Graph Indexing Based on Discriminative Frequent Structure Analysis

Xifeng Yan
University of Illinois at Urbana-Champaign
Philip S. Yu
IBM T. J. Watson Research Center
Jiawei Han
University of Illinois at Urbana-Champaign

Graphs have become increasingly important in modelling complicated structures and schemaless data such as chemical compounds, proteins, and XML documents. Given a *graph query*, it is desirable to retrieve graphs quickly from a large database via indices. In this paper, we investigate the issues of indexing graphs and propose a novel indexing model based on *discriminative frequent structures* that are identified through a graph mining process. We show that the compact index built under this model can achieve better performance in processing graph queries. Since discriminative frequent structures capture the intrinsic characteristics of the data, they are relatively stable to database updates, thus facilitating sampling-based feature extraction and incremental index maintenance. Our 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 generalized and applied to indexing sequences, trees, and other complicated structures as well.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems – Query processing, Physical Design; G.2.1 [**Discrete Mathematics**]: Combinatorics – Combinatorial algorithms

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Graph Database, Frequent Pattern, Index

This is a preliminary release of an article accepted by ACM Transactions on Database Systems. The definitive version is currently in production at ACM and, when released, will supersede this version.
Authors' address: X. Yan, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, Email: xyan@cs.uiuc.edu; P. Yu, IBM T. J. Watson Research Center, Hawthorne, NY 10532, Email: psyu@us.ibm.com; J. Han, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, Email: hanj@cs.uiuc.edu.

1. INTRODUCTION

Indices play an essential role at efficient search and query processing in database and information systems. Technology has evolved from single-dimensional to multi-dimensional indexing, claiming a broad spectrum of successful applications, ranging from relational database systems to spatiotemporal, time-series, multimedia, text- and Web-based information systems. However, the traditional indexing approach may encounter challenges in certain applications involving complex objects, such as graphs. This is because a complex graph may contain an exponential number of subgraphs. It is ineffective to build an index based on vertices or edges because such features are non-selective and unable to distinguish graphs, while building index structures based on subgraphs may lead to an explosive number of index entries. To support fast access to graph databases, it is necessary to investigate new methodologies for index construction.

The importance of graph data model has been recognized in various domains. Take computer vision as an example, graphs can represent complex relationships, such as the organization of entities in images, which can be used to identify objects and scenes. In social network analysis, researchers use graphs to model social entities and their connections. In chemical informatics and bio-informatics, graphs are used to model compounds and proteins. Commercial graph management systems, such as Daylight [James et al. 2003] for compound registration, have already been put to use in chemical informatics. Benefiting from such systems, researchers are able to perform screening, designing, and knowledge discovery from compound and molecular databases.

At 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. It is known that subgraph isomorphism is an NP-complete problem [Cook].

Clearly, it is necessary to build graph indices in order to help processing graph queries. XML query is a kind of graph query, which is usually built around path expressions. Various indexing methods [Goldman and Widom 1997; Milo and Suciu 1999; Cooper et al. 2001; Kaushik et al. 2002; Chung et al. 2002; Shasha et al. 2002; Chen et al. 2003] have been developed to process XML queries. These methods are optimized for path expressions and semi-structured data. In order to answer arbitrary graph queries, systems like GraphGrep and Daylight are proposed [Shasha et al. 2002; James et al. 2003]. 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: all the existing paths in a database up to $maxL$ edges are enumerated and indexed, 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 find every graph g_i that contains all the paths (up to $maxL$ edges) in query q.

The path-based approach has two advantages:

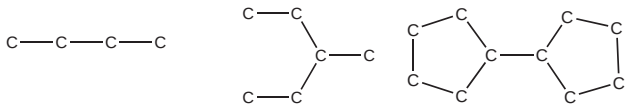
(1) Paths are easier to manipulate than trees and graphs, and
(2) the index space is predefined: all the paths up to $maxL$ edges are selected.

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 effective for complex graph queries. The advantages mentioned above now become the weak points of path-based indexing:

(1) Structural information is lost, and
(2) there are too many paths: the set of paths in a graph database sometimes is huge.

The first weakness of the path-based approaches can be illustrated using the following example.

Example 1. Figure 1 is a sample chemical dataset extracted from an AIDS antiviral screening database[1]. For simplicity, we ignore the bond type. Assume we submit a query shown in Figure 2 to the sample database. Although graph (c) in Figure 1 is the only answer, a path-based approach cannot prune graphs (a) and (b) 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 3) are not discriminative enough to tell the difference among the sample graphs. This indicates that path may not be a good structure to serve as index feature. ■



(a) (b) (c)

Fig. 1. A Sample Database

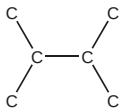


Fig. 2. A Sample Query

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

The second weakness shows that 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 around 100,000 paths with length up to 10.

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 outperform a path-based one significantly. However, it is impossible to index all the substructures in a graph database due to their exponential number. To overcome this difficulty, we devise a novel indexing model based on *discriminative frequent structures* that select the most discriminative structures to index.

Frequent structures are subgraphs that occur recurrently in a database. 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*. A graph g is *frequent* if its support is no less than a minimum support threshold, $minSup$. The relationship between frequent patterns and the underlying dataset is similar to that between phrases and documents in a text database. When the vocabulary size is small, it is obviously more beneficial to index phrases than individual words for efficient document retrieval. We believe that frequent graph can serve as indexing feature in graph databases. One may raise a fundamental problem: *If only frequent structures are indexed, how to answer those queries which only have infrequent ones?* This problem can be solved by replacing the uniform support constraint with a size-increasing support function, which has very low support thresholds for small structures but high thresholds for large ones. Therefore, for a query containing only an infrequent subgraph, the system can return a complete answer set.

The number of frequent subgraphs may still be prohibitive. We develop a mechanism to scale down the number of frequent subgraphs to be indexed. We select the most discriminative structures from the set of frequent structures. This idea leads to the development of our new algorithm, gIndex. Compared with path-based indexing, gIndex can scale down the number of indexing features in 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 maximal discriminative structures to reduce the number of subgraphs to be examined for index access and query processing.

Frequent patterns are relatively stable to database updates, thereby making incremental maintenance of index affordable. This feature provides a surprisingly efficient solution on index construction: *We can first mine discriminative frequent structures from a sample of a large database, and then build the complete index based on these structures by scanning the whole database once.* This solution has an obvious advantage when the database itself cannot be fully loaded in memory. In that case, the mining of frequent patterns without sampling usually involves multiple disk scans and thus becomes very slow.

In this paper, we thoroughly explore the issues of feature selection, index search, index construction, and incremental maintenance. 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 inspires us to further explore the application of data mining in data management. The concepts developed here can also be generalized and applied to indexing sequences, trees, and other complex structures.

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

2. GRAPH QUERY PROCESSING: FRAMEWORK AND COST MODEL

In this section, we introduce the preliminary concepts for graph query processing, outline the query processing framework, and present the cost model. The analysis of a graph indexing solution is given in the end of this section.

2.1 Preliminary Concepts

As a general data structure, labeled graph is used to model complex structured and schemaless data. In labeled graph, vertices and edges 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 $size(g)$, 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 $f : V(g) \rightarrow V(g')$, such that (1) $\forall u \in V(g)$, $f(u) \in V(g')$ and $l(u) = l'(f(u))$, 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 graph query q, it returns the query answer set $D_q = \{g_i | q \subseteq g_i, g_i \in D\}$.*

Example 2. Figure 1 shows a labeled graph dataset. We will use it as our running example. For the query shown in Figure 2, the answer set has only one element: graph (c) in Figure 1. ∎

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

(1) Index on single attributes (vertex label or edge label) is not selective enough, but an arbitrary combination of multiple attributes leads to an explosive number of index entries,

(2) query is relatively bulky, i.e., containing multiple edges, and

(3) sequential scan and test are expensive in a large database.

2.2 Framework for Graph Query Processing

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

(1) *Index construction*, which is a preprocessing step, performed before real query processing. It is done by a data mining procedure, essentially mining, evaluating, and selecting indexing features (i.e., substructures) of graphs in a database. The feature set[2] is denoted by F. For any feature $f \in F$, D_f is the set of graphs containing f, $D_f = \{g_i | f \subseteq g_i, g_i \in D\}$. In real implementation, D_f is an id list, i.e., the ids of graphs containing f. This structure is similar to the inverted index in document retrieval.

(2) *Query processing*, which consists of 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 q's features in the feature set. 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 whether they really satisfy the query. In a graph database, we have to verify the candidate answer set to prune false positives.

Obviously, the index built on vertex label (e.g., atom type in Figure 1) is not effective for fast search in a large graph database. Based on domain knowledge on chemical compounds, we may choose basic structures like benzene ring, a ring with six carbons, as an indexing feature. However, it is often unknown beforehand which structures are valuable for indexing. In this paper, we propose using data mining techniques to find them.

2.3 Cost Model

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

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

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

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

candidate answer set as much as possible. When a database is large such that the index cannot be held in the memory, T_{search} will affect the query response time.

Since we have to find all the features in the index that are contained by a query, it is important to maintain a compact feature set in the main memory. Otherwise, the cost of accessing the index may be even greater than that of accessing the database itself. Notice that the id lists of features will be kept on disk. In the next two sections, we will begin our examination of minimizing the candidate answer set $|C_q|$ and the feature set $|F|$.

2.4 Problem Analysis

Since a user may submit various queries with arbitrary structures, it is space costly to index all of them. Intuitively, the common structures of query graphs are more valuable to index since they provide better indexing coverage. When the query log is not available, we can index the common structures in a graph database.

DEFINITION 3 FREQUENCY. *Given a graph set $D = \{g_1, g_2, \ldots, g_n\}$ and a graph f, the frequency of f in D is the percentage of graphs in D containing f, $frequency(f) = \frac{|D_f|}{|D|}$.*

The graph indexing problem could be defined broadly as a "subgraph cover" problem: *given a set of query graphs, find a small subset that covers all of them, in which each graph has at least one subgraph in the cover.* If the cover set is indexed, we can generate the candidate answer set for a query graph by accessing its corresponding subgraph(s) in the index. Since small cover sets are preferred due to their compact indices, the size of the cover set becomes an important criterion in evaluating an indexing solution. A small cover set usually includes subgraphs with high frequency.

In order to answer all kinds of queries through the approach outlined in Section 2.2, the index needs to have the "downward-complete" property on subgraph containment. That is, if a graph f with size larger than 1 is present in the index, at least one of its subgraphs will be included in the index. Otherwise, a query formed by its subgraphs cannot be answered through the index. Just having the frequency and downward-complete requirements admits trivial solutions such as "index the common node labels". However, such solution does not provide the best performance. The reason is that node labels are not selective enough for complex query graphs. In the next section, we introduce a measure to select discriminative fragments by comparing their selectivity with existing indexed fragments. Note that we refer "*fragment*" to a small subgraph (i.e., structure) in graph databases and query graphs.

In summary, a good indexing solution should have the following three requirements: (1) the indexed fragments should have high frequency; (2) the index needs to have the "downward-complete" property; (3) the indexed fragments should be discriminative. In the following discussion, we introduce our solution that can satisfy these three requirements simultaneously by indexing discriminative frequent fragments with the size-increasing support constraint.

3. DISCRIMINATIVE FRAGMENT

According to the problem analysis presented in Section 2.4, 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 (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. Thus f' should be removed from the feature set. In this case, we say f' is not more *discriminative* than f.

EXAMPLE 3. All the graphs in the sample database (Figure 1) contain carbon chains: $c-c$, $c-c-c$, and $c-c-c-c$. These 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 have 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 is close to $\bigcap_{f\in F\wedge f\subseteq x} D_f$.*

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 that 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 being indexed. In such case, x is a redundant fragment. In contrast, there are fragments which are not redundant, called *discriminative fragments*.

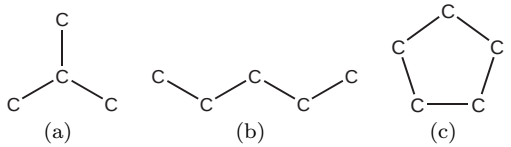


Fig. 3. Discriminative Fragments

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

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 3 (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 3 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. We devise a simple measure on the degree of redundancy. Let

fragments $f_1, f_2, \ldots$, and f_n be indexing features. Given a new fragment x, the discriminative power of x can be measured by

$$Pr(x|f_{\varphi_1}, \ldots, f_{\varphi_m}), f_{\varphi_i} \subseteq x, 1 \leq \varphi_i \leq n. \quad (2)$$

Eq. (2) shows the probability of observing x in a graph given the presence of $f_{\varphi_1}, \ldots,$ and f_{φ_m}. We denote $1/Pr(x|f_{\varphi_1}, \ldots, f_{\varphi_m})$ by γ, called *discriminative ratio*. The discriminative ratio can be calculated by the following formula:

$$\gamma = \frac{|\bigcap_i D_{f_{\varphi_i}}|}{|D_x|}, \quad (3)$$

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 features belonging to x. 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.

EXAMPLE 5. Suppose we set $\gamma_{min} = 1.5$ for the sample dataset in Figure 1. Figure 3 lists three of discriminative fragments (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 3 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 $2/1 = 2.0$. ∎

In order to mine discriminative fragments, we set a minimum discriminative ratio γ_{min} and retain fragments whose discriminative ratio is at least γ_{min}. The fragments are mined in a level-wise manner, from small size to large size.

4. FREQUENT FRAGMENT

A straightforward approach of mining discriminative fragments is to enumerate all possible fragments in a database and then prune redundant ones. This approach does not work when the fragment space is extremely large. Furthermore, a lot of infrequent fragments do not have reasonable coverage at all. Indexing these structures will not improve the query performance significantly. In this section, we will show that mining discriminative frequent fragments provides an approximate solution.

As one can see, frequent graph is a relative concept. Whether a graph is frequent or not depends on the setting of *minSup*. Figure 4 shows two frequent fragments in the sample database with $minSup = 2$. 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

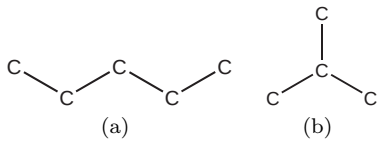


Fig. 4. Frequent Fragments

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 for query q. If *minSup* is low, it is not expensive to verify the graphs in D_f. 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 a 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 $|D_{f_i}| \geq minSup$ and $|D_{f_{i+1}}| < minSup$. 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 $|D_{f_i}|$. For many queries, $|D_{f_i}|$ is 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.* 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 *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 frequent fragments because there may exist an exponential number of frequent fragments under low support.

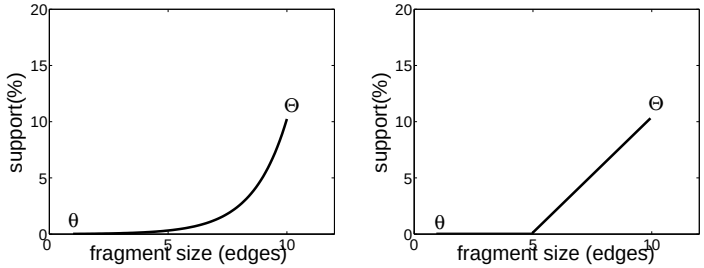
Should we enforce a uniform *minSup* for all the fragments? Let's examine a simple example: a *completely connected* graph with 10 vertices, each of which has a distinct label. There are 45 1-edge subgraphs, 360 2-edge ones, and more than $1,814,400$ 8-edge ones[3]. As one can see, in order to reduce 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 6 Frequent Pattern with Size-increasing Support. *Given a monotonically nondecreasing function, $\psi(l)$, pattern g is frequent under the* size-increasing support constraint *if and only if $|D_g| \geq \psi(size(g))$, 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 set *minSup* to be 1 for size-0 fragment to

[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 $(k < n)$.

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. The first advantage also shortens the mining process when graphs have big structures in common.



(a) exponential (b) piecewise-linear

Fig. 5. Size-increasing Support Functions

EXAMPLE 6. Figure 5 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 $\psi(l)$. ■

Using frequent fragments with the size-increasing support constraint, we have a smaller number of fragments to check their discriminative ratio. It is also interesting to examine the relationship between low supported fragments and discriminative fragments. A low supported fragment is not necessarily discriminative. For example, assume there is a low supported structure with three connected discriminative substructures. If this structure has the same support with one of its three substructures, it is not discriminative according to Definition 5.

5. GINDEX

In this section, we present the gIndex algorithm, examine the index data structures, and discuss the incremental maintenance of indices 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.

5.1 Discriminative Fragment Selection

Applying the concepts introduced in Sections 3 and 4, gIndex first generates all frequent fragments with a size-increasing support constraint. Meanwhile, it distills these fragments to identify discriminative ones. The feature selection proceeds in

a level-wise manner, i.e., Breadth-First Search (BFS). Algorithm 1 outlines the pseudo-code of feature selection.

Algorithm 1 Feature Selection

Input: Graph database D, Discriminative ratio γ_{min}, Size-increasing support function $\psi(l)$, and 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;

5.2 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.

A traditional sequentialization method is to concatenate rows or columns of the adjacency matrix of a graph into an integer sequence. However, adjacency matrix is space inefficient for sparse graphs. In our implementation, we apply a graph sequentialization method, called *DFS coding* [Yan and Han 2002; 2003], to store graphs. Certainly, other canonical labeling systems can also be used in our index framework.

EXAMPLE 7. DFS coding can translate a graph into an 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 order 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

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

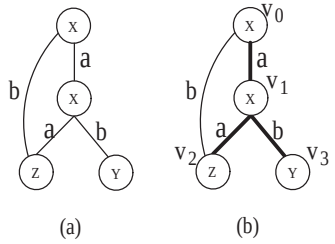


Fig. 6. DFS Code Generation

$(v_0, v_1), (v_1, v_2), (v_1, v_3)$. 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 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 as $\langle$ (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 forms a DFS code, a lexicographic order is designed in [Yan and Han 2002; 2003] to order DFS codes: the minimum DFS code among g's DFS codes is chosen as its canonical label. The readers are referred to [Yan and Han 2003] for the details of the DFS coding. ■

In the next subsections, we will introduce how to store and search the sequentialized graphs (e.g. 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., minimum DFS code). We insert the edge sequences of discriminative fragments in a prefix tree, called *gIndex Tree*.

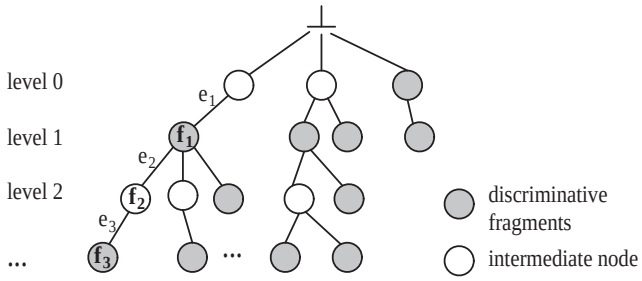


Fig. 7. gIndex Tree

EXAMPLE 8. Figure 7 shows a gIndex tree, where each node represents a sequentialized fragment (a minimum 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 the nodes 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: 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) maximal 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.

THEOREM 1. *Given a graph database and a minimum discriminative ratio, 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 number of graphs in the database.*

Proof. Let $f_0, f_1, \ldots$, and f_{k-1} be the discriminative fragments on a path, 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 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

$$N/\gamma_{min}^k \geq |D_{f_{k-1}}| \geq \psi(l), \tag{4}$$

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

EXAMPLE 9. 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. 4, 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 presented by Zaki and Gouda [2003] and handles multiple rather than one id list. The implementation of differential id list will not be examined further since it is beyond the scope of this study.

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 the search space.

With graph sequentialization, we can map any graph to an integer by hashing its canonical label. We use $c(g)$ to represent the label of a graph g, where $c(g)$ could be the minimum DFS code of g or defined by other labeling systems.

Definition 7 Graphic Hash Code. *Given a hash function h, a canonical labeling function c, and a graph g, $h(c(g))$ is called* graphic hash code.

We treat the graphic hash code as the hash value of a graph. Since two isomorphic graphs g and g' have the same canonical label, then $h(c(g)) = h(c(g'))$. Graphic hash code can help quickly locating fragments in the gIndex tree. In our implementation, the gIndex tree resides in the main memory, the inverted id-lists reside on disk and are cached in memory upon request.

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. An alternative is to enumerate features in the gIndex tree first and then check whether the query contains these features or not.

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

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 $size(x) \leq maxL$ **do**
3: **if** $x \in F$ **then**
4: $C_q = C_q \cap D_x$;
5: **return** C_q;

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 *Maximal 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 *maximal discriminative fragments*. In order to calculate C_q, we only need to perform intersection operations on the id lists of maximal discriminative fragments.

5.3.3 *Inner Support.* The previous support definition only counts the frequency of a fragment in a graph dataset. Actually, one fragment may appear several times even in one graph.

Definition 8 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 $inner\_support(x, g) \leq inner\_support(x, G)$.*

Proof. Let ρ_g be an embedding of g in G. For any embedding of x in g, ρ_x, $\rho_g \circ \rho_x$ is an embedding of x in G. Furthermore, given two different embeddings of x in g, ρ_x and ρ'_x, $\rho_g \circ \rho_x$ and $\rho_g \circ \rho'_x$ are not the same. Therefore, $inner\_support(x, g) \leq inner\_support(x, G)$. ∎

GraphGrep [Shasha et al. 2002] 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 to 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. The simplest approach is to perform a subgraph isomorphism test on each graph one by one. GraphGrep [Shasha et al. 2002] proposed an alternative approach. It records all the embeddings of paths in a 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 does not perform well. Thus, we only implement the simplest approach 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. The index maintained in this way is still of high quality if the statistics of the original database and the updated database are similar. Here, the statistics mean 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 as indexing features.

Algorithm 3 Insert/Delete

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

1: **for each** fragment $x \subseteq g$ and $size(x) \leq maxL$ **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 updating strategy leads to another interesting result: *a single database scan algorithm for 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

(with Algorithm 3) by scanning the remaining database **once**. This strategy can significantly reduce the index construction time, especially when the database is large. Notice that the improvement comes from the efficient mining of the sample database.

The index constructed by the above sampling method may have a different gIndex tree structure from the one constructed from scratch. First, in the sample database a frequent fragment may be incorrectly labeled as infrequent and be missed from the index. Secondly, the discriminative ratio of a frequent fragment may be less than its real value and hence be pruned occasionally. Nevertheless, the misses of some features will not affect the performance seriously due to the overlappings of features in gIndex. The strength of our index model does not rely on a single fragment, but on an ensemble of fragments. We need not care the composition of the index too much as long as it achieves competitive performance. Our empirical study shows that the quality of index based on a small sample does not deteriorate at all in comparison with the index built from scratch. In the following discussion we give an analytical study on the error bound of frequency and discriminative ratio of a fragment x given a sample database. Toivonen [1996] presents the frequency error bound of itemsets, which can also be applied to graphs.

THEOREM 2. *Given a fragment x and a random sample $\hat{D}$ of size n, if*

$$n \geq \frac{1}{2\epsilon^2} ln(\frac{2}{\delta}), \tag{5}$$

the probability that $|frequency(x, \hat{D}) - frequency(x)| > \epsilon$ is at most δ, where $frequency\ (x, \hat{D})$ is the frequency of x in $\hat{D}$.

Proof. [Toivonen 1996]. ∎

If we want to find the complete set of frequent fragments above the minimum support threshold, we may set a lower support in order to avoid misses with a high probability. A variation of the above theorem was developed for this purpose.

THEOREM 3. *Given a fragment x, a random sample $\hat{D}$ of size n, and a probability parameter δ, the probability that x is a miss is at most δ when*

$$low\_frequency \leq min\_frequency - \sqrt{\frac{1}{2n} ln(\frac{1}{\delta})}, \tag{6}$$

where $low\_frequency$ is the lowered minimum frequency threshold and $min\_frequency$ is the requested minimum frequency threshold.

Proof. [Toivonen 1996]. ∎

Theorem 3 shows that we may set a lower minimum support to solve the pattern loss problem. However, it becomes a problem of less interest since the support threshold is set flexibly and empirically in our solution. We do not witness great performance fall for a slightly different minimum support setting. Next, we estimate the error bound of discriminative ratio γ. Let $\hat{\gamma}$ be the random variable that expresses the discriminative ratio of an arbitrary sample $\hat{D}$.

THEOREM 4. *Given a fragment x, a set of features $f_{\varphi_1}, f_{\varphi_2}, \ldots, f_{\varphi_m}$, $f_{\varphi_i} \subseteq x$,*

$1 \leq i \leq m$, and a random sample $\hat{D}$ of size n, if

$$n \geq \frac{(2+\epsilon)^3}{p\epsilon^2} ln(\frac{4}{\delta}), \tag{7}$$

the probability that $|\gamma - \hat{\gamma}| > \epsilon\gamma$ is at most δ, where p is the frequency of x and γ is its discriminative ratio with respect to $f_{\varphi_1}, f_{\varphi_2}, \ldots$, and f_{φ_m}.

Proof. Let X be the total number of graphs in $\hat{D}$ containing x. X has the binomial distribution $B(n,p)$, $E[X] = pn$. The Chernoff bounds [Alon and Spencer 1992] shows

$$Pr[(X - pn) > a] < e^{-a^2/2pn + a^3/2(pn)^2}, \tag{8}$$

$$Pr[(X - pn) < -a] < e^{-a^2/2pn}, \tag{9}$$

where $a > 0$.

Substituting a with $\epsilon' E[X] = \epsilon' pn$, we find

$$Pr[(X - pn) > \epsilon' pn] < e^{-\epsilon'^2 pn/2 + \epsilon'^3 pn/2} = e^{(\epsilon'^3 - \epsilon'^2)pn/2},$$
$$Pr[(X - pn) < -\epsilon' pn] < e^{-\epsilon'^2 pn/2} = e^{-\epsilon'^2 pn/2}.$$

Hence,

$$Pr[|X - pn| > \epsilon' pn] < 2e^{(\epsilon'^3 - \epsilon'^2)pn/2}.$$

We set $\delta' = 2e^{(\epsilon'^3 - \epsilon'^2)pn/2}$. If

$$n \geq \frac{2}{p(\epsilon'^2 - \epsilon'^3)} ln(\frac{2}{\delta'}),$$

the probability that $|X - E[X]| > \epsilon' E[X]$ is at most δ'.

Let Y be the total number of graphs in $\hat{D}$, each of which has fragments $f_{\varphi_1}, f_{\varphi_2}, \ldots$, and f_{φ_m}. Applying the Chernoff bounds again, we note that if

$$n \geq \frac{2}{q(\epsilon'^2 - \epsilon'^3)} ln(\frac{2}{\delta'}),$$

the probability that $|Y - E[Y]| > \epsilon' E[Y]$ is at most δ', where q is the probability that a graph contains $f_{\varphi_1}, f_{\varphi_2}, \ldots$, and f_{φ_m}. Since fragments $f_{\varphi_1}, f_{\varphi_2}, \ldots, f_{\varphi_m}$ are subgraphs of x, $q \geq p$. Therefore, if $n \geq \frac{2}{p(\epsilon'^2 - \epsilon'^3)} ln(\frac{2}{\delta'})$, with probability $1 - 2\delta'$, both X and Y have a frequency error less than ϵ'. We have

$$\begin{aligned} \gamma - \hat{\gamma} &= \frac{E[Y]}{E[X]} - \frac{Y}{X} \\ \frac{E[Y]}{E[X]} - \frac{E[Y](1+\epsilon')}{E[X](1-\epsilon')} < \gamma - \hat{\gamma} &< \frac{E[Y]}{E[X]} - \frac{E[Y](1-\epsilon')}{E[X](1+\epsilon')} \\ |\gamma - \hat{\gamma}|/\gamma &< \frac{2\epsilon'}{1-\epsilon'} \end{aligned} \tag{10}$$

Set $\epsilon = \frac{2\epsilon'}{1-\epsilon'}$ and $\delta = 2\delta'$. Thus, if

$$n \geq \frac{(2+\epsilon)^3}{p\epsilon^2} ln(\frac{4}{\delta}), \tag{11}$$

the probability that the relative error of discriminative ratio is greater than ϵ is at most δ. ■

ϵ	δ	p	n
0.2	0.05	0.1	12,000
0.2	0.05	0.05	24,000
0.1	0.01	0.1	56,000
0.1	0.01	0.05	120,000

Table I. Sufficient Sample Size Given ϵ, δ, and p

Table I shows the sufficient sample size given some typical settings of parameters. For example, we observed that the performance achieved with $\gamma_{min} = 2.0$ is not very different from the one with $\gamma_{min} = 2.4$ in the chemical database we tested (see the experiment results on the sensitivity of discriminative ratio). It implies that the system will still work well under a high error ratio, e.g., $\epsilon = 10\% \sim 20\%$, and a small sample.

Because of the sampling error, some valid fragments may not meet the minimum support constraint or pass feature selection as stated above. However, since the setting of the size-increasing support function and the minimum discriminative ratio itself is empirical in our approach, such misses will not cause a problem. When the patterns have very low supports, we have to scan the database to discover these patterns, which may require a multi-pass algorithm.

Algorithm 4 Sampling based Index Construction

Input: Graph database D and Maximum fragment size $maxL$.
Output: Graph index I.

1: extract a sample D' from D;
2: select a feature set F from D' (Algorithm 1);
3: build an index I on D' using the features in F;
4: **for each** graph $g \in D - D'$ **do**
5: insert g into the index I (Algorithm 3);
6: **return** I;

Once an index is constructed using a sample, the incremental maintenance algorithm (Algorithm 3) will process the remaining graphs. Algorithm 4 depicts the sample-based index construction procedure. Because the mining time spent on the sample is limited, the time complexity of the sampling-based index construction is $O(cN)$, where c is the maximum cost of updating the index for one graph and N is the number of graphs in the database. Observe that c is usually a large constant. Thus, the sampling approach can build the index within time cost *linear* to the database size. This result is confirmed by our experiments.

The quality of an index may degrade over time after lots of insertions and deletions. A measure is required to monitor the indexed features which may be out-of-date after updates. The effectiveness of gIndex can be measured by $\frac{|\bigcap_f D_f|}{|D_x|}$ over some set of randomly selected query graphs, where $f \in F$ and $f \subseteq x$. This is the ratio of the candidate answer set size over the actual answer set size. We monitor the ratio based on sampled queries and check whether its average value changes over time. A sizable increase of the value implies that 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 [Shasha et al. 2002]. Our experiments demonstrate that gIndex achieves smaller indices and is able to outperform GraphGrep in various query loads. The effectiveness of the index returned by the incremental maintenance algorithm is also confirmed by our study: 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 were conducted on the real dataset since it is the source of real demands.

(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 Developmental Therapeutics Program. As of March 2002, the dataset contains 43,905 classified chemical molecules.

(2) The synthetic data generator was provided by Kuramochi and Karypis [Kuramochi and Karypis 2001]. The generator allows a 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 [Shasha et al. 2002], 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 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, denoted by Γ_N, consists of N graphs that 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 10 edges in our experiments).

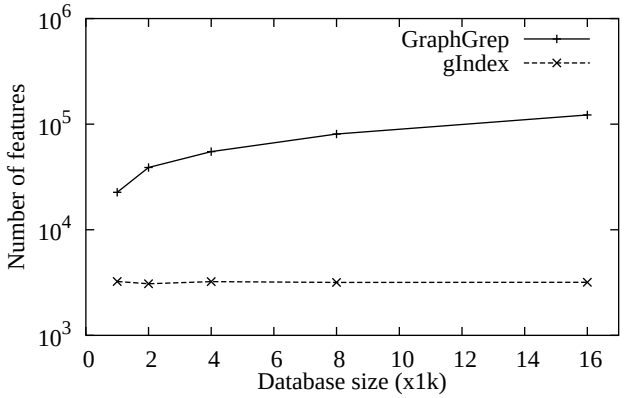


Fig. 8. Index Size

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. We use the answer set ratio, $AVG(|C_q|)/AVG(|D_q|)$, to measure the indexing strength. A ratio closer to 1 means better performance.

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

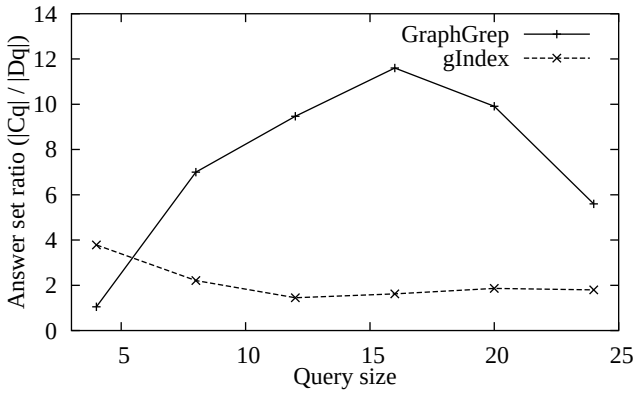


Fig. 9. Low Support Queries

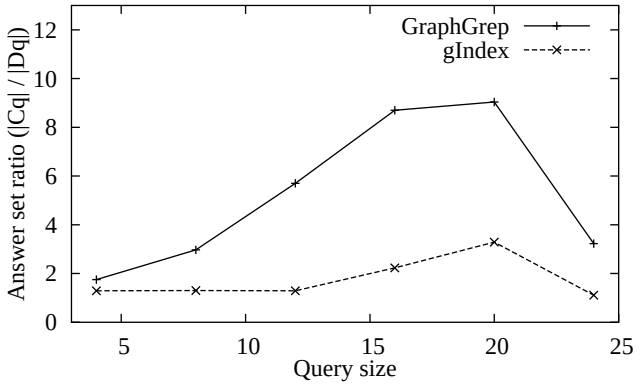


Fig. 10. High Support Queries

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 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. 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 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 shows the performance according to the query answer set size (query

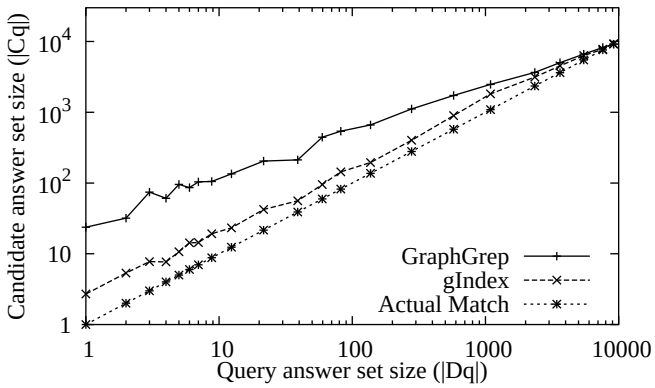


Fig. 11. Performance on the Chemical Data

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. We also plot the size of query answer set: $|D_q|$, which is the highest performance that an algorithm can achieve. The closer $|C_q|$ to $|D_q|$, the better the performance. The performance gap between gIndex and GraphGrep shrinks when the 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 its 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.

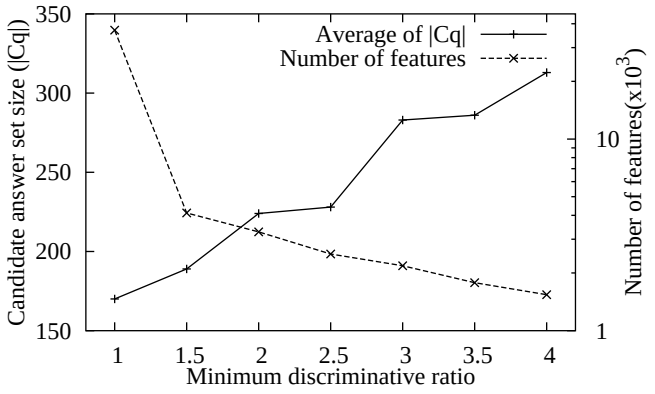


Fig. 12. Sensitivity of Discriminative Ratio

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.

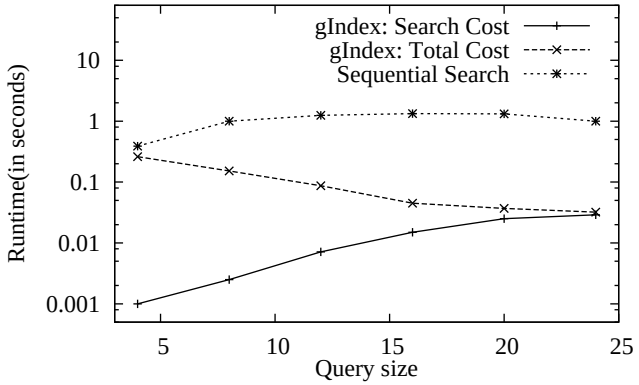


Fig. 13. gIndex Runtime

Figure 13 shows the candidate search cost (T_{search}) and the total cost (overall query response time) per query in a database with 10,000 chemical compounds. The figure also depicts the cost of brute-force search by scanning the database sequentially. This experiment is done on various query sets, from Q_4 to Q_{24}. It verifies that sequential subgraph isomorphism testing takes more time than candidate filtering, suggesting that an effective index should reduce the number of candidate graphs as much as possible. Figure 13 also indicates the query response time is closer to T_{search} when query graphs are bigger. This is reasonable since fewer graphs will become candidates for big query graphs.

The scalability of gIndex is presented in Figure 14. We vary the database size from $2,000$ to $10,000$ and construct the index from scratch for each database. We repeat the experiments for various minimum discriminative ratio thresholds. As shown in the figure, the index construction time is proportional to the database size. The linear increasing trend is pretty predicable. For example, when the minimum discriminative ratio is set at 2.0, 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 also shows that the index construction time does not change too much for a given database when the minimum discriminative ratio is above 2.0. We find that the construction time consists of two parts, frequent graph mining and discriminative feature selection. Given a database and a size-increasing support

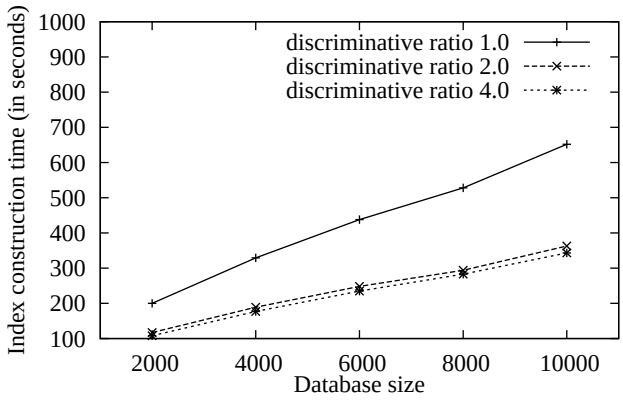


Fig. 14. Scalability: Index Construction Time

function, the cost of frequent graph mining remains constant. When the ratio is below 2.0, the number of discriminative features does not vary much (see Figure 12). Thus, the overall computation time has little deviation.

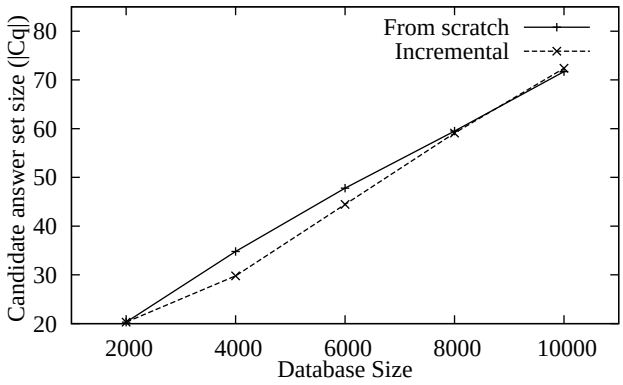


Fig. 15. Incremental Maintenance

The stability of frequent fragments leads to the effectiveness of our incremental maintenance algorithm. Assume that 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 can be reused for the whole dataset D'. This remark is confirmed by 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 15 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 15.

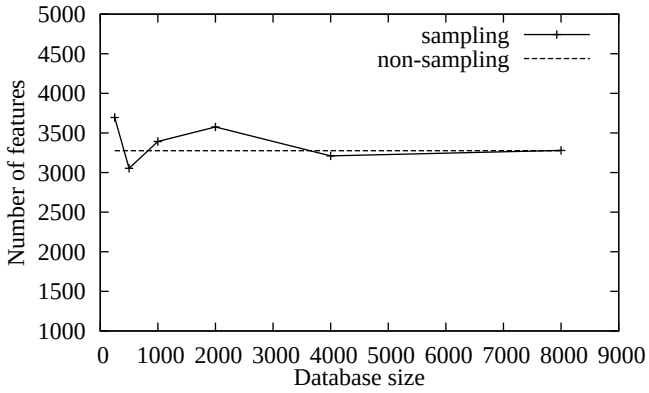


Fig. 16. Sampling: Index Size

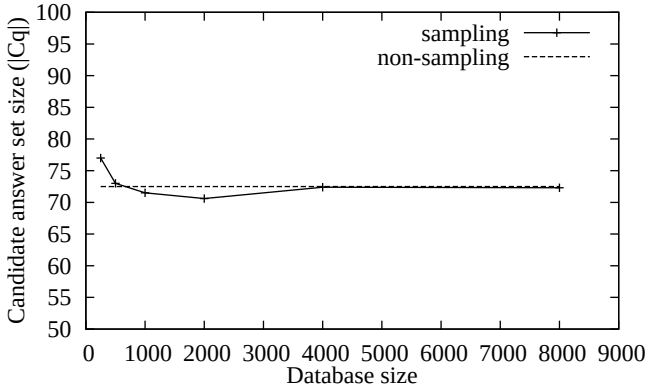


Fig. 17. Sampling: Performance

This experiment also supports 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 full scan**. This has an obvious advantage when the database itself cannot be fully loaded in memory. In that case, the mining of frequent graphs without sampling usually involves multiple disk scans and becomes very slow [Savasere et al. 1995; Toivonen 1996]. Using the sampling technique, the linear scalability illustrated in Figure 14 can be retained for extremely large databases.

The next experiment will show the quality of indices built on samples. We select $\Gamma_{10,000}$ as the test dataset and randomly draw 250, 500, 1,000, 2,000, 4,000, and

8,000 graphs from $\Gamma_{10,000}$ to form samples with different size. Totally six indices are built on these samples and are updated by the remaining graphs in the dataset. Figures 16 and 17 depict the index size and the performance (average candidate answer set size) of our sampling-based approach. The query set tested is Q_{16}. For comparison, we also plot the corresponding curves for the index built from $\Gamma_{10,000}$ (the dotted lines in the figures). It demonstrates the index built from small samples (e.g., 500 graphs) can achieve the same performance with the index built from the whole dataset (10,000 graphs). This result proves the effectiveness of our sampling method and the scalability of gIndex in large scale graph databases. Although the convergence on the number of features happens only for large sampling fractions as shown in Figure 16, the performance does not fluctuate dramatically with different sample sizes.

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 is 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 [Kuramochi and Karypis 2001]. 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 $D10kI10T20S1kL50$.

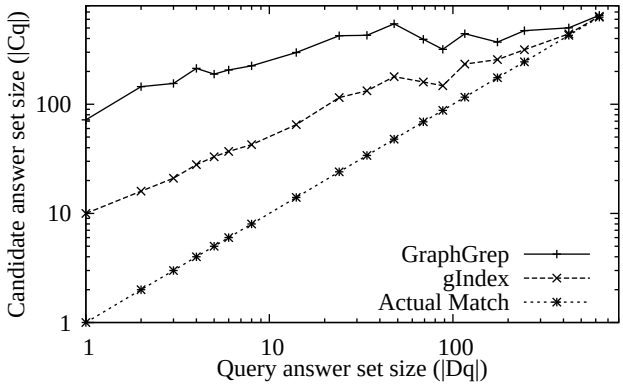


Fig. 18. Performance on a Synthetic Dataset

We first test a synthetic dataset $D10kI10T50S200L4$ and 1,000 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 18 shows the average size of the candidate answer sets with different support queries. As shown in Figure 18, gIndex performs much better than GraphGrep. When $|D_q|$ approaches 300, GraphGrep performs well too.

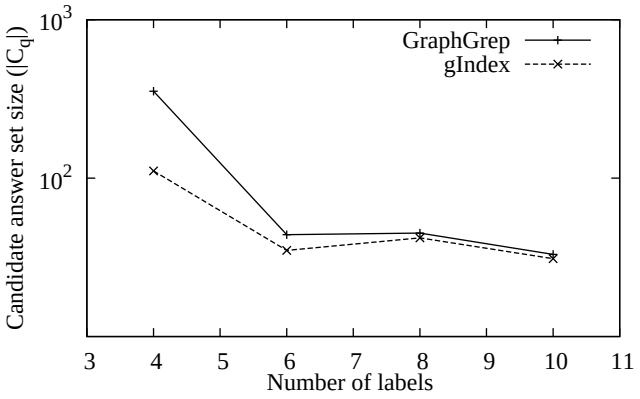


Fig. 19. Various Number of Labels

In some situations, GraphGrep and gIndex can achieve similar performance. When the size of query graphs is very large, the pruning based on the types of node and edge labels could be good enough. In this case, whether using paths or using structures as indexing features is not important any more. 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 graph. This characteristic results in a simpler index structure. For example, if every vertex in one graph has a unique label, we only need to index vertex labels. This is similar to the inverted index technique (word - document id list) used in document retrieval. In order to verify this conclusion, we vary the number of labels from 4 to 10 in the dataset $D10kI10T50S200$ and test the performance of both algorithms. Figure 19 shows that they are actually very close to each other when L is greater than 6.

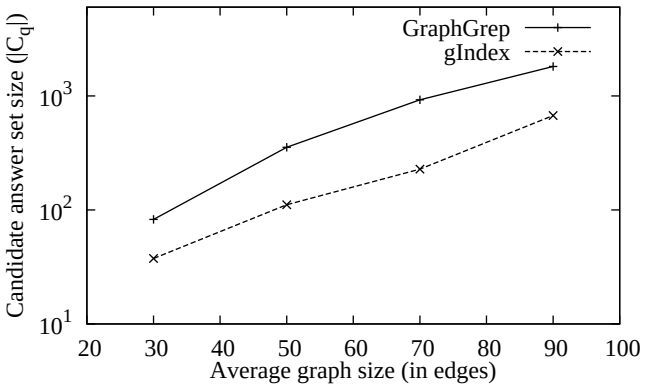


Fig. 20. Various Graph Size

Figure 20 depicts the performance comparison on the dataset $D10kI10S200L4$

with various graph sizes. In this experiment, we test 1,000 12-edge query graphs. It shows that gIndex can still outperform GraphGrep when the graph size increases. We also tested other synthetic datasets with different parameters. Similar results are also observed in these experiments.

gIndex and GraphGrep have limitations on dense graph databases that have a small number of labels. In this kind of database, the number of paths and frequent fragments increases dramatically. It is very hard to enumerate all of them. Imagine a graph becomes more and more dense, it is likely to contain any kind of query structure, which will make candidate pruning ineffective. Fortunately, these graphs are not of practical importance. Real graphs such as chemical compounds, protein networks, and image models are usually very sparse and only have a limited number of cycles.

7. DISCUSSION

In this section, we discuss the related work and the issues for further exploration.

7.1 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 [1997] represent each graph as a vector of features and index graphs in high dimensional space using R-trees. Shokoufandeh et al. [1999] 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. [2001] propose a metric indexing scheme which organizes graphs hierarchically according to their mutual distances. The SUBDUE system developed by Holder et al. [1994] uses minimum description length to discover substructures that compress the database and represent structural concepts in the data. SUBDUE is a structural pattern mining software, not systematically optimized for graph query processing. In 3D protein structure search, algorithms using hierarchical alignments on secondary structure elements [Madej et al. 1995], or geometric hashing [Wolfson and Rigoutsos 1997], have already been developed. There are other literatures related to graph retrieval in these fields which we cannot enumerate here. In short, these systems are designed for other graph retrieval tasks, such as exact or similar whole graph retrieval [Petrakis and Faloutsos 1997; Shokoufandeh et al. 1999; Beretti et al. 2001] and 3D geometric graph retrieval [Madej et al. 1995; Wolfson and Rigoutsos 1997]. 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 indexing techniques for path expression are initially shown in DataGuide [Goldman and Widom 1997] and 1-index [Milo and Suciu 1999]. A(k)-index [Kaushik et al. 2002] further proposes k-bisimilarity to exploit local similarity existing in semistructured databases. APEX [Chung et al. 2002] and D(k)-index [Chen et al. 2003] consider the adaptivity of index structure to fit the query load. Index Fabric [Cooper et al. 2001] represents every path in a tree as a string and stores it in a Patricia trie. For more complicated graph queries, Shasha et al. [Shasha et al. 2002] extend the path-based technique to do full scale graph retrieval, which is also used in Daylight system [James et al. 2003]. Srinivasa et al. [Srinivasa and Kumar 2003] build 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 [2003] have a general introduction on the recent progress of graph-based data mining. In frequent graph mining, Inokuchi et al. [2000], Kuramochi and Karypis [2001], and Vanetik et al. [2002] propose Apriori-based algorithms to discover frequent subgraphs. Yan and Han [2002; 2003] and Borgelt and Berthold [2002] apply the pattern-growth approach to directly generate frequent subgraphs. In this paper, we adopt a pattern-growth approach [Yan and Han 2002] 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.

In comparison with previous methods for indexing graphs, our approach suggests a new direction. Most of the previous indexing methods treat data in one dimension or a combination of dimensions uniformly, i.e., either index all of them or none of them. For structural data, it is difficult to determine which dimension to select. gIndex performs selective indexing based on the analysis of data characteristics. Since only some of the index entries will be selected and built, gIndex substantially reduces the total size of indices and alleviates the combinatorial explosion problem in multi-dimensional indexing.

7.2 A New Perspective on the Applications of Frequent Pattern Mining

Frequent pattern mining was originally proposed for knowledge discovery and association rule mining [Agrawal and Srikant 1994]. Research on frequent pattern mining has been focused on how to develop efficient algorithms for mining frequent patterns in large databases and how to identify interesting frequent patterns using different measures. However, since frequent pattern information represents inherent data characteristics, its application should not be confined to data mining only. Through our study, we broaden the scope of the application of frequent pattern mining and bring the usage of frequent patterns to a new perspective: frequent patterns are used as basic building blocks for indexing and database management. We show that due to the complexity of structured data like graphs, it is impossible to index all the substructures, and discriminative frequent pattern analysis will lead us to the development of compact but effective index structures. In this case, we are not interested in whether an individual discriminative frequent pattern exposes some knowledge. Instead, all of discriminative frequent patterns work together to achieve impressive performance.

7.3 Extensions to Single Graph Indexing

Although gIndex is about indexing multiple graphs, it can naturally extend to tackling the single graph indexing problem: Given a single large graph and a query graph, the task is to discover all the subregions (or subgraphs) where the query graph is embedded. Many applications have the demand of searching interesting units in a massive graph such as biological network, social network and circuit. Several issues have to be addressed in order to make the extension successful. First, we need an algorithm to mine frequent structures in a single graph. Fortunately, the solution is already available [Kuramochi and Karypis 2004]. Second, a positioning

scheme should be devised to assign the positions for each structure in the graph, which can facilitate the intersection operation (similar to $\bigcap_f D_f$ in gIndex). Note that an efficient intersection operation is very critical to the performance since both discriminative ratio computation and query processing rely on this operation. After we obtain all discriminative features through feature selection, we may apply the techniques developed in this paper to construct the indices and process the queries in a single graph.

7.4 Issues for Further Exploration

The indexing strategy proposed in this paper can be categorized into a new theme: discriminative frequent pattern-based (DFP) indexing. It is radically different from traditional indexing methods. It best fits those applications where single-dimensional indexing is not selective, multi-dimensional indexing is explosive, query is bulky, and search within complex objects is costly. We believe in such applications, the previous indexing methods encounter real challenges. There are lots of unanswered questions calling for further exploration. We name a few as follows.

First, it is important to broaden the application domains and explore the boundary of our approach. Obviously, we can apply DFP in sequence or tree databases. Nevertheless, there are many applications where the traditional indexing approach may still have its edge. For example, for gene network indexing, since genes are highly selective and user queries in many cases are also not bulky (e.g., containing only several genes), it is likely that the traditional inverted indexing approach will be a better choice. Another question is if the analyzed graphs have nearly no common subgraphs or always have big structures in common, how well does gIndex perform? The answer is positive as to indexing performance. However, it may encounter difficulties in index construction since gIndex intends to enumerate all frequent patterns in big common structures. The recent work on closed frequent graph mining [Yan and Han 2003] provides a solution by skipping the search space efficiently.

Second, we have assumed all the features are of homogeneous type, such as discriminative subgraphs, or discriminative subsequences. However, features can be of heterogeneous type in some cases. For example, in graph mining, one may consider not only subgraphs but also subsequences or subsets as features. In certain applications, this may reduce the number of features to be indexed and further enhance the efficiency of search.

Third, we considered only exact matching queries in graphs. However, in practice, there are many applications where the approximate matching is desirable. For example, in DNA sequence analysis, approximate matching on DNA subsequences is expected since mutation, insertion, and deletion are normal in genome sequences. Similar cases apply to image, video, chemical compound, and network flow analysis. It is an interesting research problem to see how the DFP indexing approach should be further developed for such applications.

8. CONCLUSIONS

Indexing complex and structured objects to facilitate the processing of nontrivial queries is an important task in database applications. In this study, we analyze why the traditional indexing approach may encounter challenges in these applications

and propose a new indexing methodology: *discriminative frequent pattern-based indexing*. This new indexing methodology is based on frequent pattern mining and discriminative feature selection. Our study on graph indexing shows that DFP leads to compact index, better query processing efficiency, and easy maintenance due to stable indexing structures. We believe our study may attract further exploration of new methods for indexing complex objects by integration of sophisticated data mining and data analysis techniques.

9. ACKNOWLEDGMENTS

The work was done in IBM T.J. Watson Research Center and University of Illinois at Urbana-Champaign. The work of the first and third authors is supported in part by the U.S. National Science Foundation NSF IIS-02-09199, the University of Illinois, and an IBM Faculty Award. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies. We would like to thank Rosalba Giugno and Dennis Shasha for providing GraphGrep; and Michihiro Kuramochi and George Karypis for providing the synthetic graph data generator. The authors are grateful to anonymous reviewers for their constructive and helpful comments on the initial versions of the paper.

REFERENCES

Agrawal, R. and Srikant, R. 1994. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. on Very Large Data Bases (VLDB'94)*. Santiago, Chile, 487–499.

Alon, N. and Spencer, J. 1992. *The Probabilistic Method*. John Wiley Inc., New York, NY.

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

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

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

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

Cook, S. The complexity of theorem-proving procedures. In *Proc. 3rd ACM Symp. on Theory of Computing (STOC'71)*. Shaker Heights, OH.

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

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

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

Inokuchi, A., Washio, T., and Motoda, H. 2000. 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)*. Lyon, France, 13–23.

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

Kaushik, R., Shenoy, P., Bohannon, P., and Gudes, E. 2002. Exploiting local similarity for efficient indexing of paths in graph structured data. In *Proc. 2000 Int. Conf. on Data Engineering (ICDE'02)*. San Jose, CA, 129–140.

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

Kuramochi, M. and Karypis, G. 2004. Finding frequent patterns in a large sparse graph. In *Proc. 2004 SIAM Int. Conf. on Data Mining (SDM'04)*. Orlando, FL.

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

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

Petrakis, E. and Faloutsos, C. 1997. Similarity searching in medical image databases. *Knowledge and Data Engineering 9,* 3, 435–447.

Savasere, A., Omiecinski, E., and Navathe, S. 1995. An efficient algorithm for mining association rules in large databases. In *Proc. 1995 Int. Conf. Very Large Data Bases (VLDB'95)*. Zurich, Switzerland, 432–443.

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

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

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

Toivonen, H. 1996. Sampling large databases for association rules. In *Proc. 1996 Int. Conf. on Very Large Data Bases (VLDB'96)*. Bombay, India, 134–145.

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

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

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

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

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

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