An Algorithm towards Indexing Evolving Graph Databases

Anna Fariha, Shariful Islam, Chowdhury Farhan Ahmed, Byeong-Soo Jeong

Abstract— **In the field of extracting valuable information or knowledge from large database, data mining is a powerful tool. Frequent substructure mining, also denoted by graph mining, requires searching for frequent substructures or sub-graphs from large structured or graph database. *gIndex* is a more robust algorithm for mining graphs, especially for indexing frequent sub-graph patterns over previous Apriori or path based graph mining algorithms. *gIndex* finds the super-graphs of the given query graph from the graph database. The main specialty of *gIndex* is to maintain an index of graph database according to discriminative fragments. This research work proposes a further improvement over the existing *gIndex* algorithm. The proposed algorithm in this paper is specially designed for handling sudden change in database graph patterns. The algorithm, EGDIM, is capable of processing queries in dynamic and evolving database which *gIndex* can not handle. To achieve the same performance for evolving graph databases, more information is stored in the index data structure to quickly answer the graph query, discarding the unnecessary graphs. EGDIM also ensures a good running time for processing graph queries in the evolving graph databases.**

***Index Terms*— Data mining, Knowledge discovery, Graph mining, Indexing.**

I. INTRODUCTION

The process of discovering knowledge and determining patterns and their relationships by analyzing large database is called data mining. Frequent patterns are patterns (such as itemsets, subsequences, or substructures) that appear in a data set frequently [10, 11]. **Apriori** algorithm is an innovative way to find association rules on large scale databases, allowing implication outcomes that consist of more than one item. Despite the simplicity of Apriori algorithm, it is costly for mining frequent itemsets in large database, so **FP-growth** [10] approach is proposed for better performance.

Structured data mining is the process of finding and extracting useful information from semi-structured data sets. Frequent substructure mining is a special kind of frequent pattern mining [1, 2]. Graphs are widely used data structures for representing schema less data in biological, chemical and other important fields. A new and special field of data mining, **Graph Mining**, is used to mine data, or sub-graphs, from complicated structures. Graph represents not only the properties of the elements, but also the relationship between the elements. The main challenge to solve the problem of processing sub-graph queries on a database, by finding the set of graphs in the database that are super-graphs of the query, is the size of the database and the NP completeness of sub-graph isomorphism problem [3, 4, 5, 6].

In Graph mining, we have to be able to search in the database for the presence of a query graph. In the core of graph mining problem, there lies the **sub-graph query problem**. **Apriori**-based frequent substructure mining algorithms, which proceed in a bottom-up manner and adopt a level-wise mining methodology, 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 like ***gSpan*** [1] and ***GraphGrep*** [9]. ***gSpan*** algorithm and ***GraphGrep,*** which are path based graph mining algorithms, were not efficient to search the whole database for answering the graph query. For answering graph queries faster, an index of the graph database can be [7, 8] kept. Both frequent pattern and path or substructure can be used as indexing feature. For faster query search, an improved version of graph mining algorithm, ***gIndex*** [2] is proposed, which uses frequent substructure as its indexing feature. ***gIndex*** builds graph indices to help processing graph queries and retrieve related graphs and is used in indexing sequences, trees, and other complex structures.

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

May 10, 2012

Anna Fariha is an MS student, Department of Computer Science and Engineering, University of Dhaka, Bangladesh (phone: +880-1733-736-670; e-mail: purpleblueanna@gmail.com).

Shariful Islam is an MS student, Department of Computer Science and Engineering, University of Dhaka, Bangladesh (phone: +880-1744-439-165; e-mail: tulip.du@gmail.com).

Chowdhury Farhan Ahmed is working as an Associate Professor, Department of Computer Science and Engineering, University of Dhaka, Bangladesh (phone: +880-1914-249515; e-mail: farhan@cse.univdhaka.edu).

Byeong-Soo Jeong is working as a Professor, Department of Computer Engineering, Kyung Hee University, 1 Seochun-dong, Kihung-gu, Youngin-si, Kyunggi-do, 446-701, Republic of Korea (e-mail: jeong@khu.ac.kr).

DOI: 10.5176/2010-2283_2.2.161

fragments for compactness, **discriminative fragments as indexing feature** and **Apriori pruning**, that enables ***gIndex*** to optimize its efficiency.

Although paths are easy to manipulate and taking paths as the indexing feature leaves the indexing space predefined, frequent substructures, being more expressive in representing the properties of a graph, preserving the structural information, are ideal candidate for the indexing feature. The problem of ***gIndex*** is, it only takes into account those frequent substructures which are discriminative enough and assumes that the database is stable to updates. Performance of ***gIndex*** degrades due to random growth of database, because it does not take into account any nearly frequent substructures.

This research extends the existing ***gIndex*** algorithm to achieve an even better performance for evolving database. A new method, **EGDIM** (**E**volving **G**raph **D**atabase **I**ndexing **M**ethod), is proposed, which keeps track of the recent semi-frequent substructures having support close to **minimum support**. These semi-frequent substructures are considered as indexing features when their support reaches the minimum support. An idea to remove the least recently encountered discriminative frequent sub-structure from the index structure is also proposed for memory efficiency.

II. PRELIMINARIES

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

Definition 1. 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 [12].

Definition 2. Sub-graph isomorphism problem: The sub-graph isomorphism problem is a computational task in which, given two graphs A and B, one must determine whether A contains a sub-graph that is isomorphic to B [13].

Definition 3. 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 sub-graph [10]. $|Dg|$ is the number of graphs in D where g is a sub-graph [2].

Definition 4. 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 5. Frequent fragment: A frequent fragment (or graph, sub-structure, sub-graph) is a graph whose support is no less than ***min_sup*** [10].

Definition 6. DFS code and DFS lexicographic order: ***DFS code*** is an edge sequence for subscripting graphs to build an order among different graph sub-scripting.

Definition 7. 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 8. Feature set: The set of sub-graphs 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, $D_f = \{g_i \mid f \subseteq g_i, g_i \in D_g\}$ [2].

Definition 9. 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, $C_q = \cap_f D_f\ (f \subseteq q \text{ and } f \subseteq F)$; each graph in C_q contains all q's features in the feature set. Therefore, D_q is a subset of C_q [2].

Definition 10. Redundant fragment: Fragment x is redundant with respect to feature set F if $D_x \approx \cap_{f \in F \wedge f \subseteq x} D_f$ [2]. That means if a fragment's presence can be predicted by the presence of it's sub-graphs, then it is a redundant fragment.

Definition 11. Discriminative fragment: Fragment x is discriminative with respect to feature set F if $D_x \ll \cap_{f \in F \wedge f \subseteq x} D_f$ [2].

Definition 12. Prefix tree: It is an efficient tree data structure used for indexing graphs according to their prefixes after translating fragments into sequences.

III. EGDIM- EVOLVING GRAPH DATABASE INDEXING METHOD

Although ***gIndex*** started a new way to answer the sub-graph queries through an indexing approach, it has some drawbacks. The main drawback of ***gIndex*** algorithm is that, when it starts with small size of initial database, it 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. Although **EGDIM** can handle dynamic or evolving data in the updates of the database, the main drawback of this approach was, 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, in **EGDIM**, 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.

Five phases of **EGDIM** are similar to ***gIndex*** algorithm but adds some improvements for performing well in dynamic database. The first two phases are for building the initial index tree from the given database. The next three phases are for incremental updates. We will describe the phases with appropriate example in this section for the sample graph database shown in **figure 1**.

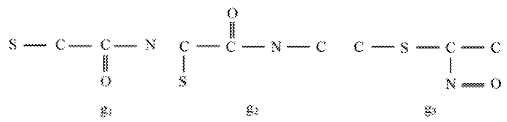


Figure 1: Example graph database

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 1** provides the pseudo code for the discriminative fragment selection phase.

Algorithm 1. Feature selection algorithm

Input:

Graph database, ***D***.
Discriminative ratio, $\gamma_{\mathbf{min}}$.
Size-increasing support function, $\psi(l)$.
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:

let $F = \{f_\phi\}$, $T = \{t_\phi\}$, $D_{f\phi} = D$, and $l = 0$
while $l \leq maxL$ do
    for each fragment x, discriminative and having size l do
        if x is frequent then
            $F = F \cup \{x\}$
        else if x $\in$ set of previous ***k*** x ***min_sup*** graphs
            $T = T \cup \{x\}$
$l = l + 1$
return ***F, T***

In the process of keeping semi-frequent fragments, only fragments from last ***constant*** x ***min_sup*** graphs 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*** x ***min_sup*** graphs of the database are kept. Here, ***k*** is an integer constant, 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** algorithm. It is because the more the value of ***k*** is, the more **EGDIM** algorithm can cope up with random occurrences of graphs in database.

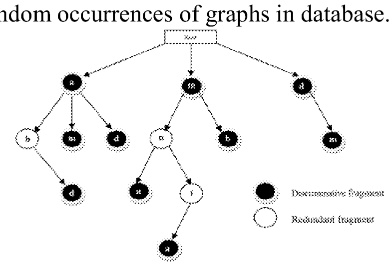


Figure 2: Prefix tree

In **index construction**, **canonical labeling** is used. For canonical labeling, **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 ***s'*** if and only if ***s*** is a prefix of ***s'***. In the prefix tree, all discriminative nodes and some redundant nodes are present. All leafs of the tree are discriminative nodes and each has an **id list**.

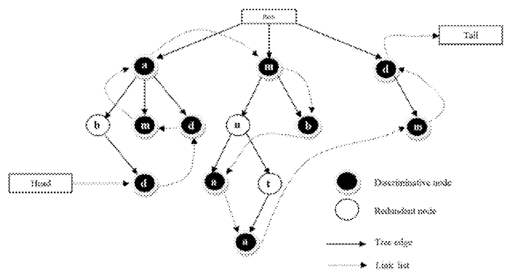


Figure 3: Nodes are linked according to the access time

For storing the redundant nodes, **Apriori pruning**, which states that, if a fragment is not frequent then there is no need to check for its super-graphs, is done. The prefix tree is implemented using **graphic hash code.** Given a hash function ***h***, canonical labeling function ***c***, and a graph ***g***, $h(c(g))$ is called **graphic hash code**. Since two isomorphic graphs, ***g*** and ***g'*** have the same canonical labeling, so, $h(c(g)) = h(c(g'))$.

In the implementation, a **link list** between the discriminative nodes is maintained, according to their update sequence, as shown in **Figure 3**. This link list maintains order according to their access time. To accommodate dynamic change in the tree, fragments that are not used for a long time are removed from the prefix tree

In the **search** section, for a query graph, ***q***, like ***gIndex***, **EGDIM** algorithm 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 and generates the candidate set, C_q, with the respective graphs of the id list. **Apriori pruning** and **hashing** is used in this section.

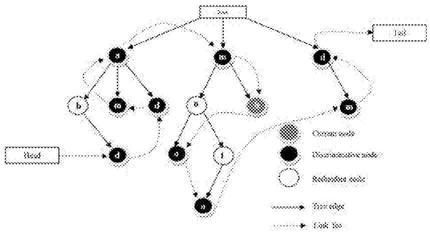


Figure 4: Prefix Tree

To check whether graphs in the candidate set are actually super-graphs of the query graph, the candidate graphs are checked one by one and sub-graph isomorphism test is run.

Verification approach in **EGDIM** algorithm is almost same as the verification approach used in ***gIndex*** algorithm except, whenever a discriminative fragment is found, the current time in that node is stored as its **last access time** and link list is updated. To maintain track of time, a **global time counter** is kept, which 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. For each time a node's access time counter is updated, it will surely hold the largest value so far, so, 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**.

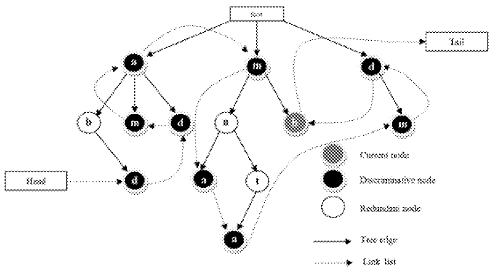


Figure 5: Prefix tree after updating link list

In the **maintenance** section for insertion, the fragments available from the last ***k x 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, illustrated in **Figure 6**. If the new fragment becomes frequent, then we add it to the prefix tree. The maintenance algorithm is shown in **algorithm 2**.

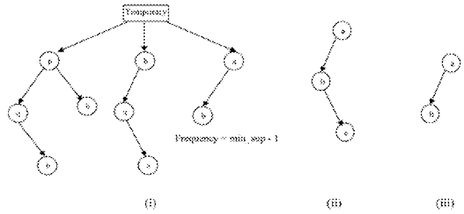


Figure 6: (i) Temporary fragment set (ii) Current data (iii) New frequent fragment

Another improvement, 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 **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 is found in the link list which has been accessed within the threshold time period.

Algorithm 2. Maintenance algorithm

Input:

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

Method:

```
for each fragment x
    if x ∈ F then
        update x.graph_id_list
    else
        T = T ∪ {x}
        if |x.graph_id_list| ≥ min_sup then
            F = F ∪ {x}
            T = T - x
while cur_time - Head.last_access _time > Q
    Head = Head.next
```

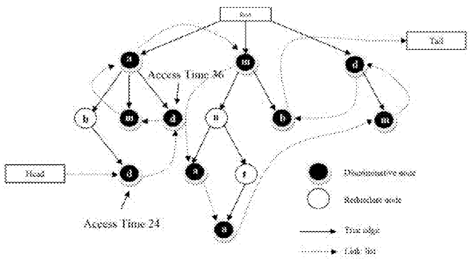


Figure 7: Prefix tree with discriminative fragment's last access time

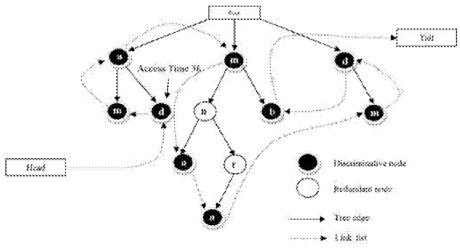


Figure 8: Updated prefix tree

For example, if the access threshold time is ***50***, and current global time is ***75***, in **figure 7**, the last access time for the first discriminative fragment is ***24***. So this fragment is not accessed for ***75 - 24 = 51*** time units which exceeds the threshold time and this fragment will be removed. This removal will free some space which is needed for storing the temporary fragments. After removing the unused fragment, the prefix tree looks like the one in **figure 8**. This fulfills the need for extra space for storing temporary fragments.

IV. EXPERIMENTAL RESULTS

In this section, performance evaluation of **EGDIM** is focused using several scenarios to compare **EGDIM** algorithm with ***gIndex*** algorithm for mining frequent graphs from an evolving graph database and cases where the algorithms start with a small size of initial database are also discussed.

We worked on chemical data set collected from [14]. The data set was fed to the algorithms sequentially to make them behave as evolving data set. The experiment on various values for the minimum support was performed and tested using several queries. Given a query, the performance was measured with the number of super-graphs answered by the algorithms.

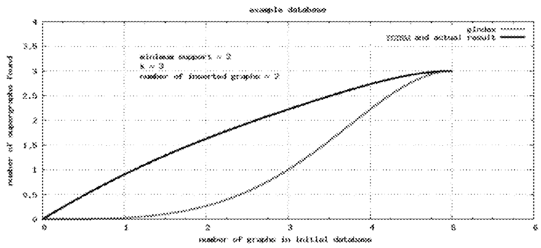


Figure 9: Comparison between ***gIndex***, **EGDIM** algorithm and actual result after inserting two new graphs into the initial database for the same query

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*** and **EGDIM** algorithm converges to the actual result. But, initially, for very small size of initial database, **EGDIM** algorithm outperforms ***gIndex*** algorithm. For later cases, **EGDIM** algorithm performs at least as much as ***gIndex*** algorithm but no less than it. The value of ***k***, a constant needed to store the temporary fragments in **EGDIM** algorithm, was set to ***3*** and minimum support was set to ***2***. Two new graphs were inserted to the initial database and it is showed that ***gIndex*** does not perform well enough where **EGDIM** algorithm copes up with the change of the database.

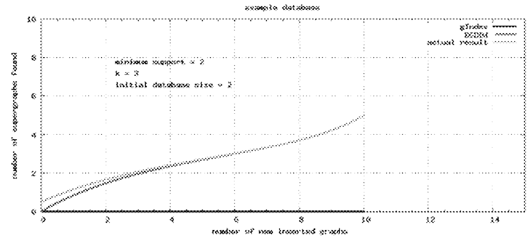


Figure 10: Comparison between ***gIndex***, **EGDIM** algorithm actual result for various numbers of new inserted graphs into the initial database for a query

For fixed size initial database, the next experiment is performed. **Figure 10** shows result for a scenario where the number of graphs in the initial database is ***2***. As number of new inserted graphs increases, **EGDIM** algorithm produces result almost as good as the real result for an individual query where ***gIndex*** algorithm doesn't even find the query. As the size of initial database is very small, ***gIndex*** can not assume the characteristics of the database. But **EGDIM** algorithm easily copes up with the evolving database.

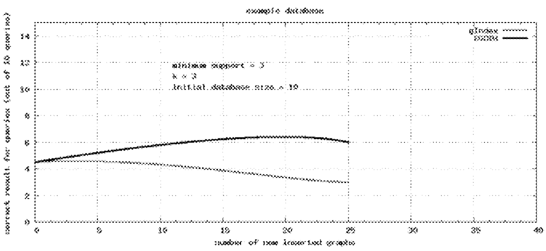


Figure 11: Comparison between ***gIndex*** and **EGDIM** algorithm for various numbers of new inserted graphs into the initial database for different sets of queries

Figure 11 shows an experimental result where a comparison is shown between the ***gIndex*** algorithm and **EGDIM** algorithm using different sets of queries, and properly answered number of queries. This experiment also shows that, as the number of newly inserted graphs increases, **EGDIM** algorithm easily copes up with the evolving database.

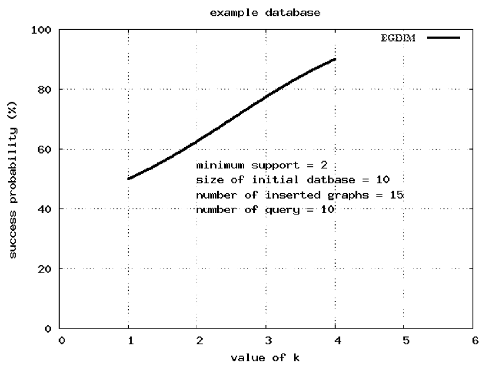


Figure 12: Performance of **EGDIM** algorithm for various values of ***k*** for a fixed number of queries

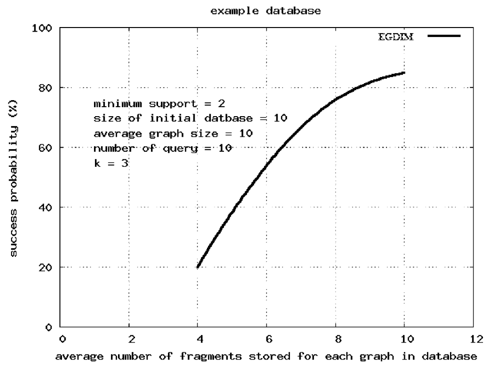


Figure 13: Performance of **EGDIM** algorithm for various numbers of fragments stored for each graph in database for a fixed number of queries

For various values of ***k***, the constant to decide how many fragments will be stored from the new updates, **EGDIM** algorithm responses in different ways shown in **Figure 12**. As the value of ***k*** increases, more fragments will be stored in the temporary memory and probability of answering the query properly will increase. As this performance comes with the cost of memory, it is necessary to select a suitable value of ***k*** that will enable **EGDIM** to perform optimally.

Another important issue is, if more fragments of each graph are stored, it also increases the probability of proper response to the query. **Figure 13** shows an experimental result on various numbers of fragments stored for each graph in the database. When the amount of fragments stored is increased, **EGDIM** algorithm's response tends to 100% accuracy. As it is memory consuming to store too much fragments, we have to face a trade-off in selecting the value of ***k***. 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 main part we have focused in **EGDIM** algorithm is the candidate set generation phase. **EGDIM** algorithm focuses on providing correct result with the cost of memory space. Experimental result also proves this fact that **EGDIM** algorithm performs better with evolving data than ***gIndex*** algorithm. It will never give a wrong result when ***gIndex*** will provide a right one.

V. Conclusion

In this research work, an idea to improve the existing ***gIndex*** algorithm is developed to make it work for evolving database. **EGDIM** algorithm, using some extra memory space, stores some fragments of new updates of database and easily adjusts with evolving databases, where ***gIndex*** algorithm was unable to work properly with small sized initial database and cope up with dynamic change in database updates. Maintaining a link list for keeping track of the access time of the fragments is necessary in **EGDIM**, because it identifies the least recently accessed fragment in the index tree. Deletion of unnecessary fragments, which helps in providing extra memory space necessary for storing extra fragments, were also presented in **EGDIM**. As the experiments reflect, **EGDIM** algorithm 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. Although **EGDIM** algorithm's performance may be slower than ***gIndex*** in a few cases, it will guarantee more accurate result than ***gIndex***.

References

[1] X. Yan and J. Han, "gSpan: Graph-based substructure pattern mining" in 2002 Int. Conf. on Data Mining (ICDM02), Maebashi, Japan, 2002, pp. 721-724.

[2] 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.

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

[4] J. Cheng, Y. Ke, and W. Ng, "Efficient query processing on graph databases", ACM Trans. Database Syst., pp. 34:2:1-2:48, April 2009.

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

[6] J. Cheng, Y. Ke, W. Ng, and A. Lu, "Fg-index: towards verification-free query processing on graph databases", in Proceedings of the ACM SIGMOD international conference on Management of data, pp. 857-872, 2007.

[7] B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon, "A fast index for semistructured data", in 2001 Int. Conf. on Very Large Data Bases (VLDB01), Roma, Italy, 2001.

[8] 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, Berlin, Heidelberg, 2010, pp. 195 - 203.

[9] D. Shasha, J. Wang, and R. Giugno, "Algorithmics and applications of tree and graph searching", 21th ACM Symp. on Principles of Database Systems (PODS02), pp. 3952, Madison, WI, 2002.

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

[11] S. K. Tanbeer, C. F. Ahmed, and B.S. Jeong, "Mining regular patterns in data streams", in DASFAA, 2010, pp. 399 – 413.

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

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

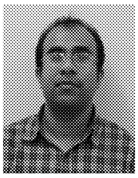
[14] X. yan. Website. http://www.cs.ucsb.edu/~xyan/. G. O. Young, "Synthetic structure of industrial plastics" in *Plastics*, 2nd ed. vol. 3, J. Peters, Ed. New York: McGraw-Hill, 1964, pp. 15–64.

Anna Fariha received her B.Sc degree in Computer Science and Engineering from the University of Dhaka, Bangladesh, in 2010. Currently she is pursuing her M.Sc degree in the Department of Computer Science and Engineering, University of Dhaka. Her research interests are in the areas of data mining and knowledge discovery.

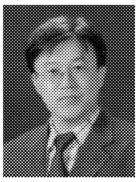
E-mail: purple.blue.anna@gmail.com

Shariful Islam received his B.Sc degree in Computer Science and Engineering from the University of Dhaka, Bangladesh, in 2010. Currently he is pursuing his M.Sc degree in the Department of Computer Science and Engineering, University of Dhaka. His research interests are in the areas of data mining and knowledge discovery.

E-mail: tulip.du@gmail.com



Chowdhury Farhan Ahmed received his B.S. and M.S. degrees in Computer Science from the University of Dhaka, Bangladesh in 2000 and 2002 respectively, and Ph.D. degree in Computer Engineering from Kyung Hee University, South Korea in 2010. From 2003-2004 he worked as a faculty member at the Institute of Information Technology, University of Dhaka, Bangladesh. Since 2004, he has been working as a faculty member in the Department of Computer Science and Engineering, University of Dhaka, Bangladesh. His research interests are in the areas of data mining and knowledge discovery.
E-mail: farhan@cse.univdhaka.edu



Byeong-Soo Jeong received his B.S. degree in Computer Engineering from Seoul National University, Korea in 1983, his M.S. degree in Computer Science from the Korea Advanced Institute of Science and Technology, Korea in 1985, and his Ph.D. in Computer Science from the Georgia Institute of Technology, Atlanta, USA in 1995. In 1996, he joined the faculty at Kyung Hee University, Korea where he is now a professor at the College of Electronics & Information. From 1985 to 1989, he was on the research staff at Data Communications Corp., Korea. From 2003 to 2004, he was a visiting scholar at the Georgia Institute of Technology, Atlanta. His research interests include database systems, data mining, and mobile computing.
E-mail: jeong@khu.ac.kr