Graph Indexing: A Frequent Structure-based Approach[*]

Xifeng Yan[†]   Philip S. Yu[‡]   Jiawei Han[†]

[†]Department of Computer Science
University of Illinois at Urbana-Champaign
{xyan, hanj}@cs.uiuc.edu
[‡]IBM T. J. Watson Research Center
psyu@us.ibm.com

ABSTRACT

Graph has become increasingly important in modelling complicated structures and schemaless data such as proteins, chemical compounds, and XML documents. Given a *graph query*, it is desirable to retrieve graphs quickly from a large database via *graph-based indices*. In this paper, we investigate the issues of indexing graphs and propose a novel solution by applying a graph mining technique. Different from the existing *path-based methods*, our approach, called *gIndex*, makes use of *frequent substructure* as the basic indexing feature. Frequent substructures are ideal candidates since they explore the intrinsic characteristics of the data and are relatively stable to database updates. To reduce the size of index structure, two techniques, *size-increasing support constraint* and *discriminative fragments*, are introduced. Our performance study shows that gIndex has 10 times smaller index size, but achieves 3–10 times better performance in comparison with a typical path-based method, *GraphGrep*. The gIndex approach not only provides an elegant solution to the graph indexing problem, but also demonstrates how database indexing and query processing can benefit from data mining, especially frequent pattern mining. Furthermore, the concepts developed here can be applied to indexing sequences, trees, and other complicated structures as well.

1. INTRODUCTION

Graphs have become increasingly important in modelling complicated structures and schemaless data such as proteins, circuits, images, Web, and XML documents. Conceptually, any kind of data can be represented by graphs. Besides the prevalent use of XML in Web documents, we also witness the wide usage of graph databases in various domains. For example, in computer vision, graphs are used to represent complex relationships, such as the organization of entities in images. These relationships can be used to identify objects and scenes. Efficient retrieval of graph-based models is an essential problem in pattern recognition, as indicated by the wealth of research literature. In chemical informatics and bio-informatics, scientists use graphs to represent compounds and proteins. Daylight system [8], a commercial product for compound registration, has already been used in chemical informatics. Benefiting from such a system, researchers are able to do screening, designing, and knowledge discovery from compound or molecular databases.

In the core of many graph-related applications, lies a common and critical problem: *how to efficiently process graph queries and retrieve related graphs.* In some cases, the success of an application directly relies on the efficiency of the query processing system. The classical graph query problem can be described as follows: *Given a graph database $D = \{g_1, g_2, \ldots, g_n\}$ and a graph query q, find all the graphs in which q is a subgraph.* It is inefficient to perform a sequential scan on the graph database and check whether q is a subgraph of g_i. Sequential scan is very costly because one has to not only access the whole graph database but also check subgraph isomorphism which is NP-complete.

Clearly, it is necessary to build graph indices in order to help processing graph queries. XML query is a simple kind of graph query, which is usually built around path expressions. Various indexing methods [6, 12, 5, 9, 4, 14, 3] 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 are proposed in [14, 8]. Since all of these methods take *path* as the basic indexing unit, we categorize them as *path-based indexing approach*. In this paper, GraphGrep is taken as an example of path-based indexing since it represents the state of the art technique for graph indexing. Its general idea is as follows: enumerate all the existing paths in a database up to $maxL$ length and index them, 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 every graph g_i that contains all the paths (up to $maxL$ length) in query q.

The path-based approach has two advantages:

1. Paths are easier to manipulate than trees and graphs.

2. The index space is predefined: all the paths up to $maxL$ length are selected.

In order to answer tree- or graph- structured queries, a path-based approach has to break them into paths, search each path separately for the graphs containing the path, and join the results. Since the structural information could be lost when breaking such queries apart, it is likely that many false positive answers will be returned. Thus, a path-based approach is not suitable for complex graph queries. The advantages mentioned above of path-based indexing now become its weak points for indexing graphs:

1. Path is too simple: structural information is lost.

2. There are too many paths: the set of paths in a graph database usually is huge.

The following example illustrates the disadvantages of path-based approaches. Figure 1 is a sample chemical dataset extracted from an AIDS antiviral screening database [1]. For simplicity, we ignore the bond type.
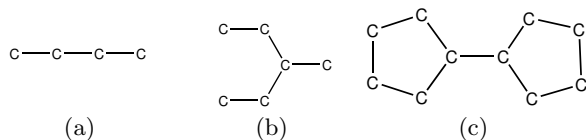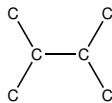

Figure 1: A Sample Database

Figure 2: A Sample Query

Figure 2 shows a sample query, 2,3-dimethylbutane. Assume that this query is posed to the sample database. Although only graph (c) in Figure 1 is the answer, graphs (a) and (b) cannot be pruned since both of them contain all the paths existing in the query graph: c, $c - c$, $c - c - c$, and $c - c - c - c$. In this case, carbon chains (up to length 4) are not discriminative enough to tell the difference in the sample graphs. This indicates that path may not be a good structure to serve as the index feature for graph databases.

As another problem, a graph database may contain too many paths if its graphs are large and diverse. For example, by randomly extracting 10,000 graphs from the antiviral screening database, we find that there are totally around 100,000 paths with length up to 10. Most of them are redundant based on our observation. It is inefficient to index all of them.

The above analysis motivates us to search for an alternative solution. *"Can we use graph structure instead of path as the basic index feature?"* This study provides a firm answer to this question. It shows that a graph-based index can significantly improve query performance over a path-based one. Certainly, the number of graph structures is usually much larger than the number of paths in a graph database. To overcome this difficulty, only *frequent subgraphs* (i.e., frequent substructures) with length up to $maxL$ are retained

[1] http://dtp.nci.nih.gov/docs/aids/aids_data.html.

for indexing; whereas the frequent subgraphs can be generated by existing frequent graph mining algorithms efficiently [7, 10, 20, 17, 2, 21].

In order to avoid the exponential growth of the number of frequent subgraphs, the support threshold is progressively increased when the subgraphs grow large. That is, we use low support for small subgraphs and high support for large subgraphs. Meanwhile, the concept of discriminative structure is introduced to reduce the redundancy among the frequent subgraphs selected as index features. These ideas lead to the development of our new algorithm, gIndex. gIndex can scale down the number of indexing features in the above example on the AIDS antiviral screening database to 3,000, but improve query response time by 3 to 10 times on average. gIndex also explores novel concepts to improve query search time, including using the Apriori pruning and maximum discriminative structures to reduce the number of subgraphs to be examined for index access and query processing.

Frequent subgraphs are ideal candidates for indexing since they are relatively stable to database updates, thereby making incremental maintenance of index affordable. They also provide an efficient solution on index construction: we can first mine discriminative structures from a small portion of a large database, and then build the complete index based on these structures by scanning the whole database **once**.

In this paper, the issues of feature selection, index search, index construction, and incremental maintenance are thoroughly explored. The contribution of this study is not only at providing a novel and efficient solution to graph indexing, but also at the demonstration of how data mining technology may help solving indexing and query processing problems. This may inspire us to further explore the application of data mining in query processing and data management. Furthermore, the concepts developed here can also be applied to indexing sequences, trees, and other complex structures.

The remaining of the paper is organized as follows. Section 2 defines the preliminary concepts and briefly analyzes the graph query processing problem. Section 3 introduces frequent fragment and the size-increasing support constraint. Discriminative fragment is introduced in Section 4. Section 5 formulates the algorithm and presents the index construction and incremental maintenance processes. Our performance study is reported in Section 6. Related work is discussed in Section 7, and Section 8 summarizes our study.

2. PRELIMINARIES

As a general data structure, labeled graph is used to model complicated structures and schemaless data. In labeled graph, vertex and edge represent entity and relationship, respectively. The attributes associated with entities and relationships are called *label*s. XML is a kind of directed labeled graph. The chemical compounds shown in Figure 1 are undirected labeled graphs. In this paper, we investigate indexing techniques for *undirected labeled graphs*. It is straightforward to extend our method to process other kinds of graphs.

As a notational convention, the *vertex set* of a graph g is denoted by $V(g)$, the *edge set* by $E(g)$, and the *size* of a graph by $len(g)$, which is defined by $|E(g)|$ in this paper. A label function, l, maps a vertex or an edge to a label. A graph g is a subgraph of another graph g' if there exists a subgraph isomorphism from g to g', denoted by $g \subseteq g'$. g' is called a super-graph of g.

DEFINITION 1 (SUBGRAPH ISOMORPHISM). *A subgraph isomorphism is an injective function $f : V(g) \to V(g')$, such that (1) $\forall u \in V(g)$, $l(u) = l'(f(u))$, and (2) $\forall (u, v) \in E(g)$, $(f(u), f(v)) \in E(g')$ and $l(u, v) = l'(f(u), f(v))$, where l and l' are the label function of g and g', respectively. f is called an embedding of g in g'.*

DEFINITION 2 (GRAPH QUERY PROCESSING). *Given a graph database $D = \{g_1, g_2, \ldots, g_n\}$ and a query graph q, it returns the query answer set $D_q = \{g_i | q \subseteq g_i, g_i \in D\}$.*

EXAMPLE 1. *Figure 1 shows a sample labeled graph dataset. This dataset will be used as our running example. For the query q shown in Figure 2, the query answer set, D_q, has only one element: graph (c) in Figure 1.*

In general, graph query can be any kind of SQL statements applied to graphs. Besides the topological condition, one may also use other conditions to perform indexing. In this paper, we focus on indexing graphs only based on their topology.

The processing of graph queries can be divided into two major steps:

1. *Index construction,* which is a preprocessing step, performed by enumerating and selecting features in graph database D. The *graph feature set* is denoted by F [2]. For any graph feature $f \in F$, D_f is the set of graphs containing f, $D_f = \{g_i | f \subseteq g_i, g_i \in D\}$.

2. *Query processing,* which consists of two 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 q's features in the feature set. Therefore, D_q is a subset of C_q. (2) *Verification,* which checks graph g in C_q to verify whether q is really a subgraph of g.

Cost Analysis. In graph query processing, the major concern is Query Response Time:

$$T_{search} + |C_q| * T_{iso\_test}, \qquad (1)$$

where T_{search} is the time spent in the search step and T_{iso_test} is the average time of subgraph isomorphism testing, which is conducted over query q and graphs in C_q. In the verification step, it takes $|C_q| * T_{iso\_test}$ to prune false positives in C_q. Usually the verification time dominates Eq. (1) since the computational complexity of T_{iso_test} is NP-complete. Approximately, the value of T_{iso_test} does not change too much for a given query. Thus, the key to improve query response time is to minimize the size of the candidate answer set, $|C_q|$. If a graph database is very large such that the index cannot be held in the memory, T_{search} may be critical for the query response time.

We are also interested in minimizing the index size M, which is approximately proportional to the size of the feature set $|F|$:

$$M \propto |F| \qquad (2)$$

---

[2]A graph without any vertex and edge is denoted by $f_\varnothing$, $f_\varnothing$ is viewed as a special feature, which is a subgraph of any graph. For completeness, F must include $f_\varnothing$.

Thus, in order to reduce the index size, it is important to maintain a compact feature set. Otherwise, if the index is too large to reside in the memory, the cost of accessing F may be even greater than that of accessing the graph database itself. In the next section, we will begin our examination of minimizing $|C_q|$ and $|F|$.

3. FREQUENT FRAGMENT

Given a graph database D, $|D_g|$ is the number of graphs in D where g is a subgraph. $|D_g|$ is called *(absolute) support,* denoted by $support(g)$. A graph g is *frequent* if its support is no less than a minimum support threshold, $minSup$. As one can see, frequent graph is a relative concept. Whether a graph is frequent depends on the setting of $minSup$. We use the term "*fragment*" to refer to a small subgraph (i.e., substructure) existing in graph databases and query graphs. Figure 3 shows two frequent fragments in the sample database with $minSup = 2$.
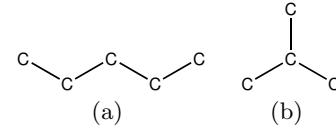

Figure 3: Frequent Fragments

Frequent fragments expose the intrinsic characteristic of a graph database. Suppose all the frequent fragments with minimum support $minSup$ are indexed. Given a query graph q, if q is frequent, the graphs containing q can be retrieved directly since q is indexed. Otherwise, q probably has a frequent subgraph f whose support may be close to $minSup$. Since any graph with q embedded must contain q's subgraphs, D_f is a candidate answer set of query q. If $minSup$ is low, it is not expensive to verify the small number of graphs in D_f in order to find the query answer set. Therefore, it is feasible to index frequent fragments for graph query processing.

A further examination helps clarify the case where query q is not frequent in the graph database. 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 $support(f_i) \geq minSup$ and $support(f_{i+1}) < minSup$. Since all the frequent fragments with minimum support $minSup$ are indexed, the graphs containing f_j ($1 \leq j \leq i$) are known. Therefore, we can compute the candidate answer set C_q by $\bigcap_{1 \leq j \leq i} D_{f_j}$, whose size is at most $support(f_i)$. For many queries, $support(f_i)$ is likely to be close to $minSup$. Hence the intersection of its frequent fragments, $\bigcap_{1 \leq j \leq i} D_{f_j}$, leads to a small size of C_q. Therefore, the cost of verifying C_q is minimal when $minSup$ is low. This is confirmed by our experiments in Section 6.

The above discussion exposes our key idea in graph indexing: *It is feasible to construct high-quality indices using only frequent fragments.* However, 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 $minSup$. If $minSup$ is set too high, the size of C_q may be too large. If $minSup$ is set too low, it is too difficult to generate all the frequent fragments because there may exist an exponential number of frequent fragments under low support.

Should we enforce a uniform *minSup* for all the fragments? Let's examine a simple example: a *completely connected* graph with 10 vertices, each of which has a distinct label. There are 45 1-edge subgraphs, 360 2-edge ones, and more than $1,814,400$ 8-edge ones[3]. As one can see, in order to reduce the overall index size, it is appropriate for the index scheme to have *low* minimum support on *small* fragments (for effectiveness) and *high* minimum support on *large* fragments (for compactness). This criterion on the selection of frequent fragments for effective indexing is called *size-increasing support constraint.*

DEFINITION 3 (SIZE-INCREASING SUPPORT). *Given a monotonically nondecreasing function, $\psi(l)$, pattern g is frequent under the size-increasing support constraint if and only if $support(g) \geq \psi(len(g))$, and $\psi(l)$ is a size-increasing support function.*

By enforcing the size-increasing support constraint, we bias the feature selection to small fragments with low minimum support and large fragments with high minimum support. Especially, we always choose the (absolute) *minSup* to be 1 for size-0 fragment to ensure the completeness of the indexing. This method leads to two advantages: (1) the number of frequent fragments so obtained is much less than that with the lowest uniform *minSup*, and (2) low-support large fragments may be indexed well by their smaller subgraphs; thereby we do not miss useful fragments for indexing.
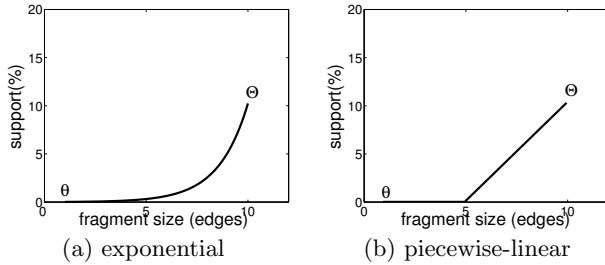

Figure 4: Size-increasing Support Functions

EXAMPLE 2. *Figure 4 shows two size-increasing support functions:* exponential *and* piecewise-linear. *We select size-1 fragments with minimum support θ and larger fragments with higher support until we exhaust fragments up to size of maxL with minimum support Θ. A typical setting of θ and Θ is 1 and $0.1N$, respectively, where N is the size of the database. We have a wide range of monotonically nondecreasing functions to use as $\psi(l)$.*

By using frequent fragments with the size-increasing support constraint, we have a smaller number of fragments to index. However, the number of indexed fragments may still be huge when the support is low. For example, 1,000 graphs may easily produce 100,000 fragments of that kind. It is both time and space consuming to index them. In the next section, we design a distillation procedure to acquire the best fragments, i.e., *discriminative fragments*, from the frequent fragments. In the end, only the most useful fragments are retained as indexing features.

---

[3]For any n-vertex complete graph with different vertex labels, the number of size-k connected subgraphs is greater than $C_n^{k+1} \times (k+1)!/2$, which is the number of size-k paths ($k < n$).

4. DISCRIMINATIVE FRAGMENT

Do we need to index every frequent fragment? Let's have some analysis. If two similar frequent fragments, f_1 and f_2, are contained by the same set of graphs in the database, i.e., $D_{f_1} = D_{f_2}$, it is probably wise to include only one of them in the feature set. Generally speaking, among similar fragments with the same support, it is often sufficient to index only the *smallest common fragment* since more query graphs may contain the smallest fragment. 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. Thus f' should be removed from the feature set. In this case, we say f' is not more *discriminative* than f. Note that this is contrary to the *closed graph* concept introduced in [21], which is to reduce the number of frequent subgraphs generated in graph mining, where the maximum fragments are retained.

EXAMPLE 3. *All the graphs in the sample database (Figure 1) contain carbon-chains: c, c−c, c−c−c, and c−c−c−c. Fragments c−c, c−c−c, and c−c−c−c do not provide more indexing power than fragment c. Thus, they are useless for indexing.*

So far, we considered only the discriminative power between a fragment and one of its subgraphs. This concept can be further extended to the combination of its subgraphs.

DEFINITION 4 (REDUNDANT FRAGMENT). *Fragment x is redundant with respect to feature set F if $D_x \approx \bigcap_{f \in F \wedge f \subseteq x} D_f$.*

Each graph in set $\bigcap_{f \in F \wedge f \subseteq x} D_f$ 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 the presence of fragment x in a graph can be predicted well by the presence of its subgraphs. Thus, fragment x should not be used as an indexing feature since it does not provide any benefit to pruning if its subgraphs are already being used as indexing features. In such case, x is a redundant fragment. In contrast, there are fragments which are not redundant, called *discriminative fragments*.

DEFINITION 5 (DISCRIMINATIVE FRAGMENT). *Fragment x is discriminative with respect to F if $D_x \ll \bigcap_{f \in F \wedge f \subseteq x} D_f$.*

EXAMPLE 4. *Let us examine the query example in Figure 2. As shown in Example 3, carbon chains, $c − c$, $c − c − c$, and $c − c − c − c$, are redundant and should not be used as indexing features in this dataset. The carbon ring (Figure 5 (c)) is a discriminative fragment since only graph (c) in Figure 1 contains it while graphs (b) and (c) in Figure 1 have all of its subgraphs. Fragments (a) and (b) in Figure 5 are discriminative too.*

Since D_x is always a subset of $\bigcap_{f \in F \wedge f \subseteq x} D_f$, x should be either redundant or discriminative. Obviously, redundant fragment is a relative concept. We provide a simple measure on the degree of redundancy. Let fragments $f_1, f_2, \ldots, f_n$ be indexing features. Given a new fragment x, the discriminative power of x can be measured by

$$Pr(x|f_{\varphi_1}, \ldots, f_{\varphi_m}), f_{\varphi_i} \subseteq x, 1 \leq \varphi_i \leq n. \quad (3)$$

Eq. (3) shows the presence probability of x given the presence of $f_{\varphi_1}, \ldots, f_{\varphi_m}$ in a graph. We denote $1/Pr(x|f_{\varphi_1}, \ldots,$

f_{φ_m}) by γ, called *discriminative ratio*. γ has the following properties:

1. $\gamma \geq 1$.

2. when $\gamma = 1$, fragment x is completely redundant since the graphs indexed by this fragment can be fully indexed by the combination of fragment f_{φ_i}.

3. when $\gamma \gg 1$, fragment x is more discriminative than the combination of fragments f_{φ_i}. Thus, x becomes a good candidate to index.

4. γ is related to the fragments which are already in the feature set.

The discriminative ratio can be calculated by the following formula:

$$\gamma = \frac{|\bigcap_i D_{f_{\varphi_i}}|}{|D_x|}, \qquad (4)$$

where D_x is the set of graphs containing x and $\bigcap_i D_{f_{\varphi_i}}$ is the set of graphs which contain the subgraphs of x in the feature set.

In order to mine discriminative fragments, we may set a minimum discriminative ratio γ_{min} and retain any fragment whose discriminative ratio is no less than γ_{min}. We shall generate discriminative fragments from small size to large size.
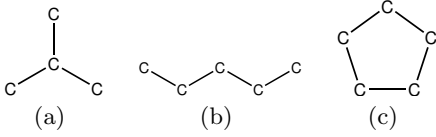

Figure 5: Discriminative Fragments

EXAMPLE 5. *Suppose we set $\psi(l) \equiv 1$ and $\gamma_{min} = 1.5$ for the sample dataset in Figure 1. Figure 5 lists three of discriminative fragments (usually, we shall also add $f_\varnothing$, a fragment without any vertex and edge, into the feature set as the initial fragment). There are other discriminative fragments in this sample dataset. The discriminative ratio of fragments (a), (b), and (c) is 1.5, 1.5, and 2.0, respectively. The discriminative ratio of fragment (c) in Figure 5 can be computed as follows: suppose fragments (a) and (b) have already been selected as index features. There are "two" graphs in the sample dataset containing fragment (b) and "one" graph containing fragment (c). Since fragment (b) is a subgraph of fragment (c), the discriminative ratio of fragment (c) is $2/1 = 2.0$.*

5. GINDEX

In this section, we present the gIndex algorithm, examine the data structures storing the index, and discuss the incremental maintenance of index that supports insertion and deletion operations. We illustrate the design and implementation of gIndex in five subsections: (1) discriminative fragment selection, (2) index construction, (3) search, (4) verification, and (5) incremental maintenance.

---

[4]$support(x) \geq \psi(len(x))$ and $|\bigcap_i D_{f_{\varphi_i}}|/|D_x| \geq \gamma_{min}$, for $f_{\varphi_i} \subseteq x$.

Algorithm 1 featureSelection

Input: Graph database D, Discriminative ratio γ_{min},
        Size-increasing support function $\psi(l)$,
        Maximum fragment size $maxL$.
Output: Feature set F.

1: let $F = \{f_\varnothing\}$, $D_{f_\varnothing} = D$, and $l = 0$;
2: **while** $l \leq maxL$ **do**
3:     **for each** fragment x, whose size is l **do**
4:         **if** x is frequent and discriminative[4] **then**
5:             $F = F \cup \{x\}$;
6:     $l = l + 1$;
7: **return** F;

5.1 Discriminative Fragment Selection

Applying the concepts introduced in Sections 3 and 4, gIndex first generates all frequent fragments with the size-increasing support constraint. Meanwhile, it distills these fragments to eliminate the redundant ones. This feature selection process proceeds in a level-wise manner, i.e., Breadth-First Search (BFS). Algorithm 1 outlines the pseudo-code of feature selection.

5.2 Index Construction

Once discriminative fragments are selected, gIndex has efficient data structures to store and retrieve them. It translates fragments into sequences and holds them in a prefix tree. Each fragment is associated with an id list: the ids of graphs containing this fragment. We present the details of index construction in this section.

5.2.1 Graph Sequentialization

Substantial portion of computation involved in index construction and searching is related to *graph isomorphism checking*. One has to quickly retrieve a given fragment from the index. Considering that graph isomorphism testing is hard (It is suspected to be in neither P nor NP-complete, though it is obviously in NP); it is inefficient to scan the whole feature set to match fragments one by one. An efficient solution is to translate a graph into a sequence, called *canonical label*. If two fragments are the same, they must share the same canonical label.
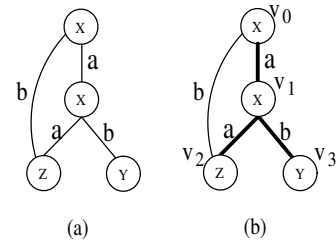

Figure 6: DFS Code Generation

A traditional sequentialization method is to concatenate rows or columns of the adjacency matrix of a graph into an integer sequence. Since most graphs in real applications are sparse graphs, the traditional sequentialization method may not work efficiently. There are too many useless 0's in the

integer sequence. Furthermore, it is not space efficient to store adjacency matrices. A novel graph sequentialization method, called *DFS coding*, was introduced in [20, 21].

DFS coding can translate a graph into a unique edge sequence, which is generated by performing a depth first search (DFS) in a graph. The bold edges in Figure 6(b) constitute a DFS search tree. Each vertex is subscripted by its discovery time in a DFS search. The *forward edge* set contains all the edges in the DFS tree while the *backward edge* set contains the remaining edges. For the graph shown in Figure 6(b), the forward edges are discovered in the order of $(v_0, v_1), (v_1, v_2), (v_1, v_3)$. Now we put backward edges into the order as follows. Given a vertex v, all of its backward edges should appear after the forward edge pointing to v. For vertex v_2 in Figure 6(b), its backward edge (v_2, v_0) should appear after (v_1, v_2). Among the backward edges from the same vertex, we can enforce an order: given v_i and its two backward edges, $(v_i, v_j), (v_i, v_k)$, if $j < k$, then edge (v_i, v_j) will appear before edge (v_i, v_k). So far, we complete the ordering of the edges in a graph. Based on this order, a complete edge sequence for Figure 6(b) is formed: $\langle (v_0, v_1), (v_1, v_2), (v_2, v_0), (v_1, v_3) \rangle$. This sequence is called a DFS code.

We represent a labeled edge by a 5-tuple, $(i, j, l_i, l_{(i,j)}, l_j)$, where l_i and l_j are the labels of v_i and v_j respectively and $l_{(i,j)}$ is the label of the edge connecting v_i and v_j. Thus, the above edge sequence can be written $\langle$ (0, 1, X, a, X) (1, 2, X, a, Z) (2, 0, Z, b, X) (1, 3, X, b, Y) $\rangle$. Since each graph can have many different DFS search trees and each of them has a DFS code, a lexicographic order is designed in [20, 21] to order the DFS codes. For any graph g, the minimum DFS code is chosen among g's DFS codes as its canonical label, denoted by $dfs(g)$. In the next subsections, we will introduce how to store and search the minimum DFS codes of discriminative fragments.

5.2.2 gIndex Tree

Using the above sequentialization method, each fragment can be mapped to an edge sequence (e.g., DFS code). We insert the edge sequences of discriminative fragments in a prefix tree, called *gIndex Tree*.
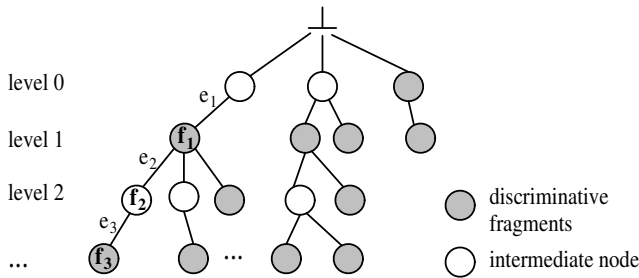

Figure 7: gIndex Tree

EXAMPLE 6. *Figure 7 shows a gIndex tree, where each node represents a fragment (a DFS code). For example, two discriminative fragments $f_1 = \langle e_1 \rangle$ and $f_3 = \langle e_1\ e_2\ e_3 \rangle$ are stored in the gIndex tree (for brevity, we use e_i to represent edges in the DFS codes). Although fragment $f_2 = \langle e_1\ e_2 \rangle$ is not a discriminative fragment, we have to store f_2 in order to connect fragments f_1 and f_3.*

The gIndex tree records all size-n discriminative fragments in level n (size-0 fragments are graphs with only one vertex and no edge; the root node in the tree is $f_\varnothing$). In this tree, code s is an ancestor of s' if and only if s is a prefix of s'. We use black nodes to denote discriminative fragments. White nodes (redundant fragments) are intermediate nodes which connect the whole gIndex tree. All leaf nodes are discriminative fragments since it is useless to store redundant fragments in leaf nodes. In each black node f_i, an id list (I_i), the ids of graphs containing f_i, is recorded. White nodes do not have any id list. Assume we want to retrieve graphs which contain both fragments f_i and f_j, what we need to do is to intersect I_i and I_j.

gIndex tree has two advantages over other index structures such as B+ tree. First, gIndex tree records not only discriminative fragments, but also some redundant fragments. This setting makes the Apriori pruning possible (Section 5.3.1). Secondly, gIndex tree can reduce the number of intersection operations conducted on id lists of discriminative fragments by using (approximate) maximum fragments only (Section 5.3.2). In short, the search time T_{search} will be significantly reduced by using gIndex tree.

5.2.3 Remark on gIndex Tree Size

Upon examining the size of the gIndex tree, we find that the graph id lists associated with black nodes fill the major part of the tree. We may derive a bound for the number of black nodes on any path from the root to a leaf node. In the following discussion, we do not count the root as a black node.

Let the discriminative fragments on a path be $f_0, f_1, \ldots, f_{k-1}$, where $f_i \subset f_{i+1}$, $0 \leq i \leq k - 2$. According to the definition of discriminative fragments, $|\bigcap_j D_{f_j}|/|D_{f_i}| \geq \gamma_{min}$, where $0 \leq j < i$. Hence $|D_{f_0}| \geq \gamma_{min}|D_{f_1}| \geq \ldots \geq \gamma_{min}^{k-1}|D_{f_{k-1}}|$. Since $|D_{f_0}| \leq N/\gamma_{min}$ and $|D_{f_{k-1}}| \geq 1$, we must have $k \leq \log_{\gamma_{min}} N$.

THEOREM 1. *For any path in the gIndex tree, the number of black nodes on the path is $O(\log_{\gamma_{min}} N)$, where N is the size of the graph database.*

Theorem 1 delivers the upper bound on the number of black nodes on any path from the root to a leaf node. Considering the size-increasing support constraint, we have

$$N/\gamma_{min}^k \geq |D_{f_{k-1}}| \geq \psi(l), \tag{5}$$

where l is the size of fragment f_{k-1} ($l \geq k - 1$).

EXAMPLE 7. *Suppose the size-increasing support function $\psi(l)$ is a linear function: $\frac{l}{maxL} \times 0.01N$, where $maxL = 10$. This means we index discriminative fragments whose size is up to 10. If we set γ_{min} to be 2, from Eq. 5, we know $\frac{1}{2^k} \geq \frac{k-1}{1000}$. It implies the maximum value of k, i.e., the number of black nodes on any path in the gIndex tree, is less than 8.*

Are there lots of graph ids recorded in the gIndex tree? For the number of ids recorded on any path from the root to a leaf node, the following bound is obtained:

$$\sum_{i=0}^{k-1} |D_{f_i}| \leq (\frac{1}{\gamma_{min}} + \frac{1}{\gamma_{min}^2} \ldots + \frac{1}{\gamma_{min}^k})N,$$

where $f_0, f_1, \ldots, f_{k-1}$ are discriminative fragments on the path. If $\gamma_{min} \geq 2$, $\sum_{i=0}^{k-1} |D_{f_i}| \leq N$. Otherwise, we have

more ids to record. In this case, it is space inefficient to record D_{f_i}. An alternative solution is to store the differential id list, i.e., $\triangle D_{f_i} = \bigcap_x D_{f_x} - D_{f_i}$, where $f_x \in F$ and $f_x \subset f_i$. Such a solution generalizes a similar idea which was presented in [22], but handles multiple rather than one id list. The scope of our study does not permit a further examination of the differential id list.

5.2.4 gIndex Tree Implementation

The gIndex tree is implemented using a hash table to help locating fragments and retrieving their id lists quickly; both black nodes and white nodes are included in the hash table. This is in lieu of a direct implementation of the tree structure. Nonetheless, the gIndex tree concept is crucial in determining the redundant (white) nodes which, as included in the index, will facilitate the pruning of search space.

With graph sequentialization, we can map any graph to an integer by hashing its canonical label.

DEFINITION 6 (GRAPHIC HASH CODE). *Given a sequence hash function h and a graph g, $h(dfs(g))$ is called* graphic hash code.

We treat the graphic hash code as the hash value of a graph. If two graphs g and g' are isomorphic, then $h(dfs(g)) = h(dfs(g'))$. Graphic hash code can help quickly locating fragments in the gIndex tree.

5.3 Search

Given a query q, gIndex enumerates all its fragments up to a maximum size and locates them in the index. Then it intersects the id lists associated with these fragments. Algorithm 2 outlines the pseudo-code of the search step.

Algorithm 2 Search

Input: Graph database D, Feature set F, Query q,
       and Maximum fragment size $maxL$.
Output: Candidate answer set C_q.

1: let $C_q = D$;
2: **for each** fragment $x \subseteq q$ and $len(x) \leq maxL$ **do**
3:     **if** $x \in F$ **then**
4:         $C_q = C_q \cap D_x$;
5: **return** C_q;

5.3.1 Apriori Pruning

The pseudo-code in Algorithm 2 must be optimized. It is inefficient to generate every fragment in the query graph first and then check whether it belongs to the index. Imagine how many fragments a size-10 complete graph may have. We shall apply the Apriori rule: if a fragment is not in the gIndex tree, we need not check its super-graphs any more. That is why we record some redundant fragments in the gIndex tree. Otherwise, if a fragment is not in the feature set, one cannot conclude that none of its super-graphs will be in the feature set.

A hash table H is used to facilitate the Apriori pruning. As explained in Section 5.2.4. It contains all the graphic hash codes of the nodes shown in the gIndex tree including intermediate nodes. Whenever we find a fragment in the query whose hash code does not appear in H, we need not check its super-graphs any more.

5.3.2 Maximum Discriminative Fragments

Operation $C_q = C_q \cap D_x$ is done by intersecting the id lists of C_q and D_x. We now consider how to reduce the number of intersection operations. Intuitively, if query q has two fragments, $f_x \subset f_y$, 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 fragments (or indexing features) contained in query q, i.e., $F(q) = \{f_x | f_x \subseteq q \wedge f_x \in F\}$. Let $F_m(q)$ be the set of fragments in $F(q)$ that are not contained by other fragments 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 fragments in $F_m(q)$ are called *maximum discriminative fragments*. In order to calculate C_q, we only need to perform intersection operations on the id lists of maximum discriminative fragments. Sometimes, it is expensive to compute $F_m(q)$ if the subgraph enumeration algorithm does not generate all super-graphs of each fragment. Thus, we may replace $F_m(q)$ with the *approximate maximum discriminative fragment set* $F'_m(q) = \{f_x | f_x \in F(q), \nexists f_y, s.t., f_y$ is f_x's descendant in the gIndex tree and $f_y \in F(q)\}$. $F'_m(q)$ includes the deepest black nodes that a query can reach in the gIndex tree, which is easy to compute.

5.3.3 Inner Support

The previous support definition is only counting the frequency of a fragment in a graph dataset. Actually, one fragment may appear several times even in one graph.

DEFINITION 7 (INNER SUPPORT). *Given a graph g, the inner support of subgraph x is the number of embeddings of x in g, denoted by $inner\_support(x, g)$.*

LEMMA 1. *If g is a subgraph of G and fragment $x \subset g$, then $inner\_support(x, g) \leq inner\_support(x, G)$.*

GraphGrep [14] uses the above lemma to improve the filtering power. In order to put the inner support to use, we have to store the inner support of discriminative fragments together with their graph id lists, which means the space cost is doubled. The pruning power of Lemma 1 is related with the size of queries. If a query graph is large, it is pretty efficient using inner support.

5.4 Verification

After getting the candidate answer set C_q, we have to verify whether the graphs in C_q really contain the query graph or not. The simplest way to do it is to perform a subgraph isomorphism test on each graph one by one. GraphGrep [14] proposed an alternative approach. It records all the embeddings of paths in the graph database. Rather than doing real subgraph isomorphism testing, it performs join operations on these embeddings to figure out the possible isomorphism mapping between the query graph and the graphs in C_q. Considering there are lots of paths in the index and each path may have tens of embeddings, we find that in some cases it even performs worse than the simplest approach. Thus, we only implement the simple one in our study.

5.5 Insert/Delete Maintenance

In this section, we present our index maintenance algorithm to handle insert/delete operations. For each insert or delete operation, we simply update the id lists of involved fragments as shown in Algorithm 3. Algorithm 3 is very efficient and the index quality may be still good if the statistics of old database and new database are similar. Here,

the statistics means the frequent graphs and their supports in a graph database. If they do not change, then the discriminative fragments will not change at all. Thus, we only need to update the id lists of those fragments in the index, just as Algorithm 3 does. Fortunately, frequent patterns are relatively stable to database updates. A small number of insert/delete operations will not change their distribution too much. This property becomes one key advantage of using frequent fragments in the index.

Algorithm 3 Insert/Delete

Input: Graph database D, Feature set F,
    Inserted (Deleted) graph g and its id gid,
    Maximum fragment size $maxL$.

1: **for each** fragment $x \subseteq g$ and $len(x) \leq maxL$ **do**
2:    **if** $x \in F$ **then**
3:     Insert:
     insert gid into the id list of x;
4:     Delete:
     delete gid from the id list of x;
5: **return**;

The incremental update property leads to another interesting result: *a single database scan algorithm for the index construction*. Rather than mining discriminative fragments from the whole graph database, one can actually first sample a small portion of the original database randomly, load it into the main memory, mine discriminative fragments from this small amount of data and then build the index by scanning the remaining database **once**. This strategy can significantly reduce the index construction time, especially when the database is large. As long as the sample data reflects the data distribution in the original database, the single scan algorithm works very well, which was confirmed in our experiments.

The quality of index may degrade over time after lots of insertions and deletions. Thus, we need a measure to monitor the quality of the discriminative fragments indexed which may be out-of-date after updates. The effectiveness of the gIndex can be measured by $\frac{|\bigcap_f D_f|}{|D_x|}$, where $f \in F, f \subseteq x$, over some set of randomly selected query graphs. This is the ratio of the candidate answer set size over the actual answer set size. We monitor the measure based on sampled queries and check whether its average value changes much over time. A sizable increase of the value implies that the effectiveness of the index has deteriorated, probably because some discriminative fragments are missing from the indexing features. In this case, we have to consider recomputing the index from scratch.

6. EXPERIMENTAL RESULT

In this section, we report our experiments that validate the effectiveness and efficiency of the gIndex algorithm. The performance of gIndex is compared with that of GraphGrep, a path-based approach [14]. Our experiments demonstrate that:

1. The index size of gIndex is more than 10 times smaller than that of GraphGrep;

2. gIndex outperforms GraphGrep by 3 to 10 times in various query loads; and

3. the index returned by the incremental maintenance algorithm is effective: it performs as well as the index computed from scratch provided the data distribution does not change much.

We use two kinds of datasets in our experiments: one real dataset and a series of synthetic datasets (we ignore the edge labels). Most of our experiments have been performed on the real dataset since it is the source of real demand.

1. The real dataset we tested is that of an AIDS antiviral screen dataset containing chemical compounds. This dataset is available publicly on the website of the Developmental Therapeutics Program. As of March 2002, the dataset contains 43,905 classified chemical molecules.

2. The synthetic data generator was provided by Kuramochi et al. [10]. The generator allows the user to specify the number of graphs (D), their average size (T), the number of seed graphs (S), the average size of seed graphs (I), and the number of distinct labels (L).

All our experiments are performed on a 1.5GHZ, 1GB-memory, Intel PC running RedHat 8.0. Both GraphGrep and gIndex are compiled with gcc/g++.

6.1 AIDS Antiviral Screen Dataset

The experiments described in this section use the antiviral screen dataset. We set the following parameters in GraphGrep and gIndex for index construction. In GraphGrep, the maximum length of indexing paths is 10: GraphGrep enumerates all possible paths with length up to 10 and indexes them. Another parameter in GraphGrep, the fingerprint set size [14], is set as large as possible (10k). The fingerprint set consists of the hash values of indexing features. In our experiments, we do not use the technique of fingerprint since it has the similar effect on GraphGrep and gIndex: the smaller the fingerprint set, the smaller the index size and the worse the performance. In gIndex, the maximum fragment size $maxL$ is also 10; the minimum discriminative ratio γ_{min} is 2.0; and the maximum support Θ is $0.1N$. The size-increasing support function $\psi(l)$ is 1 if $l < 4$; in all other cases, $\psi(l)$ is $\sqrt{\frac{l}{maxL}}\Theta$. This means that all the fragments with size less than 4 are indexed. It should be noted that the performance is not sensitive to the selection of $\psi(l)$. There are other size-increasing support functions which can be applied, e.g., $\frac{l}{maxL}\Theta$, $(\frac{l}{maxL})^2\Theta$, and so on. We choose to have the same maximum size of features in GraphGrep and gIndex so that a fair comparison between them can be done.

We first test the index size of GraphGrep and gIndex. As mentioned before, GraphGrep indexes paths while gIndex uses discriminative frequent fragments. The test dataset consists of N graphs, denoted by Γ_N, which are randomly selected from the antiviral screen database. Figure 8 depicts the number of features used in these two algorithms with the test dataset size varied from $1,000$ to $16,000$. The curves clearly show that the index size of gIndex is at least

10 times smaller than that of GraphGrep. They also illustrate two salient properties of gIndex: its index size is *small* and *stable*. When the database size increases, the index size of gIndex does not change much. The stability of the index is due to the fact that frequent fragments and discriminative frequent fragments do not change much if the data have similar distribution. In contrast, the index size of GraphGrep may increase significantly because GraphGrep has to index all possible paths existing in a database (up to length-10 in our experiments).
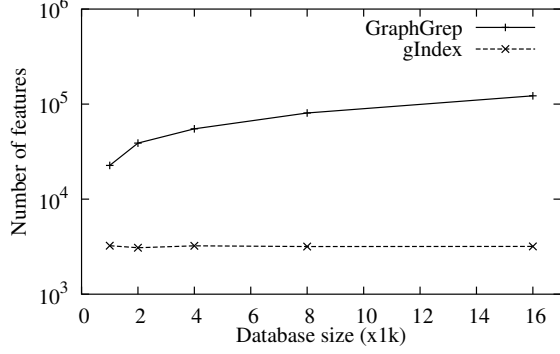
Figure 8: Index Size

Having verified the index size of GraphGrep and gIndex, we now check their performance. In Section 2, we build a query cost model. The cost of a given query is characterized by the number of candidate graphs we have to verify, i.e., the size of candidate answer set C_q. We average the cost in the following way: $AVG(|C_q|) = \frac{\sum_{q \in Q} |C_q|}{|Q|}$. The smaller the cost, the better the performance. $AVG(|D_q|)$ is the lower bound of $AVG(|C_q|)$. An algorithm achieving this lower bound actually matches the queries in the graph dataset precisely.
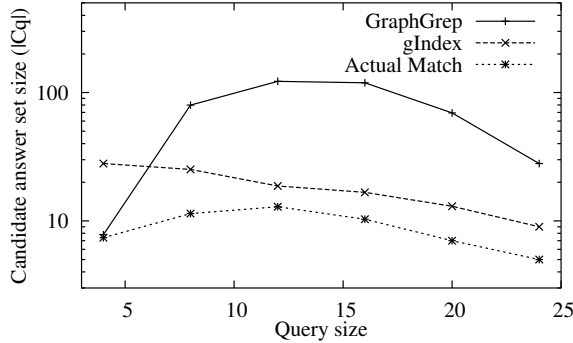

Figure 9: Low Support Queries

We select $\Gamma_{10,000}$ as the performance test dataset. Six query sets are tested, each of which has $1,000$ queries: we randomly draw $1,000$ graphs from the antiviral screen dataset and then extract a connected size-m subgraph from each graph randomly. These $1,000$ subgraphs are taken as a query set, denoted by Q_m. We generate $Q_4, Q_8, Q_{12}, Q_{16}, Q_{20}$, and Q_{24}. Each query set is then divided into two groups: low support group if its support is less than 50 and high support group if its support is between 50 and 500. We
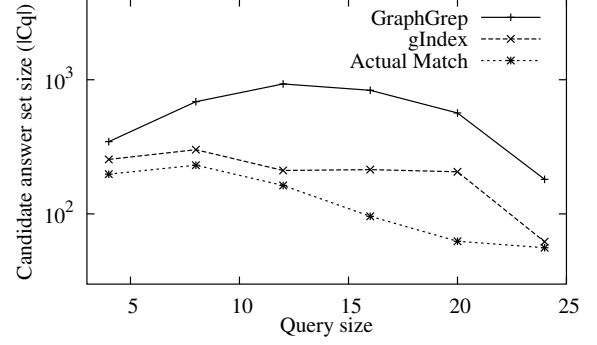

Figure 10: High Support Queries

make such elaborate partitions to demonstrate that gIndex can handle all kinds of queries very well, no matter whether they are frequent or not and no matter whether they are large or not.

Figures 9 and 10 present the performance of GraphGrep and gIndex on low support queries and high support queries, respectively. We also plot the average size of query answer sets: $AVG(|D_q|)$, which is the highest performance that an algorithm can achieve. As shown in the figures, gIndex outperforms GraphGrep nearly in every query set, except the low support queries in query set Q_4. GraphGrep works better on Q_4 simply because queries in Q_4 are more likely path-structured and the exhausted enumeration of paths in GraphGrep favors these queries. Another reason is that the setting of $\psi(l)$ in gIndex has a minimum support jump on size-4 fragments (from 1 to 632).
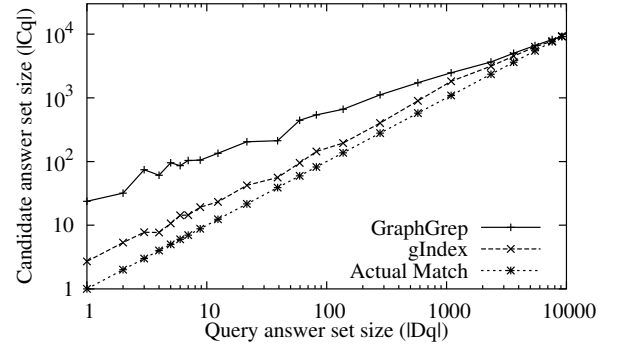

Figure 11: Performance on the Chemical Data

Figure 11 shows the performance according to the query answer set size (query support), i.e., $|D_q|$. X axis shows the actual answer set size while Y axis shows the average size of the candidate answer set, $|C_q|$, returned by these two algorithms. The closer $|C_q|$ to $|D_q|$, the better the performance. The performance gap between gIndex and GraphGrep shrinks when query support increases. The underlying reason is that higher support queries usually have simpler and smaller structures, where GraphGrep works well. When $|D_q|$ is close to $10,000$, $|C_q|$ will approach $|D_q|$ since $10,000$ is their upper bound in this test. Overall, gIndex outperforms GraphGrep by 3 to 10 times when the answer set size is below $1,000$.

Since gIndex uses frequent fragments, at the first sight, one might suspect that gIndex may not process low support queries well. However, according to the above experiments, gIndex actually performs very well on queries which have low supports or even no match in the database. This phenomena might be a bit counter-intuitive. We find that the size-increasing support constraint and the intersection power of structure-based features in gIndex are the two key factors for this robust result.
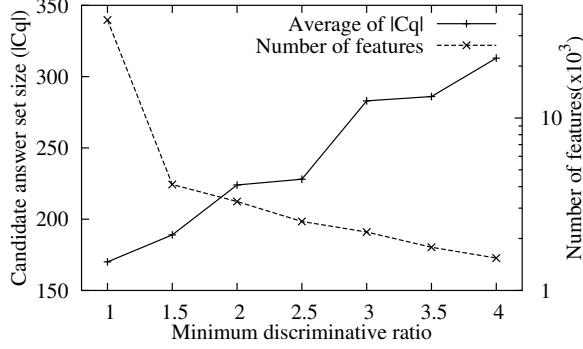

Figure 12: Sensitivity of Discriminative Ratio

Next, we check the sensitivity of minimum discriminative ratio γ_{min}. The performance and the index size with different γ_{min} are depicted in Figure 12. In this experiment, query set Q_{12} is processed on dataset $\Gamma_{10,000}$. It shows that the query response time gradually improves when γ_{min} decreases. Simultaneously, the index size increases. In practice, we have to make a trade-off between the performance and the space cost.
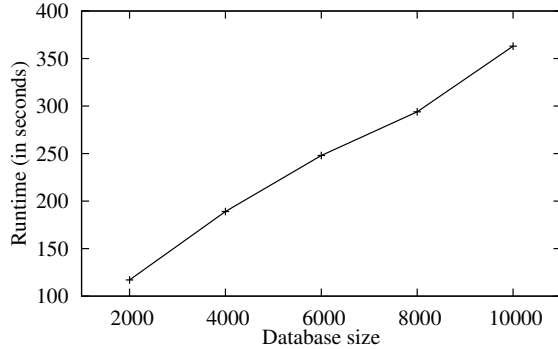

Figure 13: Scalability

The scalability of gIndex is presented in Figure 13. We vary the database size from $2,000$ to $10,000$ and construct the index from scratch for each database. As shown in the figure, the index construction time is proportional to the database size. The linear increasing trend is pretty predicable. We find that the feature set mined by gIndex for each database has around $3,000$ discriminative fragments. This number does not fluctuate a lot across different databases in this experiment, which may explain why the index construction time increases linearly. Since the size-increasing support function $\psi(l)$ follows the database size, $\psi(l) \propto \Theta \propto N$, the frequent fragment set will be relatively stable if the databases have similar distribution.
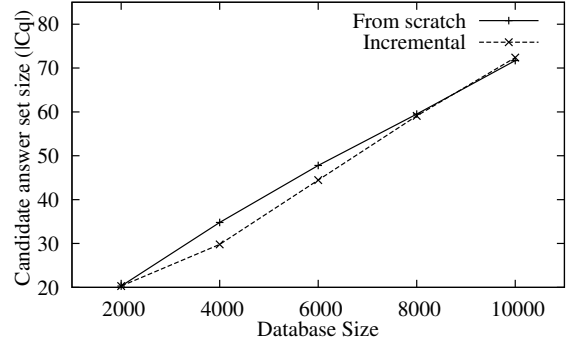

Figure 14: Incremental Maintenance

The stability of frequent fragments leads to the effectiveness of our incremental maintenance algorithm. Assume we have two databases D and $D' = D + \sum_i D_i^+$, where D_i^+'s are the updates over the original database D. As long as the graphs in D and D_i^+ are from the same reservoir, we need not build a separate index for D', instead, the feature set of D may be reused for the whole dataset D'. This remark is confirmed in the following experiment. We first take $\Gamma_{2,000}$ as the initial dataset D, and add another $2,000$ graphs into it and update the index using Algorithm 3. We repeat such addition and update four times until the dataset has $10,000$ graphs in total. The performance of the index obtained from incremental maintenance is compared with the index computed from scratch. We select the query set Q_{16} to test. Figure 14 shows the comparison between these two approaches. It is surprising that the incrementally maintained index exhibits similar performance. Occasionally, it even performs better in these datasets as pointed by the small gap between the two curves in Figure 14.

The above experiments also support a potential improvement discussed in Section 5.5: we can construct the index on a small portion of a large database, and then use the incremental maintenance algorithm to build the complete index for the whole database in **one scan**.

6.2 Synthetic Dataset

In this section, we present the performance comparison on synthetic datasets. The synthetic graph dataset is generated as follows: first, a set of S seed fragments are generated randomly, whose size is determined by a Poisson distribution with mean I. The size of each graph is a Poisson random variable with mean T. Seed fragments are then randomly selected and inserted into a graph one by one until the graph reaches its size. More details about the synthetic data generator are available in [10]. A typical dataset may have the following setting: it has 10,000 graphs and uses 1,000 seed fragments with 50 distinct labels. On average, each graph has 20 edges and each seed fragment has 10 edges. This dataset is denoted by $D10kI10T20S1kL50$.

When the number of distinct labels (L) is large, the synthetic dataset is much different from the AIDS antiviral screen dataset. Although local structural similarity appears in different synthetic graphs, there is little similarity existing among each synthetic graph. This characteristic results in a simpler index structure. We find that $maxL$ only need

to be 4 in order to achieve good performance. Both GraphGrep and gIndex perform very well on such datasets. For example, if no two vertices in one graph share the same label, we only need the vertex labels to index the graphs. This is similar to the inverted index technique (word - document id list) in document retrieval. However, when we reduce the number of distinct labels, more and more vertices share the same label. The dataset becomes more difficult to index and search. We test a synthetic dataset $D10kI10T50S200L4$ and size-12 queries (the queries are constructed using a similar method described in the previous section). The maximum size of paths and fragments is set to 5 for GraphGrep and gIndex, respectively. Figure 15 shows the average size of the candidate answer sets with different support queries.
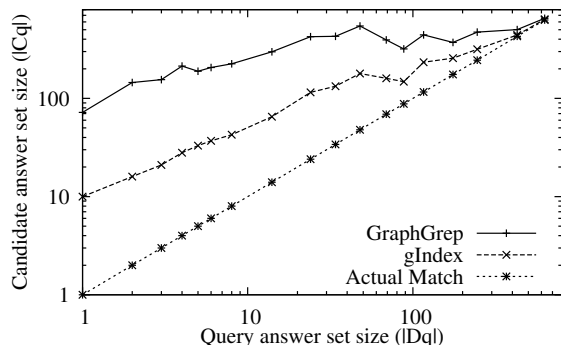

Figure 15: Performance on a Synthetic Dataset

As shown in Figure 15, gIndex performs much better than GraphGrep. When $|D_q|$ approaches 300, GraphGrep performs well. We also tested other synthetic datasets with different parameters. Similar results are also observed in these experiments.

7. RELATED WORK

The problem of graph query processing has been addressed in various fields since it is a critical problem for many applications. In content-based image retrieval, Petrakis and Faloutsos [13] represent each graph as a vector of features and index graphs in high dimensional space using R-trees. In [15], Shokoufandeh et al. represent and index graphs by a signature computed from the eigenvalues of adjacency matrix. Instead of casting a graph to a vector form, Berretti et al. [1] proposes a metric indexing scheme which organizes graphs hierarchically according to their mutual distances. In 3D protein structure search, algorithms using hierarchical alignments on secondary structure elements, e.g., Madej et al. [11], or geometric hashing [19], have already been developed for a decade. There are other literatures related to graph retrieval in these fields, which cannot be exhausted. In short, these systems are designed for other graph retrieval tasks, such as exact or similar whole graph retrieval [13, 15, 1] and 3D geometric graph retrieval [11, 19]. They are either inapplicable or inefficient to the problem studied in this paper.

In semistructured/XML databases, query languages built on path expressions become popular. Efficient index techniques for path expression are initially shown in DataGuide [6] and 1-index [12]. A(k)-index [9] further proposes k-bisimilarity to exploit local similarity existing in semistruc-

tured databases. APEX [4] and D(k)-index [3] consider the adaptivity of index structure to fit the query load. Index Fabric [5] represents every path in a tree as a string and stores it in a Patricia trie. For more complicated graph queries, Shasha et al. [14] extends the path-based technique to do full scale graph retrieval, which is also used in Daylight system [8]. Srinivasa et al. [16] builds the index based on multiple vector spaces with different abstract levels of graphs. However, no algorithm is considered to index graphs using frequent structures, which is the emphasis of this study.

Washio and Motoda [18] has a general introduction on the recent progress of graph-based data mining. In frequent graph mining, Inokuchi et al. [7], Kuramochi and Karypis [10], and Vanetik et al. [17] propose Apriori-based algorithms to discover frequent subgraphs. Yan and Han [21] and Borgelt and Berthold [2] apply the pattern-growth approach to directly generate frequent subgraphs. In this paper, we adopt a pattern-growth approach similar to [21] as the underlying graph mining engine because of its efficiency. Certainly, any kind of frequent graph mining algorithm can be used in the implementation because the mining engine itself will not influence our graph indexing results and query performance.

8. CONCLUSIONS

Graph indexing plays a critical role at efficient query processing in graph databases which have gained increasing popularity in bioinformatics, Web analysis, and other applications involving complex structures. Previous graph-indexing approaches take *paths* as indexing features and suffer from overly large index size and substantial query processing overhead.

In this paper, we have explored a rather different approach to graph indexing: indexing based on *frequent subgraph structures*. Recent progress in graph mining has turned frequent substructure-based indexing into reality. Using canonical labeling, subgraph structures can be mapped into ordered sequences. By exploring several novel concepts, especially *size-increasing support constraint* and *discriminative structure*, frequent subgraph-based indices can be made compact and effective. Also, using the incremental updating property, gIndex can be constructed by a single scan of a database. Our performance study shows that our graph indexing method, gIndex, performs better and consumes less space than the path-based indexing method.

This work can be extended to indexing trees, sequences, and other structures based on their underlying frequent patterns in the database. The frequent pattern-based indexing makes indexing adaptable to the data stored in the database and are relatively stable despite of frequent updates. This work also shows that indexing and query processing can really benefit from data mining, which may promote more studies on application of data mining at improving database system performance.

9. ACKNOWLEDGMENTS

10. REFERENCES

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

[2] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *Proc. 2002 Int. Conf. on Data Mining (ICDM'02)*, pages 211–218, Maebashi, Japan, Dec. 2002.

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

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

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

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

[7] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proc. 2000 European Symp. Principle of Data Mining and Knowledge Discovery (PKDD'00)*, pages 13–23, Lyon, France, Sept. 1998.

[8] C. A. James, D. Weininger, and J. Delany. Daylight theory manual daylight version 4.82. Daylight Chemical Information Systems, Inc, 2003.

[9] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *Proc. 2000 Int. Conf. Data Engineering (ICDE'00)*, San Jose, CA, Feb. 2002.

[10] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proc. 2001 Int. Conf. Data Mining (ICDM'01)*, pages 313–320, San Jose, CA, Nov. 2001.

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

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

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

[14] D. Shasha, J.T-L Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proc. 21th ACM Symp. Principles of Database Systems (PODS'02)*, pages 39–52, Madison, WI, Jun. 2002.

[15] A. Shokoufandeh, S. J. Dickinson, K. Siddiqi, and S. W. Zucker. Indexing using a spectral encoding of topological structure. In *Proc. IEEE Int'l Conf Computer Vision and Pattern Recognition (CVPR'99)*, Fort Collins, CO, Jun. 1999.

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

[17] N. Vanetik, E. Gudes, and S. E. Shimony. Computing frequent graph patterns from semistructured data. In *Proc. 2002 Int. Conf. on Data Mining (ICDM'02)*, pages 458–465, Maebashi, Japan, Dec. 2002.

[18] T. Washio and H. Motoda. State of the art of graph-based data mining. *SIGKDD Explorations*, 5:59–68, 2003.

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

[20] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proc. 2002 Int. Conf. on Data Mining (ICDM'02)*, pages 721–724, Maebashi, Japan, Dec. 2002.

[21] X. Yan and J. Han. CloseGraph: Mining closed frequent graph patterns. In *Proc. 2003 Int. Conf. Knowledge Discovery and Data Mining (KDD'03)*, pages 286–295, Washington, D.C., Aug. 2003.

[22] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proc. 2003 Int. Conf. Knowledge Discovery and Data Mining (KDD'03)*, pages 326–335, Washington, DC, Aug. 2003.