsos. Similarity searching in medical image databases. *Knowledge and Data Engineering*, 9(3):435–447, 1997.

[26] M. Petrovic, H. Liu, and H. Jacobsen. G-ToPSS: Fast filtering of graph-based metadata. In *Proc. of 2005 Int. Conf. on World Wide Web (WWW'05)*, pages 539–547, 2005.

[27] J. Raymond, E. Gardiner, and P. Willett. Rascal: Calculation of graph similarity using maximum common edge subgraphs. *The Computer Journal*, 45:631–644, 2002.

[28] D. Shasha, J. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proc. of the 21th ACM Symp. on Principles of Database Systems (PODS'02)*, pages 39–52, 2002.

[29] A. Shokoufandeh, S. Dickinson, K. Siddiqi, and S. Zucker. Indexing using a spectral encoding of topological structure. In *Proc. of IEEE Int. Conf. on Computer Vision and Pattern Recognition (CVPR'99)*, pages 2491–2497, 1999.

[30] S. Srinivasa and S. Kumar. A platform based on the multi-dimensional data model for analysis of bio-molecular structures. In *Proc. of 2003 Int. Conf. Very Large Data Bases (VLDB'03)*, pages 975–986, 2003.

[31] Y. Tian, R. McEachin, C. Santos, D. States, and J. Patel. SAGA: A subgraph matching tool for biological graphs. *Bioinformatics*, 23:232–239, 2007.

[32] Y. Tian and J. Patel. TALE: A tool for approximate large graph matching. *Proc. of 2008 Int. Conf. on Data Engineering (ICDE'08)*, pages 963–972, 2008.

[33] P. Willett, J. Barnard, and G. Downs. Chemical similarity searching. *J. Chem. Inf. Comput. Sci.*, 38:983–996, 1998.

[34] D. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *Proc. of 2007 Int. Conf. on Data Engineering (ICDE'07)*, pages 976–985, 2007.

[35] H. Wolfson and I. Rigoutsos. Geometric hashing: An introduction. *IEEE Computational Science and Engineering*, 4:10–21, 1997.

[36] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *Proc. of 2004 ACM-SIGMOD Int. Conf. on Management of Data (SIGMOD'04)*, pages 335–346, 2004.

[37] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *Proc. of 2005 ACM-SIGMOD Int. Conf. on Management of Data (SIGMOD'05)*, pages 766 – 777, 2005.

[38] P. Zhao, J. Yu, and P. Yu. Graph indexing: tree + delta $>=$ graph. In *Proc. of 2007 Int. Conf. on Very Large Data Bases (VLDB'07)*, pages 938–949, 2007.

[39] L. Zou, L. Chen, J. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *Proc. of the 11th Int. Conf. on Extending Database Technology (EDBT'08)*, pages 181–192, 2008.