EGDIM - Evolving Graph Database Indexing Method

Shariful Islam
Department of Computer Science and Engineering
University of Dhaka, Bangladesh
tulip.du@gmail.com

Anna Fariha
Department of Computer Science and Engineering
University of Dhaka, Bangladesh
purpleblueanna.du@gmail.com

Chowdhury Farhan Ahmed
Department of Computer Science and Engineering
University of Dhaka, Bangladesh
farhan@cse.univdhaka.edu

Byeong-Soo Jeong
Department of Computer Engineering
Kyung Hee University
jeong@khu.ac.kr

ABSTRACT

Data mining is a relatively new and promising field of computer science. It is used for extracting valuable information or knowledge from large database. Data mining requires searching for frequent patterns from large database. Frequent substructure mining is also denoted by graph mining. Some of the graph mining algorithms were Apriori based and path based. *gIndex* is more robust algorithm for mining graphs. Given a query graph, this algorithm finds the supergraphs of that query graph from the graph database. *gIndex* maintains an index of graph database according to *discriminative fragments*. In this paper, a further improvement over the existing *gIndex* algorithm is proposed. More information is stored in the index data structure to quickly answer the graph query, discarding the unnecessary graphs. The proposed method in this paper handles sudden change in database graph patterns efficiently and is capable of processing queries in dynamic and evolving database which *gIndex* can not handle. It also ensures a good running time for processing graph queries.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data Mining

General Terms

Algorithms, Management, Performance, Design, Experimentation

Keywords

Data mining, Knowledge discovery, Graph mining.

ICUIMC'12, February 20-22, 2012, Kuala Lumpur, Malaysia

1. INTRODUCTION

Data mining refers to the process of discovering knowledge and determine patterns and their relationships by analyzing large database. A primary reason for using data mining is to assist in the analysis of collections of observations of behavior [1]. Many business decisions are greatly benefited by data mining. It is a powerful technology with great potential to help industries to focus on the most important information. Data mining is applicable in biological and chemical research, bank, business companies, government decisions, computer security, education, finance, customer relationship management and many other areas. Many industries use data mining to increase sales and guess future investments. Subject-based data mining is also being used to search the associations between individuals in data. There are various fields of data mining. Frequent pattern mining, classification, clustering, graph mining, web mining are some notable fields of data mining.

Discovery of frequent patterns is the most important topic of investigation in the field of data mining. For a given database, a minimum threshold is defined. Patterns having their frequency exceeding the minimum threshold are frequent patterns. In business, mining frequent patterns may aim at finding regularities in the behavior of customer of sudden markets. It is also useful in ordering the items in a supermarket.

A special kind of frequent pattern mining is frequent substructure mining [2, 3, 4, 5, 6, 7]. Substructures can be represented by graphs. This opens a new and special field of data mining, **Graph Mining**. Graphs are widely used data structures for representing schema less data in biological, chemical and other important fields. Proteins, DNA, WEB and XML documents, links, circuits and chemical compounds are represented using graphs [8, 9]. These are complicated structures and graph is the best data structure to represent them. Graph represents not only the properties of the elements, but also the relationship between the elements. As a result, graph becomes the most appropriate data structure for representing schema less and complicated structures. Web mining, biological network analysis, social networks and bioinformatics fields are based on graph applications. Efficient retrieval of the wealth of information

in graph-based models is an essential task in pattern recognition and draws attention from database community. Systems are required to search for all occurrences of a query graph in a graph database. The problem of processing subgraph queries on a database is finding the set of graphs in the database that are supergraphs of the query. The main challenge to solve this problem is the size of the database and the NP completeness of subgraph isomorphism problem [10] that is heavily required in answering subgraph queries [11, 12, 13, 14]

Apriori based frequent substructure mining algorithm, which proceeds in a bottom-up manner, adopting a level-wise mining methodology, shares similar characteristics with Apriori-based frequent itemset mining algorithms. The *gSpan* algorithm is a path based graph mining algorithm, designed to reduce the generation of duplicate graphs. But it is not efficient to search the whole database for answering the graph query. Sequential scan not only searches the whole database, it also checks each graph in the database whether that is a supergraph of the query graph, which is an NP complete problem. As a result, a better approach has to be proposed. An index of the graph database can be maintained to answer the graph queries [15],[16].

Indexing a graph database can be done using two types of features. One is frequent pattern and the other is path or substructure. *GraphGrep* [17] is an example of path based graph mining algorithm. *gIndex* [4] is an even better and robust algorithm for mining graphs. *gIndex* uses frequent substructure as its indexing feature. Although paths are easy to manipulate and taking paths as the indexing feature leaves the indexing space predefined, frequent substructures are more expressive in representing the properties of a graph. Using paths as the indexing features causes many structural information to be lost. So frequent substructures are ideal candidate for the indexing feature. This research extends the existing *gIndex* algorithm to achive an even better performance for evolving database.

gIndex focuses on some optimizations when designing the graph index. *gIndex* only takes into account those frequent substructures which are discriminative enough. In order to avoid the exponential growth of the number of frequent subgraphs, the support threshold is progressively increased when the subgraphs grow large. That is, *gIndex* uses low minimum support for small subgraphs and high minimum support for large subgraphs. Moreover, the concept of discriminative structure is introduced to reduce the redundancy among the frequent subgraphs selected as indexing features [4].

gIndex assumes that the database is stable to updates. As *gIndex* does not take into account any nearly frequent substructures, performance degrades due to random growth of database. The problems of *gIndex* motivated us to propose a better method over *gIndex*. The proposed method, **EGDIM** (**E**volving **G**raph **D**atabase **I**ndexing **M**ethod), keeps track of the recent semi-frequent substructures having support close to minimum support. When the support of these substructures reaches the minimum support, these substructures are considered as indexing features. For memory efficiency, an idea to remove the least recently encountered discriminative frequent substructre from the index structure is also proposed.

2. RELATED WORK

Frequent patterns are patterns (such as itemsets, subsequences, or substructures) that appear in a data set frequently. For example, a set of items that appear frequently together in a transaction data set is a frequent itemset. Finding such frequent patterns plays an essential role in mining associations, correlations, and many other interesting relationships among data. Moreover, it helps in data classification, clustering, and other data mining tasks as well. Thus, frequent pattern mining has become an important data mining task and a focused theme in data mining research [18], [19]. Apriori algorithm is an innovative way to find association rules on large scale, allowing implication outcomes that consist of more than one item. Despite of the simplicity of Apriori algorithm, it is costly for mining frequent itemsets in large database, so FP-growth approach is proposed for better performance.

2.1 gSpan

One problem of Apriori based algorithm is, it generates duplicate patterns. It also scans the database many times, that degrades the performance. In order to avoid such overhead, non-Apriori-based algorithms have recently been developed, most of which adopt the pattern-growth methodology. *gSpan* and *graphGrep* algorithms exist for frequent substructure mining. In algorithm 1, the procedure of *gSpan* is described.

Algorithm 1. *gSpan*. Pattern growth-based frequent substructure mining [18]

Input:
s, a DFS code.
D, a graph data set. **min_sup**, the minimum support threshold.

Output:
The frequent graph set, **S**.

Method:
$\mathbf{S} \leftarrow \phi$
Call *gSpan* (**s**, **D**, **min_sup**, **S**)
procedure PatternGrowthGraph (**s**, **D**, **min_sup**, **S**)
1. if $\mathbf{s} \neq$ **dfs (s)**, then
2. return
3. insert **s** into **S**
4. set **C** to ϕ
5. scan **D** once, find all the edges **e** such that **s** can be right-most extended to $\mathbf{s}\diamond_{\mathbf{e}}$;
insert $\mathbf{s}\diamond_{\mathbf{e}}$ into **C** and count its frequency
6. sort **C** in DFS lexicographic order
7. for each frequent $\mathbf{s}\diamond_{\mathbf{e}}$ in **C** do
8. *gSpan* ($\mathbf{s}\diamond_{\mathbf{e}}$, **D**, **min_sup**, **S**)
9. return

2.2 gIndex

Structured data mining is the process of finding and extracting useful information from semi-structured data set-

s. Graph mining is a special case of structured data mining. For mining graph, we have to be able to search in the database for the presence of a query graph. In the core of graph mining lies the subgraph query problem. Apriori-based frequent substructure mining algorithms share similar characteristics with Apriori-based frequent itemset mining algorithms. The problem of Apriori based algorithm is, it generates duplicate patterns and scans the database many times. In order to avoid such overhead, non-Apriori-based algorithms have recently been developed, most of which adopt the pattern-growth methodology. Features of *gIndex* algorithms are size-increasing support constraint, that uses low minimum support on small fragments for effectiveness and high minimum support on large fragments for compactness, using discriminative fragments as indexing feature and Apriori pruning, that enables *gIndex* to optimize its efficiency.

gIndex keeps an index for storing necessary information about the database. *gIndex* builds graph indices to help processing graph queries and retrieve related graphs. It is also used in indexing sequences, trees, and other complex structures. It will be described briefly and compared with EGDIM later.

3. PRELIMINARIES

Some terms and definitions required for the understanding of EGDIM is discussed in this section.

Definition 1. Vertex set: *V (g)* denotes the vertex set of a graph *g*. Each vertex has a label, that denotes its property.

Definition 2. Edge set: *E (g)* denotes the edge set of a graph *g*. Each edge has a label, that denotes the connection property of the link between the vertices it connects.

Definition 3. Size of a graph *g*: $|E\ (g)|$ denotes the Size of a graph *g*.

Definition 4. Subgraph and Supergraph: Given a graph A and a graph B, if removing some vertices and edges from A generates B, then B is a sub graph of A and A is a supergraph of B.

Definition 5. Graph isomorphism: In graph theory, an isomorphism of graphs A and B is a bijection between the vertex sets of A and B, $f: V(A) \rightarrow V(B)$, such that any two vertices u and v of A are adjacent in A if and only if $f(u)$ and $f(v)$ are adjacent in B [20].

Definition 6. Subgraph isomorphism problem: The subgraph isomorphism problem is a computational task in which two graphs A and B are given as input, and one must determine whether A contains a subgraph that is isomorphic to B [21].

Definition 7. Support or Frequency: Given a labeled graph data set, $D = \{G_1, G_2, ..., G_n\}$, we define support(g) or frequency(g) as the percentage (or number) of graphs in D where g is a subgraph [18]. $|D_g|$ is the number of graphs in D where g is a subgraph. $|D_g|$ is called (absolute) support, denoted by support(g) [4].

Definition 8. Minimum support: For a database D and a timestamp T, a value is defined as the minimum threshold for support to identify fragments for further consideration. It is called minimum support or *min_sup* in short.

Definition 9. Frequent fragment: A frequent fragment (or graph, substructure, subgraph) is a graph whose support is no less than min_sup [18].

Definition 10. Closed frequent subgraph: A frequent graph G is closed if and only if there is no proper supergraph G that has the same support as G [18].

Definition 11. Maximal frequent graph: A closed frequent graph G is maximal if and only if there is no proper supergraph G that is also frequent [18].

Definition 12. Right-most vertex: Given a DFS tree T, of a graph G, the last visited vertex, v_n, is called the right-most vertex [18].

Definition 13. Right-most path: Given a DFS tree T, of a graph G, we call the starting vertex in T, v_0, the root. The straight path from v_0 to the right-most vertex v_n is called the right-most path [18].

Definition 14. Forward extension: Given a graph G and a DFS tree T in G, a new edge e can introduce a new vertex and connect to a vertex on the right-most path. This type of extension is called forward extension [18].

Definition 15. Backward extension: Given a graph G and a DFS tree T in G, a new edge e can be added between the right-most vertex and another vertex on the right-most path. This type of extension is called backward extension [18]. Both forward and backward extensions are also known as right-most extension.

Definition 16. DFS code and DFS lexicographic order: DFS code is an edge sequence for subscripting graphs to build an order among different graph subscriptings. Two kinds of orderings are necessary to order DFS codes lexicographically.

Definition 17. Minimum DFS code: Based on the DFS lexicographic ordering, the minimum DFS code of a given graph G, written as **dfs(G)**, is the minimal one among all the DFS codes. The subscripting that generates the minimum DFS code is called the **base subscripting**.

Definition 18. Feature set: The set of subgraphs that are considered as a property of a graph. The graph feature set is denoted by F. For any graph feature $f \in F$, D_f is the set of graphs containing f, $D_f = \{g_i | f \subseteq g_i,\ g_i \in D_g\}$ [4].

Definition 19. Candidate query answer set: The first step of query processing, searching, enumerates all the features in a query graph, q, to compute the candidate query answer set, $C_q = \cap_f\ D_f$ *(f* $\subseteq$ *q and f* $\subseteq$ *F)*; each graph in C_q contains all q's features in the feature set. Therefore, D_q is a subset of C_q [4].

Definition 20. Redundant fragment: Fragment x is redundant with respect to feature set F if $D_x \approx \cap_{f \in F \wedge f \subseteq x} D_f$ [4]. That means if a fragment's presence can be predicted by the presence of it's subgraphs, then it is a redundant fragment.

Definition 21. Discriminative fragment: Fragment x is discriminative with respect to feature set F if $D_x \ll \cap_{f \in F \wedge f \subseteq x} D_f$ [4].

Definition 22. Discriminative ratio: The discriminative ratio can be calculated by the following formula:

$$\gamma = \frac{|\cap_i D_{\varphi_i}|}{|D_x|}$$

where D_x is the set of graphs containing x and $\cap_i D_{\varphi_i}$ is the set of graphs which contain the subgraphs of x in the feature set [4]. The value of γ denotes how discriminative the fragment is. It is always $\geq$ *1*. If it is $\gg$ *1*, then it is much discriminative than it's subgraphs.

Definition 23. Prefix tree: It is an efficient tree data structure used for indexing graphs according to their prefixes

after translating fragments into sequenc

4. EGDIM - EVOLVING GRAPH DATABASE INDEXING METHOD

Although *gIndex* started a new way to answer the subgraph queries through an indexing approach, it has some drawbacks. The main drawback of *gIndex* algorithm is that, it can not work efficiently when it has to work with small size of initial database. When working with small size of initial database, *gIndex* creates a lot of fragments which are not necessary for indexing and it also may discard fragments which may be frequent on later updates. Another lacking of *gIndex* is, it can not handle random change in database updates. This lacking occurs because *gIndex* only stores the fragments which have their frequencies above the minimum support. For the absence of this feature, once the preprocessing is done in *gIndex* algorithm, if fragments, having frequencies close to the minimum support, are encountered later, they will not be added to the *gIndex* tree, no matter how many times they are encountered.

To overcome the drawbacks of *gIndex* algorithm, we propose a modified approach, EGDIM, over *gIndex* algorithm. Here, at the beginning of frequent fragment selection process, not only the fragments which have frequencies above the minimum support is stored, but also track of fragments having frequencies close to the minimum support is kept. Another improvement that is showed is, EGDIM approach can handle **dynamic or evolving data** in the updates of the database. For example, let us consider the scenario for cellphone marketing. Prior to 2007, the database had no occurrence of iPhone. But in 2007, iPhone selling was started and soon it became one of the most frequent items in cellphone market. This change was random with respect to the initial database. *gIndex* is unable to handle this sudden change in database pattern. But EGDIM simply adopts this new frequent item for mining cellphone market field. The main drawback of this approach is that, it needs extra memory space for storing new fragments, where reducing memory space is one of the major properties of *gIndex*. To overcome this problem, this approach is extended. The idea is, if a fragment is very old (according to its use or update), then that fragment is removed. That means EGDIM removes the LRU (least recently used) fragment from stored database for semi-frequent fragments.

The algorithm for substructure mining in evolving database can be divided into five phases. These phases are similar to *gIndex* algorithm but adds some improvements for performing well in dynamic database. The first two phases are for preprocessing, that is, for building the initial index tree from the given database. The next three phases are for incremental updates. In the following subsections, we will describe the phases with appropriate exmaple based on the graph database shown in figure 1.

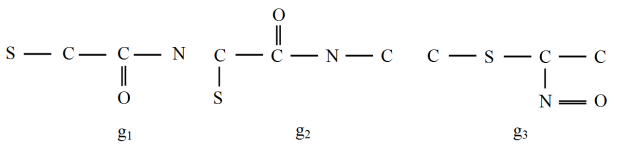

Figure 1: Example graph database

4.1 Discriminative fragment selection

In **discriminative fragment selection** phase, all frequent fragments according to **size-increasing support constraint** are generated. To generate these fragments, simple BFS (breadth first search) approach is used. BFS approach is used for the level-wise expansion of the fragments, which helps to differentiate the **discriminative fragments** from the **redundant fragments**. We take into consideration the discriminative fragments, and eliminate the redundant fragments. As some fragments of the graph database (redundant fragments) are not considered for later computation, it saves a lot of space as well as minimizes the query processing time. Algorithm 2 provides the pseudo code for the discriminative fragment selection phase.

Algorithm 2. Feature selection algorithm

Input:
Graph database **D**.
Discriminative ratio γ_{min}.
Size-increasing support function ψ**(l)**.
Maximum fragment size **maxL**.
Constant needed to store the temporary fragments **k**.
Minimum support, **min_sup**.

Output:
Feature set, **F**.
Temporary feature set, **T**.

Method:
1. let $\mathbf{F} = \{ \mathbf{f}_\phi \}$, $\mathbf{T} = \{ \mathbf{t}_\phi \}$, $\mathbf{D}_{f_\phi} = \mathbf{D}$, and $\mathbf{l} = 0$
2. while $\mathbf{l} \leq \mathbf{maxL}$ do
3. for each fragment **x**, which is discriminative having size **l** do
4. if **x** is frequent then
5. $\mathbf{F} = \mathbf{F} \cup \{\mathbf{x}\}$
6. else if $\mathbf{x} \in$ set of previous $\mathbf{k} \times$ **min_sup** graphs
7. $\mathbf{T} = \mathbf{T} \cup \{\mathbf{x}\}$
8. $\mathbf{l} = \mathbf{l} + 1$
9. return **F**, **T**

In the process of keeping semi-frequent fragments, only fragments from last **constant** $\times$ **min_sup** graphs are stored. These fragments are stored regardless of their being frequent or not. If the minimum support for a database is **min_sup**, then all fragments available from the last **k** $\times$ **min_sup** graphs of the database are kept. Here, **k** is an integer constant, that is set to a value prior to the algorithm execution according to the assumed randomness of graphs in the database. Setting the correct value of **k** is a vital point of improvement for EGDIM. It is because the more the value of **k** is, the more EGDIM can cope up with random occurrences of graphs in database.

4.2 Index construction

Here substantial time is wasted in graph isomorphism test, which is an NP-complete problem. We need to do this test for finding the graphs in the database which contain the fragment as their subgraph. For reducing this time, a new method, **canonical labelling**, is introduced. In this

method, a graph is translated into a sequence.

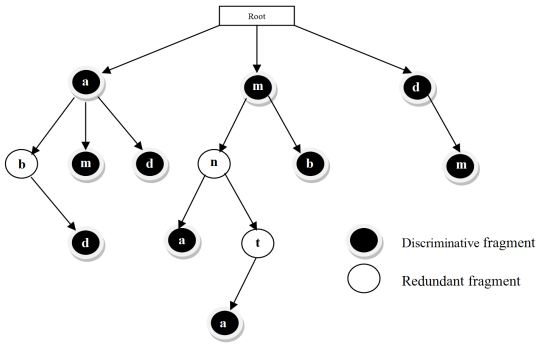

Figure 2: Prefix tree

For canonical labelling, **DFS code** is introduced. If two graphs are isomorphic, they will share the same minimum DFS code. By using the minimum DFS code, each fragment can be mapped into an edge sequence of discriminative fragments in a prefix tree. This tree is also referred as *gIndex* tree. The prefix tree records all **n**-sized discriminative fragments in level **n**. Figure 2 shows an example prefix tree.

In the index tree, code **s** is an ancestor of $\mathbf{s}'$ if and only if **s** is a prefix of $\mathbf{s}'$. In the prefix tree, all discriminative nodes as well as some redundant nodes are present. It increases the availability of a query fragment. All leafs of the tree are discriminative nodes and each discriminative node has an id list.

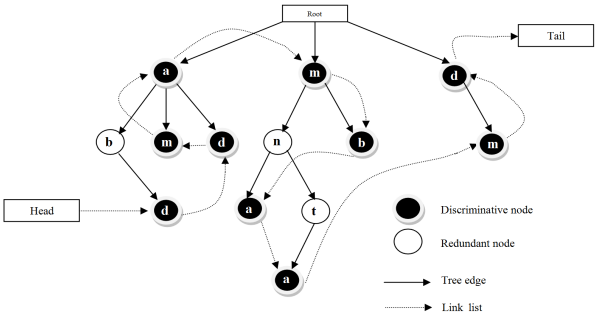

Figure 3: Prefix tree with linked nodes according to the access time

For storing the redundant nodes, **Apriori pruning** can be made. The Apriori pruning states that, if a fragment is not frequent then there is no need to check for its supergraphs. The prefix tree is implemented using a hash table for quickly locating a fragment and retrieve their id lists. Both discriminative and redundant nodes are hashed. Given a hash function **h**, canonical labelling function **c**, and a graph **g**, **h(c(g))** is called **graphic hash code**. Since two isomorphic graphs, **g** and $\mathbf{g}'$ have the same canonical labeling, $\mathbf{h}(\mathbf{c}(\mathbf{g})) = \mathbf{h}(\mathbf{c}(\mathbf{g}'))$. Graphic hash code can help the process of quickly locating a fragment in the perfix tree.

In the implementation, another task is done in this section. The task is, a **link list** between the discriminative nodes is maintained, as shown in figure 3, according to their update sequence. By using this link list, it can easily go from one discriminative node to the next one. This link list maintains order according to their access time. Using this link, fragments that are not used for a long time are removed from the prefix tree. This helps in the dynamic change of the tree.

4.3 Search

For a query graph, **q**, like *gIndex*, EGDIM also enumerates its fragments up to a maximum size and searches matches for them in the index tree. Then it generates a set by intersecting the sets of id lists associated with these fragments. Thus it generates the candidate set, $\mathbf{C_q}$, with the respective graphs of the id list. Apriori pruning and hashing is used in this section.

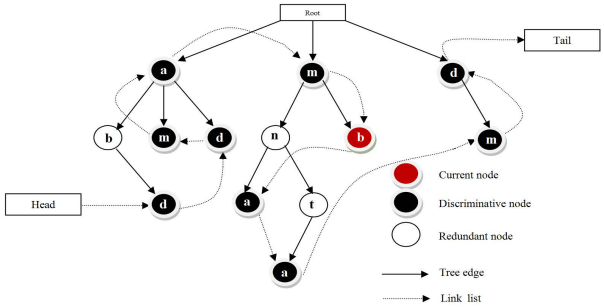

Figure 4: Prefix tree

The candidate set only tells the possible graphs that may contain the query graph. So it is necessary to check whether graphs in the candidate set are actually supergraphs of the query graph. The simplest approach is to check the candidate graphs one by one and run subgraph isomorphism test. Another approach is to record all the embeddings of paths in a graph database, rather than doing real subgraph isomorphism test. But in many cases, same time is needed in this approach as the previous approach. Moreover it is much more complex to implement. For the sake of simplicity, the first approach is implemented.

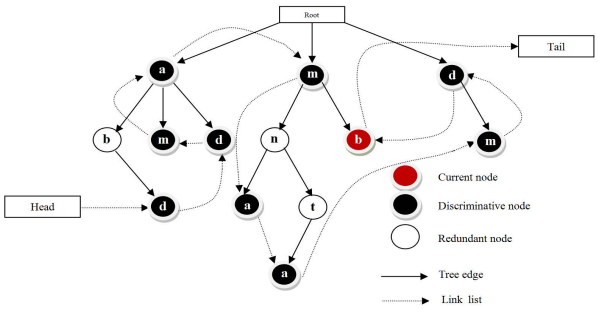

Figure 5: Prefix tree after updating link list

4.4 Verification

Verification approach in EGDIM is almost same as the verification approach used in *gIndex* algorithm. The only exception is that, whenever a discriminative fragment is found, the current time in that node is stored as it's last access time and link list is updated. To maintain track of time, a **global time counter** is kept. Value of this counter is incremented after any kind of instruction execution. For example, if we find a discriminative node which has a current access time

of 10, as shown in figure 4, and value of the global time counter is 22, the value of last access time of that node will be updated. It is clear that, for each time a nodeŠs access time counter is updated, it will surely hold the largest value so far. Using this method, it is very easy to update the link list in O(1) time complexity. The updated link list for the example is shown in figure 5.

4.5 Maintenance

In the *gIndex* algorithm, maintenance section only updates the graph id lists of discriminative nodes. But in EGDIM all the fragments available from the last **k** × **min_sup** graphs in the database are kept in a temporary section. When inserting a new fragment, it is checked if this fragment became frequent with respect to the temporary stored data. Figure 6 illustrates the idea. If the new fragment becomes frequent, then we add it to the prefix tree. The maintenance algorithm is shown in algorithm 3.

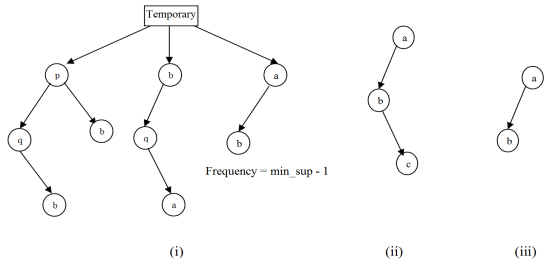

Figure 6: (i) Temporary fragment set (ii) Current data (iii) New frequent fragment

Another improvement, that is proposed in the maintenance section, is to check the link list for storing the last access time of fragments. If the oldest node is not accessed for a certain period of time, a defined threshold time, then it is removed from the link list as well as from the prefix tree. The algorithm keeps checking the link list as long as a node found in the link list which has been accessed within the threshold time period. This threshold value has to be defined at the beginning of processing for a certain data set.

Algorithm 3. Maintenance algorithm.

Input:
Feature set **F**.
Temporary feature set **T**.
Link list head **Head**.
Link list tail **Tail**.
Threshold time **Q**.

Method:
1. for each fragment **x**
2. if **x** ∈ **F** then
3. update **x.graph_id_list**
4. else
5. **T** = **T** ∪ {**x**}
6. if size of **x.graph_id_list** ≥ **min_sup** then
7. **F** = **F** ∪ **x**
8. **T** = **T** - **x**
9. while **current_time - Head.last_access_time** >**Q**
10. **Head** = **Head.next**

Let us give an example. Assume the access threshold time is 50, and current global time is 75. In figure 7, we see that the last access time for the first discriminative fragment is 24. So this fragment is not accessed for 75 - 24 = 51 time units. This value exceeds the threshold time. So this fragment will be removed.

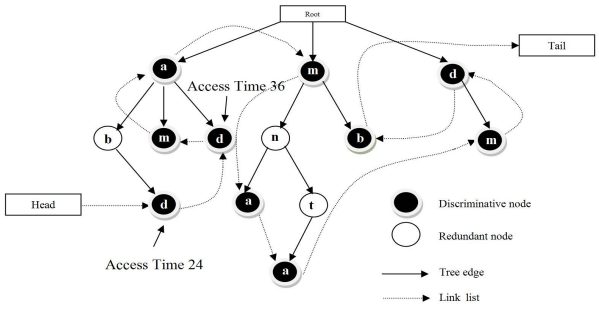

Figure 7: Prefix tree with discriminative fragment's last access time

This removal will free some space which is needed for storing the temporary fragment. After removing the unused fragment, the prefix tree looks like the one in figure 8. Thus the second approach complements the first approach, the need for extra space for storing temporary fragments.

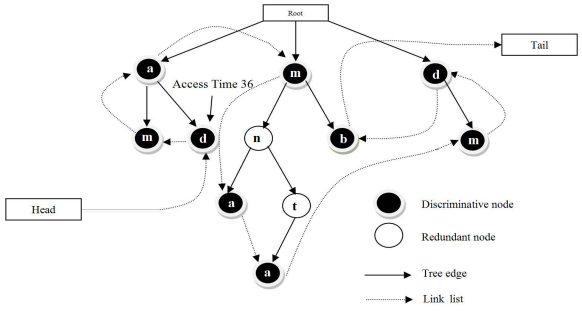

Figure 8: Updated prefix tree

We worked on improving the *gIndex* algorithm for producing better query processing for evolving database. To achieve this, in EGDIM mainly the feature selection part and the maintenance part of the *gIndex* algorithm are updated.

5. EXPERIMENTAL RESULTS

In this chapter, we will present the performance evaluation of EGDIM. We will focus on several scenarios to compare EGDIM with *gIndex* algorithm for mining frequent graphs from a graph database. We will also consider cases where the algorithm starts with a small size of initial database. We will show the experimental results on various scale updates over the initial database and present a performance comparison between the *gIndex* algorithm and EGDIM. Finally, we will discuss about some tuning of our used constants for

storing extra fragments. Memory space concern will also be briefly discussed. At the end of this chapter, we will try to analysis the cost of the algorithm using the cost analysis model presented in section 2.3.

5.1 Comparison of gIndex, EGDIM and Actual Result

In this section, we will try to show a number of scenarios and present some experimental results using graphs to show the comparison between the *gIndex* algorithm, EGDIM and the actual result. We worked on chemical data set collected from [22]. The data set was fed to the algorithms sequentially to make them behave as evolving data set. We also applied a brute force method for finding the actual result. We performed the experiment on various values for the minimum support. We tested the algorithm using several queries. Given a query, the performance was measured with the number of supergraphs answered by EGDIM and *gIndex*.

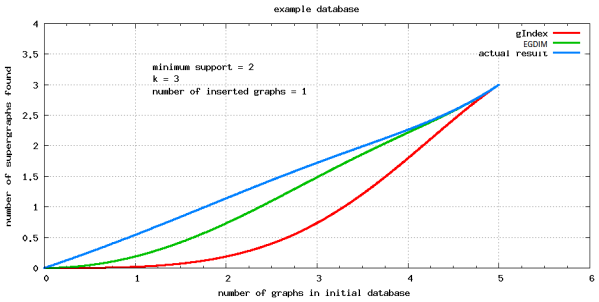

Figure 9: Comparison between *gIndex*, EGDIM and actual result after inserting one new graph into the initial database for the same query

Figure 9 shows an experimental result. Here, the initial database size was different for different test sets. As the size of initial database increases, both the *gIndex* algorithm and EGDIM converges to the actual result. But initially, for very small size of initial database, EGDIM outperforms the *gIndex* algorithm. For later cases, EGDIM performs at least as much as *gIndex* algorithm but no less than it. We set the value of **k**, a constant needed to store the temporary fragments in EGDIM, to 3 and minimum support was set to 2. We inserted one single graph to the initial database and showed that *gindex* does not perform well enough where EGDIM copes up with the change of the database.

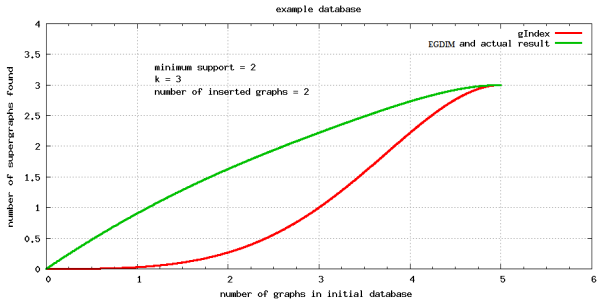

Figure 10: Comparison between *gIndex*, EGDIM and actual result after inserting two new graphs into the initial database for the same query

Figure 10 shows another test experiment result. The only difference from figure 9 is two new graphs are inserted into the initial database. Like in the previous experiment, in this experiment, also, EGDIM outperforms the *gIndex* algorithm. Even in this case, EGDIM produces the accurate result and graph for the actual result and our result is the same in this experiment.

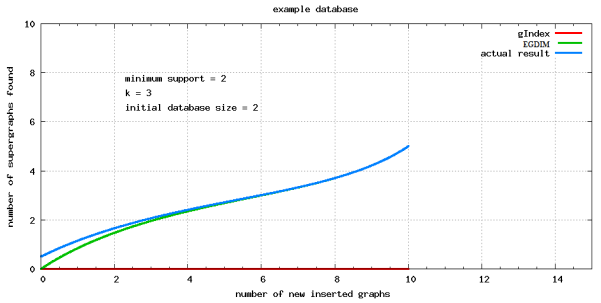

Figure 11: Comparison between *gIndex*, EGDIM and actual result for various numbers of new inserted graphs into the initial database for a query

For fixed size initial database, the next two experiments are performed. Figure 11 shows result for a scenario where the number of graphs in the initial database is 2. As number of new inserted graphs increases, EGDIM produces result almost as good as the real result for an individual query. But *gIndex* algorithm doesn't even find the query. This is because the fragments necessary to answer the query were not frequent. It proves that the *gIndex* algorithm doesn't cope up well with the new inserted graphs. As the size of initial database is very small, *gindex* can not assume the characteristics of the database. But EGDIM easily copes up with the evolving database.

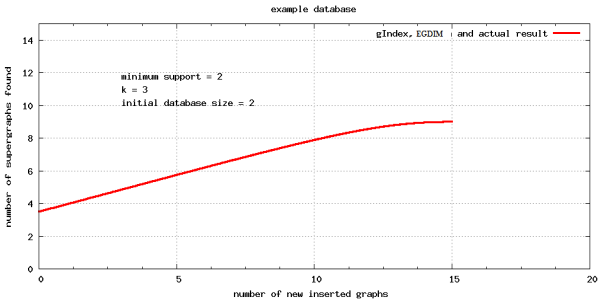

Figure 12: Comparison between *gIndex*, EGDIM and actual result for various numbers of new inserted graphs into the initial database for another query

Figure 12 shows another experimental result similar to the one in the previous example. In this example, all three results are same. This ensures that, in worst case, EGDIM works as good as *gIndex* algorithm but never degrades than it.

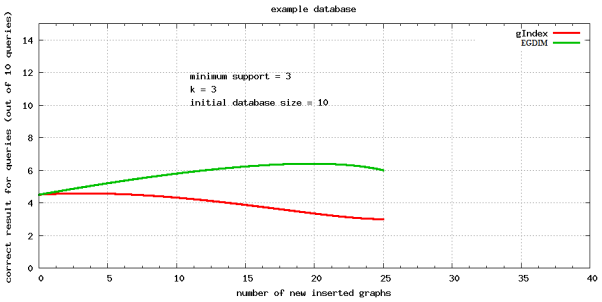

Figure 13: Comparison between *gIndex* and EGDIM for various numbers of new inserted graphs into the initial database for different sets of queries

Figure 13 shows an experimental result where a comparison is shown between the *gIndex* algorithm and EGDIM using a different metric. We used different sets of queries and noted down the number of queries properly answered by each of the algorithms. This experiment also shows that, as the number of newly inserted graphs increases, EGDIM easily copes up with the evolving database. But *gIndex* fails to cope up with the evolving database and so it degrades more and more with the increasing rate of newly inserted random graphs.

This section mainly focuses on handling the evolving database. Here, in all experiments, EGDIM performed at least as accurate as the *gIndex* algorithm and in most of the cases, it outperformed the *gIndex* algorithm.

5.2 Performance for Different Values of Constants

In this section, we will observe two things. The first thing we will observe is, for various values of **k**, the constant to decide how many fragments will be stored from the new updates, EGDIM responses in different ways. Figure 14 shows this impact of the value of **k** on EGDIM. As we increase the value ok **k**, more fragments will be stored in the temporary memory. As a result the probability of answering the query properly will increase. This performance comes with the cost of memory. The more the value of **k** is, the more we are using memory. So it is necessary to select a suitable value of **k** that will enable this algorithm to process properly and respond accurately.

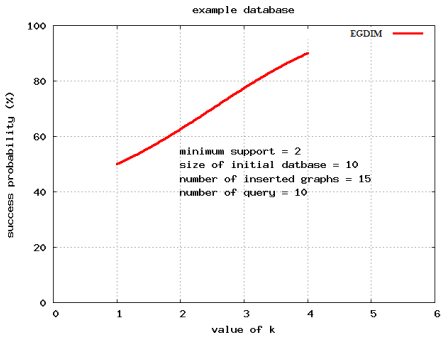

Figure 14: Performance of EGDIM for various values of k for a fixed number of queries

Another important issue to notice is, if we store more fragments of each graph in the database, it also increases the probability of proper response to the query. Figure 15 shows an experimental result on various numbers of fragments storing for each graph in the database. As expected, when we increase the amount of fragments stored, EGDIM's response tends to 100% accuracy. But it is memory space consuming to store too much fragments. Here we have to trade off also, like selecting the value of **k**.

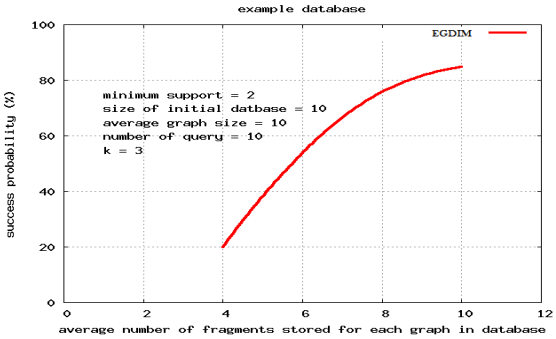

Figure 15: Performance of Our algorithm for various numbers of fragments stored for each graph in database for a fixed number of queries

It is clear from this section that, tuning the proper value for **k** and number of fragments to be stored in the memory for each graph is a vital and sensitive issue. The algorithm's performance improves with the cost of memory space. So properly selecting these values plays a vital role in the performance issue of EGDIM.

5.3 Cost Analysis

We will now calculate the cost of EGDIM. The main part we have focused in EGDIM is the candidate set generation phase. If we can properly generate the candidate set, it is highly probable that we will provide correct result. EGDIM focuses on providing correct result with the cost of memory

space.

Although it is always appreciable to optimize memory usage, it was one of the primary goals of *gIndex* too, often we have to sacrifice it because of the precise result. There is a trade off between memory space usage and performance accuracy. The main advantage of EGDIM over the *gIndex* algorithm is, it can work with evolving data. Our experiment also proves this fact that EGDIM performs better with evolving data than *gIndex* algorithm.

As we generate more graphs in candidate set than the *gIndex* algorithm for providing accuracy, our algorithm may perform slower or take a bit more time than *gIndex* algorithm due to the overhead for handling evolving data. But it will never give a wrong result when *gIndex* will provide a right one. The cost is increased in EGDIM for the adaptability of EGDIM with random and dynamic data updates. The subgraph isomorphism test is same in *gIndex* algorithm and our algorithm. The only difference is our algorithm keeps track of more fragments that comes with a cost of more memory space usage.

EGDIM performs better in the field of evolving databases than *gIndex* algorithm. We showed this result with the help of small sized initial database and later sequential updates of the database. In short, we can say that with a small cost of memory usage, we provide advancement over the *gIndex* algorithm for working with evolving data. This surely serves the field of dynamic or evolving database.

6. CONCLUSIONS

In this research work, an idea to improve the existing *gIndex* algorithm is developed to make it work for evolving database. *gIndex* algorithm was unable to work properly with small sized initial database and also was unable to cope up with dynamic change in database updates. In EGDIM, some extra memory space is used to store some fragments of new updates of database. In this paper, we discussed the idea elaborately. This idea helps in the process of identifying newly generated frequent fragments. As a result, the method easily adjusts with evolving or dynamic database. The idea to maintain a link list for keeping track of the access time of the fragments is also discussed. This information identifies the least recently accessed fragment in the index tree. The process of deletion of unnecessary fragments were also presented. This process helps in providing extra memory space necessary for storing extra fragments. EGDIM performs better in the field of evolving databases than *gIndex* algorithm. This result is showed with the help of small sized initial database and later sequential updates of the database. Two points can be focused for future improvements over our algorithm. First, our algorithm does not guarantee that the newly generated fragments are discriminative. It is beneficiary to only generate the discriminative fragments while storing new fragments from database updates. This constraint will save some memory space and minimize the search time, because, in that case, the prefix size will be shorter than the current prefix size. Second, if we can implement a mechanism to adjust the the accuracy of proper fragment removal, that will be very useful for properly maintaining the memory space usage. We can also implement some learning tools for these adjustments.Finally, we can conclude that, although EGDIM's performance may be slower than *gIndex* in a few cases, it will always guarantee more accurate result than *gIndex*.

7. REFERENCES

[1] Data mining. Website. `http://en.wikipedia.org/wiki/Data_mining`.

[2] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. pages 721Ű–724, Maebashi, Japan, 2002. 2002 Int. Conf. on Data Mining (ICDMŠ02).

[3] Jun Huan, Wei Wang, and Jan Prins. Spin: Mining maximal frequent subgraphs from graph databases. In *In KDD*, pages 581–586, 2004.

[4] X. Yan, P. S. Yu, and J. Han. Graph Indexing Based on Discriminative Frequent Structure Analysis. ACM Transactions on Database Systems, Vol. V, No. N, August 2005.

[5] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. Kluwer Academic Publishers, 2004.

[6] N Vanetik, E Gudes, and S E Shimony. Computing frequent graph patterns from semistructured data. Maebashi, Japan, 2002. 2002 Int. Conf. on Data Mining (ICDMŠ02).

[7] Michihiro Kuramochi and George Karypis. Frequent subgraph discovery. In *Proceedings of the 2001 IEEE International Conference on Data Mining*, ICDM '01, pages 313–320, Washington, DC, USA, 2001. IEEE Computer Society.

[8] Liping Wang, Qing Li, Na Li, Guozhu Dong, and Yu Yang. Substructure similarity measurement in chinese recipes. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 979–988, New York, NY, USA, 2008. ACM.

[9] Christian Borgelt. Mining molecular fragments: Finding relevant substructures of molecules. In *In Proc. of 2002 IEEE International Conference on Data Mining (ICDM*, pages 51–58. IEEE Press, 2002.

[10] J. R. Ullmann. Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. *J. Exp. Algorithmics*, 15:1.6:1.1–1.6:1.64, February 2011.

[11] James Cheng, Yiping Ke, and Wilfred Ng. Efficient query processing on graph databases. *ACM Trans. Database Syst.*, 34:2:1–2:48, April 2009.

[12] Dipali Pal and Praveen R. Rao. A tool for fast indexing and querying of graphs. In *Proceedings of the 20th international conference companion on World wide web*, WWW '11, pages 241–244, New York, NY, USA, 2011. ACM.

[13] S. Zhang, X. Gao, W. Wu, J. Li, and H. Gao. Efficient algorithms for supergraph query processing on graph databases. page 2. Springer Science, 2009.

[14] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 857–872, New York, NY, USA, 2007. ACM.

[15] B Cooper, N Sample, M Franklin, G Hjaltason, M, and Shadmon. A fast index for semistructured data. Roma, Italy, 2001. 2001 Int. Conf. on Very Large Data Bases (VLDBŠ01).

[16] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha. Enhancing graph database indexing by suffix tree structure. In *Proceedings of the 5th IAPR*

international conference on Pattern recognition in bioinformatics, PRIB'10, pages 195–203, Berlin, Heidelberg, 2010. Springer-Verlag.

[17] D Shasha, J Wang, and R Giugno. Algorithmics and applications of tree and graph searching. page 39Ű52, Madison, WI, 2002. 21th ACM Symp. on Principles of Database Systems (PODSŠ02).

[18] J. Han and M. Kamber. *Data Mining Concepts and Techniques*. Morgan Kaufmann, second edition, 2006.

[19] C. F. Ahmed, S. K. Tanbeer, B.-S Jeong, and Y.-K. Lee. Mining high utility patterns in incremental databases. In *The 3rd ACM ICUIMC, Suwon, South Korea*, pages 656–663, January 2009.

[20] Graph isomorphism. Website. `http://en.wikipedia.org/wiki/Graph_isomorphism`.

[21] Subgraph isomorphism problem. Website. `http://en.wikipedia.org/wiki/Subgraph_isomorphism_problem`.

[22] Xifeng yan. Website. `http://www.cs.ucsb.edu/~xyan/`.