Scalable Mining of Large Disk-based Graph Databases *

Chen Wang† Wei Wang† Jian Pei‡ Yongtai Zhu† Baile Shi†

†Fudan University, China, {chenwang, weiwang1, 0024065, bshi}@fudan.edu.cn

‡State University of New York at Buffalo, USA & Simon Fraser University, Canada, jianpei@cse.buffalo.edu

ABSTRACT

Mining frequent structural patterns from graph databases is an interesting problem with broad applications. Most of the previous studies focus on pruning unfruitful search subspaces effectively, but few of them address the mining on large, disk-based databases. As many graph databases in applications cannot be held into main memory, scalable mining of large, disk-based graph databases remains a challenging problem. In this paper, we develop an effective index structure, *ADI* (for adjacency index), to support mining various graph patterns over large databases that cannot be held into main memory. The index is simple and efficient to build. Moreover, the new index structure can be easily adopted in various existing graph pattern mining algorithms. As an example, we adapt the well-known *gSpan* algorithm by using the *ADI* structure. The experimental results show that the new index structure enables the scalable graph pattern mining over large databases. In one set of the experiments, the new disk-based method can mine graph databases with one million graphs, while the original *gSpan* algorithm can only handle databases of up to 300 thousand graphs. Moreover, our new method is faster than *gSpan* when both can run in main memory.

Categories and Subject Descriptors: H.2.8 [Database Applications]: Data Mining

General Terms: Algorithms, Performances.

Keywords: Graph mining, index, graph database, frequent graph pattern.

1. INTRODUCTION

Mining frequent graph patterns is an interesting research problem with broad applications, including mining structural patterns from chemical compound databases, plan databases, XML documents, web logs, citation networks, and so forth. Several efficient algorithms have been proposed in the previous studies [2, 5, 6, 8, 11, 9], ranging from mining graph patterns, with and without constraints, to mining closed graph patterns.

Most of the existing methods assume implicitly or explicitly that the databases are not very large, and the graphs in the database are relatively simple. That is, either the databases or the major part of them can fit into main memory, and the number of possible labels in the graphs [6] is small. For example, [11] reports the performance of *gSpan*, an efficient frequent graph pattern mining algorithm, on data sets of size up to 320 KB, using a computer with 448 MB main memory. Clearly, the graph database and the projected databases can be easily accommodated into main memory.

Under the large main memory assumption, the computation is CPU-bounded instead of I/O-bounded. Then, the algorithms focus on effective heuristics to prune the search space. Few of them address the concern of handling large graph databases that cannot be held in main memory.

While the previous studies have made excellent progress in mining graph databases of moderate size, mining large, disk-based graph databases remains a challenging problem. When mining a graph database that cannot fit into main memory, the algorithms have to scan the database and navigate the graphs repeatedly. The computation becomes I/O-bounded.

For example, we obtain the executable of *gSpan* from the authors and test its scalability. In one of our experiments[1], we increase the number of graphs in the database to test the scalability of *gSpan* on the database size. *gSpan* can only handle up to 300 thousand graphs. In another experiment, we increase the number of possible labels in graphs. We observe that the runtime of *gSpan* increases exponentially. It finishes a data set of 300 thousand graphs with 636 seconds when there are only 10 possible labels, but needs 15 hours for a data set with the same size but the number of possible labels is 45! This result is consistent with the results reported in [11].

Are there any real-life applications that need to mine large graph databases? The answer is yes. For example, in data integration of XML documents or mining semantic web, it is often required to find the common substructures from a huge collection of XML documents. It is easy to see applications with collections of millions of XML documents. There are

*This research is supported in part by NSF grant IIS-0308001 and National Natural Science Foundation of China (No. 60303008). All opinions, findings, conclusions and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

KDD'04, August 22–25, 2004, Seattle, Washington, USA.

[1]Details will be provided in Section 6

hundreds of even thousands of different labels. As another example, chemical structures can be modeled as graphs. A chemical database for drug development can contain millions of different chemical structures, and the number of different labels in the graphs can easily go to up to 100. These large databases are disk-based and often cannot be held into main memory.

Why is mining large disk-based graph databases so challenging? In most of the previous studies, *the major data structures are designed for being held in main memory.* For example, the adjacency-list or adjacency-matrix representations are often used to represent graphs. Moreover, most of the previous methods are based on efficient random accesses to elements (e.g., edges and their adjacent edges) in graphs. However, if the adjacency-list or adjacency-matrix representations cannot be held in main memory, the random accesses to them become very expensive. For disk-based data, without any index, random accesses can be extremely costly.

Can we make mining large, disk-based graph databases feasible and scalable? This is the motivation of our study.

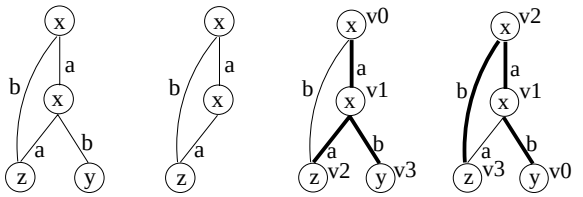
Since the bottleneck is the random accesses to the large disk-based graph databases, a natural idea is to index the graph databases properly. Designing effective and efficient index structures is one of the most invaluable exercises in database research. A good index structure can support a general category of data access operations. Particularly, a good index should be efficient and scalable in construction and maintenance, and fast for data access.

Instead of inventing new algorithms to mine large, disk-based graph patterns, *can we devise an efficient index structure for graph databases so that mining various graph patterns can be conducted scalably?* Moreover, the index structure should be easy to be adopted in various existing methods with minor adaptations.

Stimulated by the above thinking, in this paper, we study the problem of efficient index for scalable mining of large, disk-based graph databases, and make the following contributions.

- By analyzing the frequent graph pattern mining problem and the typical graph pattern mining algorithms (taking *gSpan* as an example), we identify several bottleneck data access operations in mining large, disk-based graph databases.
- We propose *ADI* (for adjacency index), an effective index structure for graphs. We show that the major operations in graph mining can be facilitated efficiently by an *ADI* structure. The construction algorithm of *ADI* structure is presented.
- We adapt the *gSpan* algorithm by using the *ADI* structure on mining large, disk-based graph databases, and achieve algorithm *ADI-Mine*. We show that *ADI-Mine* outperforms *gSpan* in mining complex graph databases and can mine much larger databases than *gSpan*.
- A systematic performance study is reported to verify our design. The results show that our new index structure and algorithm are scalable on large data sets.

The remainder of the paper is organized as follows. We define the problem of frequent graph pattern mining in Section 2. The idea of minimum DFS code and algorithm *gSpan* are reviewed in Section 3, and the major data access operations in graph mining are also identified. The *ADI* structure is developed in Section 4. The efficient algorithm *ADI-Mine* for mining large, disk-based graph databases using *ADI* is presented in Section 5. The experimental results are reported in Section 6. The related work is discussed in Section 7. Section 8 concludes the paper.



(a) Graph G (b) Subgraph G' (c) DFS-tree T_1 (d) DFS-tree T_2

Figure 1: Subgraph and DFS codes

2. PROBLEM DEFINITION

In this paper, we focus on undirected labeled simple graphs. A *labeled graph* is a 4-tuple $G = (V, E, L, l)$, where V is a set of *vertices*, $E \subseteq V \times V$ is a set of *edges*, L is a set of *labels*, and $l : V \cup E \to L$ is a labeling function that assigns a label to an edge or a vertex. We denote the vertex set and the edge set of a graph G by $V(G)$ and $E(G)$, respectively.

A graph G is called *connected* if for any vertices $u, v \in V(G)$, there exist vertices $w_1, \ldots, w_n \in V(G)$ such that $\{(u, w_1), (w_1, w_2), \ldots, (w_{n-1}, w_n), (w_n, v)\} \subseteq E(G)$.

Frequent patterns in graphs are defined based on subgraph isomorphism.

Definition 1 (Subgraph isomorphism). Given graphs $G = (V, E, L, l)$ and $G' = (V', E', L', l')$. An injective function $f : V' \to V$ is called a *subgraph isomorphism* from G' to G if (1) for any vertex $u \in V'$, $f(u) \in V$ and $l'(u) = l(f(u))$; and (2) for any edge $(u, v) \in E'$, $(f(u), f(v)) \in E$ and $l'(u, v) = l(f(u), f(v))$.

If there exists a subgraph isomorphism from G' to G, then G' is called a *subgraph* of G and G is called a *supergraph* of G', denoted as $G' \sqsubseteq G$. ∎

For example, the graph G' in Figure 1(b) is a subgraph of G in Figure 1(a).

A *graph database* is a set of tuples (gid, G), where gid is a *graph identity* and G is a graph. Given a graph database GDB, the *support* of a graph G' in GDB, denoted as $sup(G')$ for short, is the number of graphs in the database that are supergraphs of G', i.e., $|\{(gid, G) \in GDB | G' \sqsubseteq G\}|$.

For a *support threshold* min_sup ($0 \leq min\_sup \leq |GDB|$), a graph G' is called a *frequent graph pattern* if $sup(G') \geq min\_sup$. In many applications, users are only interested in the frequent recurring components of graphs. Thus, we put a constraint on the graph patterns: we only find the frequent graph patterns that are connected.

Problem definition. Given a graph database GDB and a support threshold min_sup. The problem of *mining frequent connected graph patterns* is to find the complete set of connected graphs that are frequent in GDB. ∎

3. MINIMUM DFS CODE AND GSPAN

In [11], Yan and Han developed the lexicographic ordering technique to facilitate the graph pattern mining. They also propose an efficient algorithm, *gSpan*, one of the most efficient graph pattern mining algorithms so far. In this section, we review the essential ideas of *gSpan*, and point out the bottlenecks in the graph pattern mining from large disk-based databases.

3.1 Minimum DFS Code

In order to enumerate all frequent graph patterns efficiently, we want to identify a linear order on a representation of all graph patterns such that if two graphs are in identical representation, then they are isomorphic. Moreover, all the (possible) graph patterns can be enumerated in the order without any redundancy.

The *depth-first search tree* (*DFS-tree* for short) [3] is popularly used for navigating connected graphs. Thus, it is natural to encode the edges and vertices in a graph based on its DFS-tree. All the vertices in G can be encoded in the pre-order of T. However, the DFS-tree is generally not unique for a graph. That is, there can be multiple DFS-trees corresponding to a given graph.

For example, Figures 1(c) and 1(d) show two DFS-trees of the graph G in Figure 1(a). The thick edges in Figures 1(c) and 1(d) are those in the DFS-trees, and are called *forward edges*, while the thin edges are those not in the DFS-trees, and are called *backward edges*. The vertices in the graph are encoded v_0 to v_3 according to the pre-order of the corresponding DFS-trees.

To solve the uniqueness problem, a *minimum DFS code* notation is proposed in [11].

For any connected graph G, let T be a DFS-tree of G. Then, an edge is always listed as (v_i, v_j) such that $i < j$. A linear order $\prec$ on the edges in G can be defined as follows. Given edges $e = (v_i, v_j)$ and $e' = (v_{i'}, v_{j'})$. $e \prec e'$ if (1) when both e and e' are *forward edges* (i.e., in DFS-tree T), $j < j'$ or $(i > i' \wedge j = j')$; (2) when both e and e' are *backward edges* (i.e., edges not in DFS-tree T), $i < i'$ or $(i = i' \wedge j < j')$; (3) when e is a forward edge and e' is a backward edge, $j \leq i'$; or (4) when e is a backward edge and e' is a forward edge, $i < j'$.

For a graph G and a DFS-tree T, a list of all edges in $E(G)$ in order $\prec$ is called the *DFS code* of G with respect to T, denoted as $code(G, T)$. For example, the DFS code with respect to the DFS-tree T_1 in Figure 1(c) is $code(G, T_1) = \langle (v_0, v_1, x, a, x)\text{-}(v_1, v_2, x, a, z)\text{-}(v_2, v_0, z, b, x)\text{-}(v_1, v_3, x, b, y)\rangle$, where an edge (v_i, v_j) is written as $(v_i,\ v_j,\ l(v_i),\ l(v_i, v_j),\ l(v_j))$, i.e., the labels are included. Similarly, the DFS code with respect to the DFS-tree T_2 in Figure 1(d) is $code(G, T_2) = \langle (v_0, v_1, y, b, x)\text{-}(v_1, v_2, x, a, x)\text{-}(v_2, v_3, x, b, z)\text{-}(v_3, v_1, z, a, x)\rangle$.

Suppose there is a linear order over the label set L. Then, for DFS-trees T_1 and T_2 on the same graph G, their DFS codes can be compared lexically according to the labels of the edges. For example, we have $code(G, T_1) < code(G, T_2)$ in Figures 1(c) and 1(d).

The lexically *minimum DFS code* is selected as the representation of the graph, denoted as $min(G)$. In our example in Figure 1, $min(G) = code(G, T_1)$.

Minimum DFS code has a nice property: two graphs G and G' are isomorphic if and only if $min(G) = min(G')$. Moreover, with the minimum DFS code of graphs, the problem of mining frequent graph patterns is reduced to mining frequent minimum DFS codes, which are sequences, with some constraints that preserve the connectivity of the graph patterns.

Input: a DFS code s, a graph database GDB and min_sup
Output: the frequent graph patterns
Method:

```
if s is not a minimum DFS code then return;
output s as a pattern if s is frequent in GDB;
let C = ∅;
scan GDB once, find every edge e such that
  e can be concatenated to s to form a DFS code s ◇ e
  and s ◇ e is frequent; C = C ∪ {s ◇ e};
sort the DFS codes in C in lexicographic order;
for each s ◇ e ∈ C in lexicographic order do
  call gSpan(s ◇ e, GDB, min_sup);
return;
```

Figure 2: Algorithm *gSpan*.

3.2 Algorithm gSpan

Based on the minimum DFS codes of graphs, a depth-first search, pattern-growth algorithm, *gSpan*, is developed in [11], as shown in Figure 2. The central idea is to conduct a depth-first search of minimum DFS codes of possible graph patterns, and obtain longer DFS codes of larger graph patterns by attaching new edges to the end of the minimum DFS code of the existing graph pattern. The anti-monotonicity of frequent graph patterns, i.e., *any super pattern of an infrequent graph pattern cannot be frequent*, is used to prune.

Comparing to the previous methods on graph pattern mining, *gSpan* is efficient, since *gSpan* employs the smart idea of minimum DFS codes of graph patterns that facilitates the isomorphism test and pattern enumeration. Moreover, *gSpan* inherits the depth-first search, pattern-growth methodology to avoid any candidate-generation-and-test. As reported in [11], the advantages of *gSpan* are verified by the experimental results on both real data sets and synthetic data sets.

3.3 Bottlenecks in Mining Disk-based Graph Databases

Algorithm *gSpan* is efficient when the database can be held into main memory. For example, in [11], *gSpan* is scalable for databases of size up to 320 KB using a computer with 448 MB main memory. However, it may encounter difficulties when mining large databases. The major overhead is that *gSpan has to randomly access elements (e.g., edges and vertices) in the graph database as well as the projections of the graph database many times.* For databases that cannot be held into main memory, the mining becomes I/O bounded and thus is costly.

Random accesses to elements in graph databases and checking the isomorphism are not unique to *gSpan*. Instead, such operations are extensive in many graph pattern mining algorithms, such as FSG [6] (another efficient frequent graph pattern mining algorithm) and CloseGraph [9] (an efficient algorithm for mining frequent closed graph patterns).

In mining frequent graph patterns, the major data access operations are as follows.

OP1: Edge support checking. Find the support of an edge (l_u, l_e, l_v), where l_u and l_v are the labels of vertices and l_e is the label of the edge, respectively;

OP2: Edge-host graph checking. For an edge $e = (l_u, l_e, l_v)$, find the graphs in the database where e appears;

OP3: Adjacent edge checking. For an edge $e = (l_u, l_e, l_v)$, find the adjacent edges of e in the graphs where e appears, so that the adjacent edges can be used to expand the current graph pattern to larger ones.

Each of the above operations may happen many times during the mining of frequent graph patterns. Without an appropriate index, each of the above operations may have to scan the graph database or its projections. If the database and its projections cannot fit into main memory, the scanning and checking can be very costly.

Can we devise an index structure so that the related information can be kept and all the above operations can be achieved using the index only, and thus without scanning the graph database and checking the graphs? This motivates the design of the *ADI* structure.

4. THE ADI STRUCTURE

In this section we will devise an effective data structure, *ADI* (for adjacency index), to facilitate the scalable mining of frequent graph patterns from disk-based graph databases.

4.1 Data Structure

The *ADI* index structure is a three-level index for edges, graph-ids and adjacency information. An example is shown in Figure 3, where two graphs, G_1 and G_2, are indexed.

4.1.1 Edge Table

There can be many edges in a graph database. The edges are often retrieved by the labels during the graph pattern mining, such as in the operations identified in Section 3.3. Therefore, the edges are indexed by their labels in the *ADI* structure.

In *ADI*, an edge $e = (u, v)$ is recorded as a tuple $(l(u), l(u, v), l(v))$ in the *edge table*, and is indexed by the labels of the vertices, i.e., $l(u)$ and $l(v)$, and the label of the edge itself, i.e., $l(u, v)$. Each edge appears only once in the edge table, no matter how many times it appears in the graphs. For example, in Figure 3, edge (A, d, C) appears once in graph G_1 and twice in graph G_2. However, there is only one entry for the edge in the edge table in the *ADI* structure.

All edges in the edge table in the *ADI* structure are sorted. When the edge table is stored on disk, a B+-tree is built on the edges. When part of the edge table is loaded into main memory, it is organized as a sorted list. Thus, binary search can be conducted.

4.1.2 Linked Lists of Graph-ids

For each edge e, the identities of the graphs that contain e form a *linked list of graph-ids*. Graph-id G_i is in the list of edge e if and only if there exists at least one instance of e in G_i. For example, in Figure 3, both G_1 and G_2 appear in the list of edge (A, d, C), since the edge appears in G_1 once and in G_2 twice. Please note that the identity of graph G_i appears in the linked list of edge e only once if e appears in G_i, no matter how many times edge e appears in G_i.

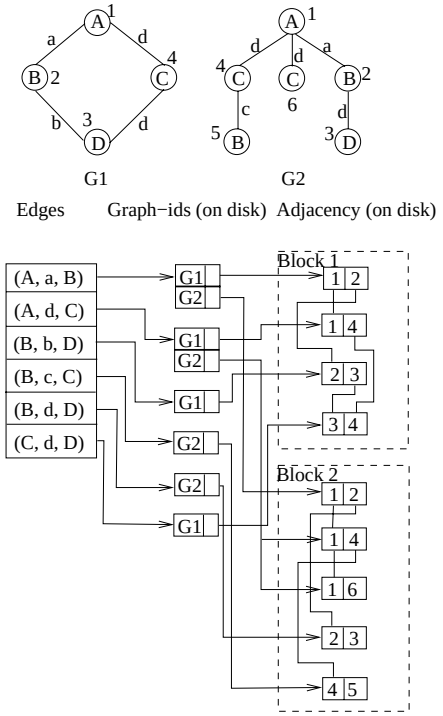


Figure 3: An *ADI* structure.

A list of graph-ids of an edge are stored together. Therefore, given an edge, it is efficient to retrieve all the identities of graphs that contain the edge.

Every entry in the edge table is linked to its graph-id linked list. By this linkage, the operation OP2: edge-host graph checking can be conducted efficiently. Moreover, to facilitate operation OP1: edge support checking, the length of the graph-id linked list, i.e., the support of an edge, is registered in the edge table.

4.1.3 Adjacency Information

The edges in a graph are stored as a list of the edges encoded. Adjacent edges are linked together by the common vertices, as shown in Figure 3. For example, in block 1, all the vertices having the same label (e.g., 1) are linked together as a list. Since each edge has two vertices, only two pointers are needed for each edge.

Moreover, all the edges in a graph are physically stored in one block on disk (or on consecutive blocks if more space is needed), so that the information about a graph can be retrieved by reading one or several consecutive blocks from disk. Often, when the graph is not large, a disk-page (e.g., of size 4k) can hold more than one graph.

Encoded edges recording the adjacency information are linked to the graph-ids that are further associated with the edges in the edge table.

4.2 Space Requirement

The storage of an *ADI* structure is flexible. If the graph database is small, then the whole index can be held into main memory. On the other hand, if the graph database is large and thus the *ADI* structure cannot fit into main

memory, some levels can be stored on disk. The level of adjacency information is the most detailed and can be put on disk. If the main memory is too small to hold the graph-id linked lists, they can also be accommodated on disk. In the extreme case, even the edge table can be held on disk and a B+-tree or hash index can be built on the edge table.

THEOREM 1 (SPACE COMPLEXITY). *For graph database* $GDB = \{G_1, \ldots, G_n\}$, *the space complexity is* $O(\sum_{i=1}^{n} |E(G_i)|)$.

Proof. The space complexity is determined by the following facts. (1) The number of tuples in the edge table is equal to the number of distinct edges in the graph database, which is bounded by $\sum_{i=1}^{n} |E(G_i)|$; (2) The number of entries in the graph-id linked lists in the worst case is the number of edges in the graph database, i.e., $\sum_{i=1}^{n} |E(G_i)|$ again; and (3) The adjacency information part records every edge exactly once. ∎

Please note that, in many application, it is reasonable to assume that the edge table can be held into main memory. For example, suppose we have 1,000 distinct vertex labels and 1,000 distinct edge labels. There can be up to $1000 \times 999 \div 2 \times 1000 = 4.995 \times 10^8$ different edges, i.e., all possible combinations of vertex and edge labels. Suppose up to 1% edges are frequent, there are only less than 5 million different edges, and thus the edge table can be easily held into main memory.

In real applications, the graphs are often sparse, that is, not all possible combinations of vertex and edge labels appear in the graphs as an edge. Moreover, users are often interested in only those frequent edges. That shrinks the edge table substantially.

4.3 Search Using ADI

Now, let us examine how the *ADI* structure can facilitate the major data access operations in graph pattern mining that are identified in Section 3.3.

OP1: Edge support checking Once an *ADI* structure is constructed, this information is registered on the edge table for every edge. We only need to search the edge table, which is either indexed (when the table is on disk) or can be searched using binary search (when the table is in main memory).

In some cases, we may need to count the support of an edge in a subset of graphs $G' \subset G$. Then, the linked list of the graph-ids of the edge is searched. There is no need to touch any record in the adjacency information part. That is, we do not need to search any detail about the edges. Moreover, for counting supports of edges in projected databases, we can maintain the support of each edge in the current projected database and thus we do not even search the graph-id linked lists.

OP2: Edge-host graph checking We only need to search the edge table for the specific edge and follow the link from the edge to the list of graph-ids. There is no need to search any detail from the part of adjacency information.

OP3: Adjacent edge checking Again, we start from an entry in the edge table and follow the links to find the list of graphs where the edge appears. Then, only the blocks containing the details of the instances of the edge are visited, and there is no need to scan the whole database. The average I/O complexity is $O(\log n + m + l)$, where n is the number of distinct edges in the graph, m is the average number of graph-ids in the linked lists of edges, and l is the average number of blocks occupied by a graph. In many applications, m is orders of magnitudes smaller than the n, and l is a very small number (e.g., 1 or 2).

Input: a graph database GDB and min_sup
Output: the *ADI* structure
Method:

```
scan GDB once, find the frequent edges;
initialize the edge table for frequent edges;
for each graph do
  remove infrequent edges;
  compute the mininmum DFS code [11];
  use the DFS-tree to encode the vertices;
  store the edges in the graph onto disk and form
    the adjacency information;
  for each edge do
    insert the graph-id to the graph-id list
      associated with the edge;
    link the graph-id to the related adjacency
      information;
  end for
end for
```

Figure 4: Algorithm of *ADI* construction.

The algorithms for the above operations are simple. Limited by space, we omit the details here. As can be seen, once the *ADI* structure is constructed, there is no need to scan the database for any of the above operations. That is, the *ADI* structure can support the random accesses and the mining efficiently.

4.4 Construction of ADI

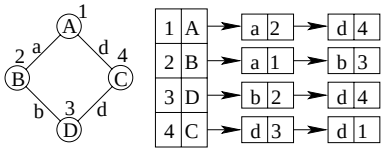
Given a graph database, the corresponding *ADI* structure is easy to construct by scanning the database only twice.

In the first scan, the frequent edges are identified. According to the apriori property of frequent graph patterns, only those frequent edges can appear in frequent graph patterns and thus should be indexed in the *ADI* structure. After the first scan, the edge table of frequent edges is initialized.

In the second scan, graphs in the database are read and processed one by one. For each graph, the vertices are encoded according to the DFS-tree in the minimum DFS code, as described in [11] and Section 3. Only the vertices involved in some frequent edges should be encoded. Then, for each frequent edge, the graph-id is inserted into the corresponding linked list, and the adjacency information is stored. The sketch of the algorithm is shown in Figure 4.

Cost Analysis

There are two major costs in the *ADI* construction: writing the adjacency information and updating the linked lists of graph-ids. Since all edges in a graph will reside on a disk page or several consecutive disk pages, the writing of adjacency information is sequential. Thus, the cost of writing adjacency information is comparable to that of making a



(a) The graph and the adjacency-lists

	1 A	2 B	3 D	4 C
1 A	0	a	0	d
2 B	a	0	b	0
3 D	0	b	0	d
4 C	d	0	d	0

(b) The adjacency-matrix

Figure 5: The adjacency-list and adjacency-matrix representations of graphs.

copy of the original database plus some bookkeeping.

Updating the linked lists of graph-ids requires random accesses to the edge table and the linked lists. In many cases, the edge table can be held into main memory, but not the linked list. Therefore, it is important to cache the linked lists of graph-ids in a buffer. The linked lists can be cached according to the frequency of the corresponding edges.

Constructing *ADI* for large, disk-based graph database may not be cheap. However, the *ADI* structure can be built once and used by the mining many times. That is, we can build an *ADI* structure using a very low support threshold, or even set $min\_sup = 1$.[2] The index is stored on disk. Then, the mining in the future can use the index directly, as long as the support threshold is no less than the one that is used in the *ADI* structure construction.

4.5 Projected Databases Using ADI

Many depth-first search, pattern-growth algorithms utilize proper projected databases. During the depth-first search in graph pattern mining, the graphs containing the current graph pattern P should be collected and form the *P-projected database*. Then, the further search of larger graph patterns having P as the prefix of their minimum DFS codes can be achieved by searching only the P-projected database.

Interestingly, the projected databases can be constructed using *ADI* structures. A projected database can be stored in the form of an *ADI* structure. In fact, only the edge table and the list of graph-ids should be constructed for a new projected database and the adjacency information residing on disk can be shared by all projected databases. That can save a lot of time and space when mining large graph databases that contain many graph patterns, where many projected databases may have to be constructed.

4.6 Why Is ADI Good for Large Databases?

In most of the previous methods for graph pattern mining, the adjacency-list or adjacency-matrix representations are used to represent graphs. Each graph is represented by an adjacency-matrix or a set of adjacency-lists. An example is shown in Figure 5.

[2]If $min\_sup = 1$, then the ADI structure can be constructed by scanning the graph database only once. We do not need to find frequent edges, since every edge appearing in the graph database is frequent.

In Figure 5(a), the adjacency-lists have 8 nodes and 8 pointers. It stores the same information as Block 1 in Figure 3, where the block has 4 nodes and 12 pointers.

The space requirements of adjacency-lists and *ADI* structure are comparable. From the figure, we can see that each edge in a graph has to be stored twice: one instance for each vertex. (If we want to remove this redundancy, the tradeoff is the substantial increase of cost in finding adjacency information). In general, for a graph of n edges, the adjacency-list representation needs $2n$ nodes and $2n$ pointers. An *ADI* structure stores each edge once, and use the linkage among the edges from the same vertex to record the adjacency information. In general, for a graph of n edges, it needs n nodes and $3n$ pointers.

Then, *what is the advantage of ADI structure against adjacency-list representation?* The key advantage is that the *ADI* structure extracts the information about containments of edges in graphs in the first two levels (i.e., the edge table and the linked list of graph-ids). Therefore, in many operations, such as the edge support checking and edge-host graph checking, there is no need to visit the adjacency information at all. To the contrast, if the adjacency-list representation is used, every operation has to check the linked lists. When the database is large so that either the adjacency-lists of all graphs or the adjacency information in the *ADI* structure cannot be accommodated into main memory, using the first two levels of the *ADI* structure can save many calls to the adjacency information, while the adjacency-lists of various graphs have to be transferred between the main memory and the disk many times.

Usually, the adjacency-matrix is sparse. The adjacency-matrix representation is inefficient in space and thus is not used.

5. ALGORITHM ADI-MINE

With the help from the ADI *structure, how can we improve the scalability and efficiency of frequent graph pattern mining?* Here, we present a pattern-growth algorithm *ADI-Mine*, which is an improvement of algorithm *gSpan*. The algorithm is shown in Figure 6.

If the *ADI* structure is unavailable, then the algorithm scans the graph database and constructs the index. Otherwise, it just uses the *ADI* structure on the disk.

The frequent edges can be obtained from the edge table in the *ADI* structure. Each frequent edge is one of the smallest frequent graph patterns and thus should be output. Then, the frequent edges should be used as the "seeds" to grow larger frequent graph patterns, and the frequent adjacent edges of e should be used in the pattern-growth. An edge e' is a *frequent adjacent edge* of e if e' is an adjacent edge of e in at least min_sup graphs. The set of frequent adjacent edges can be retrieved efficiently from the *ADI* structure since the identities of the graphs containing e are indexed as a linked-list, and the adjacent edges are also indexed in the adjacency information part in the *ADI* structure.

The pattern growth is implemented as calls to procedure *subgraph-mine*. Procedure *subgraph-mine* tries every frequent adjacent edge e (i.e., edges in set F_e) and checks whether e can be added into the current frequent graph pattern G to form a larger pattern G'. We use the DFS code to test the redundancy. Only the patterns G' whose DFS code is minimum is output and further grown. All other patterns G' are either found before or will be found later at other

Input: a graph database GDB and min_sup
Output: the complete set of frequent graph patterns
Method:

```
construct the ADI structure for the graph database if
  it is not available;
for each frequent edge e in the edge table do
  output e as a graph pattern;
  from the ADI structure, find set F_e, the set of
    frequent adjacent edges for e;
  call subgraph-mine(e, F_e);
end for
```

Procedure *subgraph-mine*
Parameters: a frequent graph pattern G, and the set of frequent adjacent edges F_e
// output the frequent graph patterns whose
// minimum DFS-codes contain that of G as a prefix
Method:

```
for each edge e in F_e do
  let G' be the graph by adding e into G;
  compute the DFS code of G'; if the DFS code is
    not minimum, then return;
  output G' as a frequent graph pattern;
  update the set F_e of adjacent edges;
  call subgraph-mine(G', F_e);
end for
return;
```

Figure 6: Algorithm *ADI-Mine*.

branches. The correctness of this step is guaranteed by the property of DFS code [11].

Once a larger pattern G' is found, the set of adjacent edges of the current pattern should be updated, since the adjacent edges of the newly inserted edge should also be considered in the future growth from G'. This update operation can be implemented efficiently, since the identities of graphs that contain an edge e are linked together in the ADI structure, and the adjacency information is also indexed and linked according to the graph-ids.

Differences Between ADI-Mine and gSpan

At high level, the structure as well as the search strategies of *ADI-Mine* and *gSpan* are similar. The critical difference is on the storage structure for graphs—*ADI-Mine* uses ADI structure and *gSpan* uses adjacency-list representation.

In the recursive mining, the critical operation is finding the graphs that contain the current graph pattern (i.e., the test of subgraph isomorphism) and finding the adjacent edges to grow larger graph patterns. The current graph pattern is recorded using the labels. Thus, the edges are searched using the labels of the vertices and that of the edges.

In *gSpan*, the test of subgraph isomorphism is achieved by scanning the current (projected) database. Since the graphs are stored in adjacency-list representation, and one label may appear more than once in a graph, the search can be costly. For example, in graph G_2 in Figure 3, in order to find an edge (C, d, A), the adjacency-list for vertices 4 and 6 may have to be searched. If the graph is large and the labels appear multiple times in a graph, there may be many adjacency-lists for vertices of the same label, and the adjacency-lists are long.

Moreover, for large graph database that cannot be held into main memory, the adjacency-list representation of a graph has to be loaded into main memory before the graph can be searched.

In *ADI-Mine*, the graphs are stored in the ADI structure. The edges are indexed by their labels. Then, the graphs that contain the edges can be retrieved immediately. Moreover, all edges with the same labels are linked together by the links between the graph-id and the instances. That helps the test of subgraph isomorphism substantially.

Furthermore, using the index of edges by their labels, only the graphs that contain the specific edge will be loaded into main memory for further subgraph isomorphism test. Irrelevant graphs can be filtered out immediately by the index. When the database is too large to fit into main memory, it saves a substantial part of transfers of graphs between disk and main memory.

6. EXPERIMENTAL RESULTS

In this section, we report a systematic performance study on the ADI structure and a comparison of *gSpan* and *ADI-Mine* on mining both small, memory-based databases and large, disk-based databases. We obtain the executable of *gSpan* from the authors. The ADI structure and algorithm *ADI-Mine* are implemented using C/C++.

6.1 Experiment Setting

All the experiments are conducted on an IBM NetFinity 5100 machine with an Intel PIII 733MHz CPU, 512M RAM and 18G hard disk. The speed of the hard disk is $10,000$ RPM. The operating system is Redhat Linux 9.0.

We implement a synthetic data generator following the procedure described in [6]. The data generator takes five parameters as follows.

D:	the total number of graphs in the data set
T:	the average number of edges in graphs
I:	the average number of edges in potentially frequent graph patterns (i.e., the frequent kernels)
L:	the number of potentially frequent kernels
N:	the number of possible labels

Please refer to [6] for the details of the data generator. For example, a data set $D10kN4I10T20L200$ means that the data set contains 10k graphs; there are 4 possible labels; the average number of edges in the frequent kernel graphs is 10; the average number of edges in the graphs is 20; and the number of potentially frequent kernels is 200. Hereafter in this section, when we say "parameters", it means the parameters for the data generator to create the data sets.

In [11], L is fixed to 200. In our experiments, we also set $L = 200$ as the default value, but will test the scalability of our algorithm on L as well.

Please note that, *in all experiments, the runtime of* ADI-Mine *includes both the* ADI *construction time and the mining time.*

6.2 Mining Main Memory-based Databases

In this set of experiments, both *gSpan* and *ADI-Mine* run in main memory.

6.2.1 Scalability on Minimum Support Threshold

We test the scalability of *gSpan* and *ADI-Mine* on the minimum support threshold. Data set $D100kN30I5T20L200$ is used. The minimum support threshold varies from 4% to 10%. The results are shown in Figure 7(a).

As can be seen, both *gSpan* and *ADI-Mine* are scalable, but *ADI-Mine* is about 10 times faster. We discussed the result with Mr. X. Yan, the author of *gSpan*. He confirms that counting frequent edges in *gSpan* is time consuming. On the other hand, the construction of *ADI* structure is relatively efficient. When the minimum support threshold is set to 1, i.e., all edges are indexed, the *ADI* structure uses approximately 57M main memory and costs 86 seconds in construction.

6.2.2 Scalability on Database Size

We test the scalability of *gSpan* and *ADI-Mine* on the size of databases. We fix the parameters $N = 30$, $I = 5$, $T = 20$ and $L = 200$, and vary the number of graphs in database from 50 thousand to 100 thousand. The minimum support threshold is set to 1% of the number of graphs in the database. The results are shown in Figure 7(b). The construction time of *ADI* structure is also plotted in the figure.

Both the algorithms and the construction of *ADI* structure are linearly scalable on the size of databases. *ADI-Mine* is faster. We observe that the size of *ADI* structure is also scalable. For example, it uses 28M when the database has 50 thousand graphs, and 57M when the database has 100 thousand graphs. This observation concurs with Theorem 1.

6.2.3 Effects of Data Set Parameters

We test the scalability of the two algorithms on parameter N—the number of possible labels. We use data set $D100kN20\text{-}50I5T20L200$, that is, the N value varies from 20 to 50. The minimum support threshold is fixed at 1%. The results are shown in Figure 7(c). Please note that the Y-axis is in logarithmic scale.

We can observe that the runtime of *gSpan* increases exponentially as N increases. This result is consistent with the result reported in [11].[3] When there are many possible labels in the database, the search without index becomes dramatically more costly. Interestingly, both *ADI-Mine* and the construction of *ADI* structure are linearly scalable on N. As discussed before, the edge table in *ADI* structure only indexes the unique edges in a graph database. Searching using the indexed edge table is efficient. The time complexity of searching an edge by labels is $O(\log n)$, where n is the number of distinct edges in the database. This is not affected by the increase of the possible labels. As expected, the size of the *ADI* structure is stable, about 57M in this experiment.

We use data set $D100kN30I5T10\text{-}30L200$ to test the scalability of the two algorithms on parameter T—the average number of edges in a graph. The minimum support threshold is set to 1%. The results are shown in Figure 7(d).

As the number of edges increases, the graph becomes more complex. The cost of storing and searching the graph also increases accordingly. As shown in the figure, both algorithms and the construction of *ADI* are linearly scalable.

We also test the effects of other parameters. The experimental results show that both *gSpan* and *ADI-Mine* are not sensitive to I—the average number of edges in potentially frequent graph patterns—and L—the number of potentially frequent kernels. The construction time and space cost of *ADI* structures are also stable. The reason is that the effects of those two parameters on the distribution in the data sets are minor. Similar observations have been reported by previous studies on mining frequent itemsets and sequential patterns. Limited by space, we omit the details here.

[3]Please refer to Figures 5(b) and 5(c) in the UIUC technical report version of [11].

6.3 Mining Disk-based Databases

Now, we report the experimental results on mining large, disk-based databases. In this set of experiments, we reserve a block of main memory of fixed size for *ADI* structure. When the size is too small for the *ADI-structure*, some levels of the *ADI* structure are accommodated on disk. On the other hand, we do not confine the memory usage for *gSpan*.

6.3.1 Scalability on Database Size

We test the scalability of both *gSpan* and *ADI-Mine* on the size of databases. We use data set $D100k\text{-}1mN30I5T20L200$. The number of graphs in the database is varied from 100 thousand to 1 million. The main memory block for *ADI* structure is limited to 250M. The results are shown in Figure 8(a). The construction time of *ADI* structure is also plotted. Please note that the Y-axis is in logarithmic scale.

The construction runtime of *ADI* structure is approximately linear on the database size. That is, the construction of the *ADI* index is highly scalable. We also measure the size of *ADI* structure. The results are shown in Figure 8(b). We can observe that the size of the *ADI* structure is linear to the database size. In this experiment, the ratio $\frac{\text{size of } ADI \text{ structure in megabytes}}{\text{number of graphs in thousands}}$ is about 0.6. When the database size is 1 million, the size of *ADI* structure is 601M, which exceeds the main memory size of our machine. Even in such case, the construction runtime is still linear.

As explained before, the construction of *ADI* structure makes sequential scans of the database and conducts a sequential write of the adjacency information. The overhead of construction of edge table and the linked lists of graph-ids is relatively small and thus has a minor effect on the construction time.

While *gSpan* can handle databases of only up to 300 thousand graphs in this experiment, *ADI-Mine* can handle databases of 1 million graphs. The curve of the runtime of *ADI-Mine* can be divided into three stages.

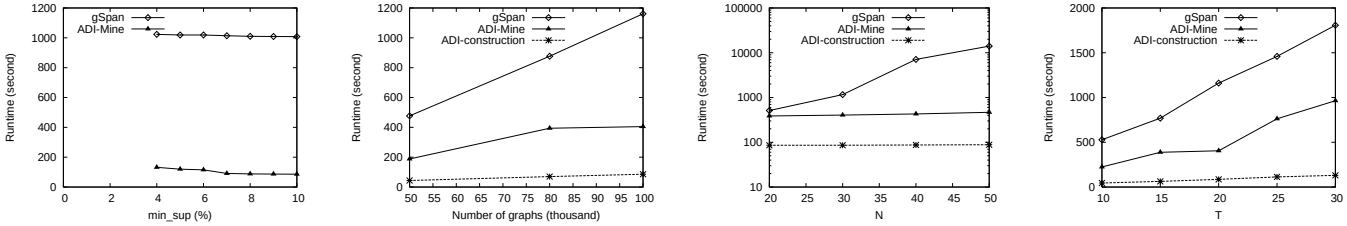
First, when the database has up to 300 thousand graphs, the *ADI* structure can be fully accommodated in main memory. *ADI-Mine* is faster than *gSpan*.

Second, when the database has 300 to 600 thousand graphs, *gSpan* cannot finish. The *ADI* structure cannot be fully held in main memory. Some part of the adjacency information is put on disk. We see a significant jump in the runtime curve of *ADI-Mine* between the databases of 300 thousand graphs and 400 thousand graphs.

Last, when the database has 800 thousand or more graphs, even the linked lists of graph-ids cannot be fully put into main memory. Thus, another significant jump in the runtime curve can be observed.

6.3.2 Tradeoff Between Efficiency and Main Memory Consumption

It is interesting to examine the tradeoff between efficiency and size of available main memory. We use data set



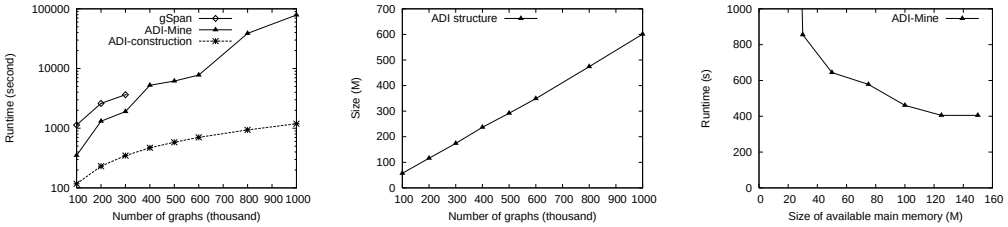
(a) scalability on min_sup $D100kN30I5T20L200$

(b) Scalability on size $D50\text{-}100kN30I5T20L200$ $min\_sup = 1\%$

(c) Scalability on N $D100kN20\text{-}50I5T20L200$ $min\_sup = 1\%$

(d) Scalability on T $D100kN30I5T10\text{-}30L200$ $min\_sup = 1\%$

Figure 7: The experimental results of mining main memory-based databases.



(a) scalability on size $D100k\text{-}1mN30I5T20L200$ $min\_sup = 1\%$

(b) Size of ADI structure $D100k\text{-}1mN30I5T20L200$ $min\_sup = 1\%$

(c) Runtime vs. main memory $D100kN30I5T20L200$ $min\_sup = 1\%$

Figure 8: The experimental results of mining large disk-based databases.

$D100kN30I5T20L200$, set the minimum support threshold to 1%, vary the main memory limit from 10M to 150M for *ADI* structure, and measure the runtime of *ADI-Mine*. The results are shown in Figure 8(c). In this experiment, the size of *ADI* structure is 57M. The construction time is 86 seconds. The highest watermark of main memory usage for *gSpan* in mining this data set is 87M. *gSpan* uses 1161 seconds in the mining if it has sufficient main memory.

When the *ADI* structure can be completely loaded into main memory (57M or larger), *ADI-Mine* runs fast. Further increase of the available main memory cannot reduce the runtime.

When the *ADI* structure cannot be fully put into main memory, the runtime increases. The more main memory, the faster *ADI-Mine* runs.

When the available main memory is too small to even hold the linked lists of graph-ids, the runtime of *ADI-Mine* increases substantially. However, it still can finish the mining with 10M main memory limit in 2 hours.

6.3.3 Number of Disk Block Reads

In addition to runtime, the efficiency of mining large disk-based databases can also be measured by the number of disk block read operations.

Figure 9(a) shows the number of disk block reads versus the minimum support threshold. When the support threshold is high (e.g., 9% or up), the number of frequent edges is small. The *ADI* structure can be held into main memory and thus the I/O cost is very low. As the support threshold goes down, larger and larger part of the *ADI* structure is stored on disk, and the I/O cost increases. This curve is consistent with the trend in Figure 7(a).

Figure 9(b) shows the number of disk block reads versus the number of graphs in the database. As the database size goes up, the I/O cost increases exponentially. This explains the curve of *ADI-Mine* in Figure 8(a).

We also test the I/O cost on available main memory. The result is shown in Figure 9(c), which is consistent with the trend of runtime curve in Figure 8(c).

6.3.4 Effects of Other Parameters

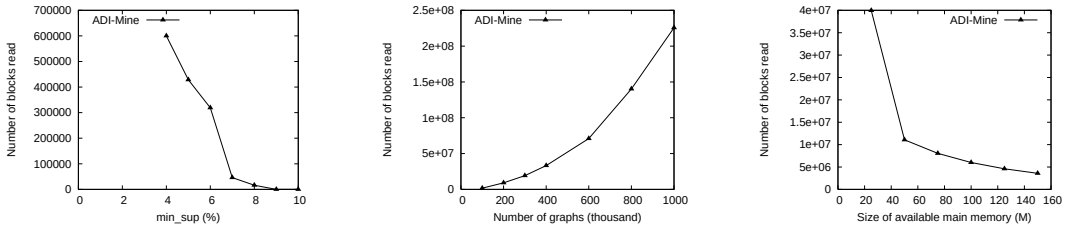
We also test the effects of the other parameters on the efficiency. We observe similar trends as in mining memory-based databases. Limited by space, we omit the details here.

6.4 Summary of Experimental Results

The extensive performance study clearly shows the following. First, both *gSpan* and *ADI-Mine* are scalable when database can be held into main memory. *ADI-Mine* is faster than *gSpan*. Second, *ADI-Mine* can mine very large graph databases by accommodating the *ADI* structure on disk. The performance of *ADI-Mine* on mining large disk-based databases is highly scalable. Third, the size of *ADI* structure is linearly scalable with respect to the size of databases. Fourth, we can control the tradeoff between the mining efficiency and the main memory consumption. Last, *ADI-Mine* is more scalable than *gSpan* in mining complex graphs—the graphs that have many different kinds of labels.

7. RELATED WORK

The problem of finding frequent common structures has been studied since early 1990s. For example, [1, 7] study the the problem of finding common substructures from chemical compounds. SUBDUE [4] proposes an approximate algorithm to identify *some*, instead of the complete set of,



(a) # blocks vs. support threshold $D100kN30I5T20L200$

(b) # blocks vs. database size $D100k$-$1mN30I5T20L200$ $min\_sup = 1\%$

(c) # blocks vs. main memory size $D100kN30I5T20L200$ $min\_sup = 1\%$

Figure 9: The number of disk blocks read in the mining.

frequent substructures. However, these methods do not aim at scalable algorithms for mining large graph databases.

The problem of mining the complete set of frequent graph patterns is firstly explored by Inokuchi et al. [5]. An Apriori-like algorithm *AGM* is proposed. Kuramochi and Karypis [6] develop an efficient algorithm, *FSG*, for graph pattern mining. The major idea is to utilize an effective graph representation, and conduct the edge-growth mining instead of vertex-growth mining. Both *AGM* and *FSG* adopt breadth-first search.

Recently, Yan and Han propose the depth-first search approach, *gSpan* [11] for graph mining. They also investigate the problem of mining frequent closed graphs [9], which is a non-redundant representation of frequent graph patterns. As a latest result, Yan et al. [10] uses frequent graph patterns to index graphs.

As a special case of graph mining, tree mining also receives intensive research recently. Zaki [12] proposes the first algorithm for mining frequent tree patterns.

Although there are quite a few studies on the efficient mining of frequent graph patterns, none of them addresses the problem of effective index structure for mining large disk-based graph databases. When the database is too large to fit into main memory, the mining becomes I/O bounded, and the appropriate index structure becomes very critical for the scalability.

8. CONCLUSIONS

In this paper, we study the problem of scalable mining of large disk-based graph database. The *ADI* structure, an effective index structure, is developed. Taking *gSpan* as a concrete example, we propose *ADI-Mine*, an efficient algorithm adopting the *ADI* structure, to improve the scalability of the frequent graph mining substantially.

The *ADI-Mine* structure is a general index for graph mining. As future work, it is interesting to examine the effect of the index structure on improving other graph pattern mining methods, such as mining frequent closed graphs and mining graphs with constraints. Furthermore, devising index structures to support scalable data mining on large disk-based databases is an important and interesting research problem with extensive applications and industrial values.

Acknowledgements

We are very grateful to Mr. Xifeng Yan and Dr. Jiawei Han for kindly providing us the executable of *gSpan* and answering our questions promptly. We would like to thank the anonymous reviewers for their insightful comments, which help to improve the quality of the paper.

9. REFERENCES

[1] D.M. Bayada, R. W. Simpson, and A. P. Johnson. An algorithm for the multiple common subgraph problem. *J. of Chemical Information & Computer Sci.*, 32:680–685, 1992.

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

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2002.

[4] L. B. Holder, D. J. Cook, and S. Djoko. Substructure discovery in the subdue system. In *Proc. AAAI'94 Workshop Knowledge Discovery in Databases (KDD'94)*, pages 359–370, Seattle, WA, July 1994.

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

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

[7] Y. Takahashi, Y. Satoh, and S. Sasaki. Recognition of largest common fragment among a variety of chemical structures. *Analytical Sciences*, 3:23–38, 1987.

[8] N. Vanetik, E. Gudes, and S.E. Shimony. Computing frequent graph patterns from semistructured data. In *Proc. 2002 Int. Conf. Data Mining (ICDM'02)*, Maebashi TERRSA, Maebashi City, Japan, Dec. 2002.

[9] X. Yan and J. Han. Closegraph: Mining closed frequent graph patterns. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'03)*, Washington, D.C, 2003.

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

[11] Y. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proc. 2002 Int. Conf. on Data Mining (ICDM'02)*, Maebashi, Japan, December 2002.

[12] M.J. Zaki. Efficiently mining frequent trees in a forest. In *Proc. 2002 Int. Conf. on Knowledge Discovery and Data Mining (KDD'02)*, Edmonton, Alberta, Canada, July 2002.