DAGC: Identification and Filtration of Automorphic Graphs in Graph Databases

Aye Nwe Thaing
University of Computer Studies,
Yangon, Myanmar

Kyaw May Oo
University of Computer Studies,
Yangon, Myanmar

ABSTRACT
Graphs are the ubiquitous models for constructing both natural and human-made structures. Many practical problems can be represented by graphs. They can be used to model many applications such as physical, biological and social systems. With the emergence of these applications, developments of graph databases are very useful to store graph data. Due to the existence of noise (e.g., duplicated graphs) in the graph database, we investigate the problem of storing the same graphs in the single graph database. Therefore, detecting and eliminating of automorphic graphs in a graph database become an important research area. In this paper, we propose a novel DAGC algorithm to identify and removal of automorphic graph storing into the graph database using AdE index structure. AdE index structure incorporates graph structural information of each graph in the database. The computational time complexity is significantly reduced compared to canonical labeling approach used in most graph matching algorithms and F-GAF algorithm. This paper demonstrates the effectiveness and efficiency of the proposed method through experiments on various types of graphs.

Keywords
Graph Database, Graph Automorphism, Graph Mining, Canonical Adjacency Matrix, Chemical Compound

1. INTRODUCTION
Graphs are used to represent networks of communication, social network, data organization, and the flow of computation etc. Graph theory is also used to study molecules in chemistry and physics. In chemistry a graph can be used for building the model of a molecule, where vertices correspond to atoms and edges bonds. In statistical physics, graphs characterize connections between interacting parts of a system.

Graph isomorphism is a problem to determine whether given two graphs G_1 and G_2 are isomorphic, to find a mapping from a set of vertices to another set. Automorphism is a special case of graph isomorphism where the two graphs are identical, which means to find a mapping from a graph to itself. Subgrpah isomorphism is to find an isomorphism between G_1 and a subgraph of G_2. In other words, it is to determine if a graph is included in the other larger graph.

There are several different ways to solve graph isomorphism. Graph isomorphism can be solved starting from a single vertex in one graph; try to find a mapping to one of the vertices in the other graph that is consistent with the labeling [1]. Then, the vertices are added one by one until either finding a complete mapping or ending up with exhausting the search space using the same process. This approach can solve both graph and subgraph isomorphism problem.

To identify automorphism between graphs, a graph can be represented in many different ways, depending on the order of its edges or vertices. To get total order of graphs, canonical labeling can be used. A canonical label is a unique code of a given graph. Canonical codes should be always the same no matter how graphs are represented, as long as those graphs have the same topological structure and the same labeling of edges and vertices. The database contains the duplicated graphs if their canonical codes are identical [1,10].

In this paper,an efficient DuplicatedAutomorphicGraphCleaning (DAGC) algorithm is proposed that uses AdE structure for identifying and filtering automorphic graphs efficiently. AdE structure can also eliminate automorphic graphs getting stored into the graph database.

The rest of the paper is organized as follows. Section II represents the formal definitions and notations used for the proposed work. Section III discusses about the related work of graph isomorphism. Section IV discusses about the proposed work. Section V explains DAGC algorithm. Section VI talks about identifying duplicated automorphic graphs. Section VII presents the analysis and illustration of three techniques. Section VIII presents the experimental results. Section IX discusses about the conclusion.

2. PRELIMINARIES CONCEPTS
This section describes the formal graph definitions and notation used for this work.

Definition 1: Labeled Graph
A labeled graph G is defined as a 4-tuple, (V, E, L_V, L_E, l) where V is the set of vertices, $E \in V \times V$ is the set of edges, L_V and L_E are the set of labels of vertices and edges and l is a labeling function assigning a label to a vertex $l: V \rightarrow L_V$ or an edge $l: E \rightarrow L_E$.

Definition 2: Graph Isomorphism
Let $G = (V, E, L_V, L_E, l)$ and $G' = (V', E', L'_V, L'_E, l')$ be two graphs. A subgraph isomorphism from G to G' is an injective function $f: V \rightarrow L_V$ such that (1) $\forall u \in V$, $l(u) = l'(f(u))$, and (2) $\forall (u,v) \in E$, $l(u,v) = l'(f(u), f(v))$.

Definition 3: Graph Automorphism
Automorphism is a special case of graph isomorphism where $G_1 = G_2$, which means to find a mapping from a graph to itself. Most simplicity, an automorphism of a graph is an isomorphism from the graph to itself.

3. RELAED WORK

The power of using graphs to model complex datasets has been recognized by various researchers in chemical domain [3,4], computer vision [5,6], and machine learning [8,10]. There have been developed algorithms that discover all frequently occurring subgraphs in a large graph databases is particularly challenging and computationally intensive, as graph and subgraph isomorphism plays a key role throughout the computations.

Isomorphism between graphs becomes active research area. Most frequent subgraphs mining and graph isomorphism problems employ canonical code of a graph to test whether two graphs are isomorphic or not. A canonical form is to construct a code word that uniquely identifies a graph up to automorphism. The code word describes the connection structure of the graph. The resulting code words are sorted lexicographically. Then, the maximal (minimal) canonical code is chosen from all possible codes for a given graph[7,10].

The search space of canonical labeling can be reduced with vertex invariants. Vertex invariant is a well-known technique in which we can partition the vertices by their degrees and labels. Then, we try all the possible permutations of vertices inside each partition. Vertex invariants do not asymptotically change the computational complexity of canonical labeling. For example, if a given graph is regular, we cannot create fine partitions and vertex invariants do not reduce the search space [9].

A repository of processed subgraphs is the most straightforward way of avoiding redundant search. Every encountered frequent subgraph is stored in a data structure, which allows us to check quickly whether a given subgraph is contained in it or not. Whenever a new subgraph is created, this data structure is accessed and if it contains the subgraph, we know that it has already been processed and thus can be discarded. Only subgraphs that are not contained in the repository are extended and, inserted into the repository [4].

An efficient Fast-Graph Automorphic Filter (F-GAF) algorithm is proposed that used grid-code representation of graphs [2]. Given a graph database, this algorithm checks the automorphism of graphs without generating huge number of permutation matrices used in canonical labeling.

4. PROPOSED WORK

In this section, we discuss about an efficient Duplicated Automorphic Graph Cleaning (DAGC) algorithm that uses (i) the edge dictionary which contains the distinct edges of graphs stored in the graph database and (ii) adjacent edge (AdE) structure to identify and filter automorphic graphs efficiently. Our proposed approach bases on edge-based representation.

The proposed algorithm consists of four steps: (1) in the preprocessing step, grouping the same label vertices from the input graph and generating the corresponding edge list, (2) inserting the distinct edge into the edge dictionary if the edge does not exist in it, (3) computing the AdE structure for each graph and (4) matching AdE structures of existing database graphs with that structure of new graph to identify whether the new one already exists or not. The notations used in the DGAC algorithm are listed in Table 1. The architecture of our proposed system is shown in figure 1.

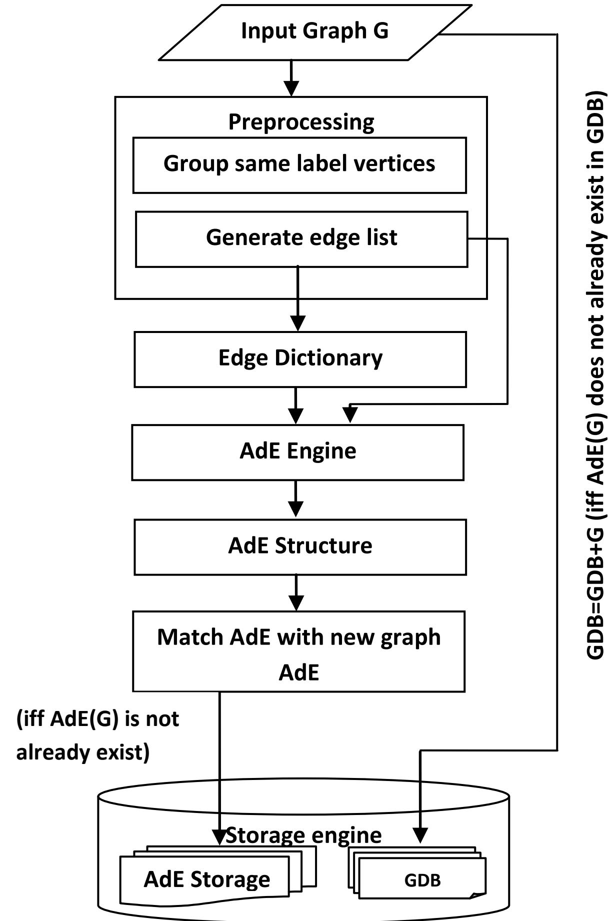

Fig 1: Architecture of the system

4.1 Preprocessing

In this phase, the input graph is preprocessed to collect information such as number of vertices *N*, number of edges *E*. To group vertices, vertices with the same label are grouped and counts the total number of same vertex labels. The vertices in the graph are considered as the same label if the label of a vertex is the same as that of another vertex in the graph. Each edge in the graph is collected to get the edge list of the given graph where each edge is represented as 3-tuples (u,l_e,v) where *v*is source vertex label, *e* is edge label and *u* is destination vertex label.

Table 1. Notations used in DAGC algorithm

Notation	Definition
GDB	Graph Database
$N \leftarrow \|V\|$	Number of vertices in G
$E \leftarrow \|E\|$	Number of edges in G
SG(G)	Same label vertices group in G
G	Input graph
E(G)	Edge list in G

AdE(G)	Adjacent edge structure of G
D_e(G)	Distinct edge in G
AdE_{store}	AdE structure stores in the AdE storage
e_{adj}	Adjacent edge of an edge e
EDict	Edge Dictionary

4.2 Edge Dictionary

The edge dictionary contains two parts: edge identifier (ID) and edge in the graph (Edge) which is the unique edge containing in the database. In the dictionary, an edge *e* is defined as 3-tuple (u,l_e,v) where *u* and *v* are the labels of the vertices and l_e is the label of the edge itself. Each edge appears only once in the edge dictionary, no matter how many times it appears in the graphs.

When a graph introduces in the graph database, the distinct edges are taken from the graph. And then it needs to check whether these edges already exist in the edge dictionary. If this contains these edges, we look up the corresponding identifier of that edge and use the identifier for further processing. If the edge is not in the dictionary already, the edge is then inserted into the dictionary and the corresponding ID of the edge dictionary increases serially. Moreover, most graphs in the chemical compound database have most similar edges and vertices and our proposed work mostly focuses on this chemical compound dataset.

4.3 Adjacent Edge Structure(AdE)

The AdE structure contains nearest neighbor information for each edge appeared in the graph. This AdE structure can be computed using the edge information from the preprocessing step and unique identifier of the edge from the edge dictionary. For all edges in the graph, the adjacent edges of each edge are computed where the identifiers of the adjacent edges are the unique edge identifiers defined in the dictionary. Moreover, the adjacent edge information for the graphs is transformed into AdE (Adjacent Edge) structure for further string comparisons efficiently.

4.4 Matching AdE Structure

AdE structure of the input graph is matched with that of other graph in the database to check whether the two graphs are automorphic or not. If the two graphs have the same AdE representation then the algorithm concludes that the graphs are automorphic and terminates without adding AdE structure and the graph itself into the storage engine. If any of these parametric quantities of AdE are different, the algorithm immediately reasons out that the graphs are different and put in AdE of the graph to the AdE storage structure and the graph into the database and terminates the process.

5. DUPLICATED AUTOMORPHIC GRAPH CLEANING (DAGC) ALGORITHM

The DAGC algorithm works according to the above four steps described in section 4. Firstly, DAGC compares the total number of vertices (*N*) in one graph with those in the other graph. Secondly, if *N* is same, same label vertices group (SGs) of these graphs are compared. Finally, AdEs of those graphs are also compared only if these graphs have same labels of vertices. This reduces the time consuming comparisons. Moreover, the proposed algorithm dramatically shortens the computational times complexity needed for identifying automorphic graphs when compared to F-GAF (Fast-Graph Automorphic Filter) algorithm and canonical labeling. Figure 2 and 3 show the DAGC algorithm for automorphic graph checking and the algorithm for generating adjacent edge structure.

Algorithm DuplicatedAutomorphicGraphCleaning (DAGC)

```
Input:  GDB←{G_1,G_2, ...,G_{i-1}},E(G_i)←input graph, EDict,
        AdE_store←{AdE(G_1),AdE(G_2),...,AdE(G_{i-1})}
Output: GDB←{G_1,G_2,...,G_{i-1},G_i},AdE_store←AdE(G_1),AdE(G_2)
        ,...,AdE(G_{i-2}),AdE(G_{i-1}),AdE(G_i)}if AdE(G_i)∉AdE_store
GDB:= Ø
ID:=0
If ∄D_e(G_i) ∈ EDict then
    EDict.ID:= EDict.ID+1
    EDict.Edge:=EDict.Edge+D_e(G_i)
End if
AdE(G_i) ←GenerateAdEStructure(E(G_i),EDict))
If G_i is the first graph entering into GDB then
        AdE_store:=AdE_store+AdE(G_i)
        GDB := GDB + G_i
        Return
End if
For each graph G_j
If (N(G_j) = N(G_i)) then
    If (SG(G_j) = SG(G_i)) then
        If ((AdE(G_j) = AdE(G_i)) then
                Result " G_j ≡ G_i" and Reject G_i
        Else
                AdE_store:=AdE_store + AdE(G_i)
                GDB:= GDB+ G_i
        End if
    End if
End if
Return
```

Fig 2: DAGC Algorithm

Algorithm GenerateAdEStructure(E(G_i),EDict)

```
∀ D_e(G_i) ∈ G_i
    Find all e_adj for D_e(G_i)
    Substitute each e_adj with corresponding EDict.ID
    AdE(G_i):= all e_adj for each D_e(G_i)
Return AdE(G_i)
```

Fig 3: GenerateAdEStructure algorithm

6. IDENTIFYING DUPLICATED AUTOMORPHIC GRAPH

Identifying symmetries is an important application of graph isomorphism. The collection of information about symmetries in the graph becomes identical to itself. If we can identify the graphs in GDB are duplicates, they can be discarded to avoid redundant work.

When a new graph G_N introduces to GDB, we need to recognize this new graph to eliminate redundant storing into the database. This process can also reduce needless loss of performance for further query processing. To identify automorphic graph, our proposed algorithm needs to check the nearest neighbor edges information of each D_e of G_N. To restrict the search space for adjacent edges information, the edge dictionary is used to discover the unique edge identifier of each D_e contain in G_N. If the edge dictionary does not contain all D_e, it is certain that the input G_N does not already exist in GDB. When all edges in G_N were previously defined in the dictionary, these edges are fixed with the identifiers of edges using the edge dictionary.

After assigning the edges with ID, we use these IDs to find adjacent edges information for each D_e in G_N. Instead of using the edges themselves, the identifiers of these edges can narrow down the storage space and efficient comparison for advance processing. Then we find all e_{adj} of each D_e and group together to become $AdE(G_N)$. To avoid the time consuming comparisons, we first compare $N(G_N)$ and $SG(G_N)$ with those of the graphs stored in GDB. If the parameters of G_N are same with those of at least one of the graphs, we need to compare $AdE(G_N)$ with AdEs collected in AdE_{store}. If these structures contain the same value, we conclude that the input G_N and one of the graphs in GDB are identical and remove G_N from GDB.

7. ANALYSIS AND ILLUSTRATION OF PROPOSED TECHNIQUE

The illustration and analysis of the computational time complexity of our proposed DAGC algorithm is depicted as follow. We apply chemical compound database in our proposed system. Figure 3 illustrates the chemical compound graph of shikimic acid and step-by-step procedure to obtain the AdE structure of $G_{shikimic}$. To check whether $G_{shikimic}$ already exists in GDB, the worst case number of comparisons taken by the proposed technique is also described.

According to the preprocessing step, the same label vertices can be grouped as follow. Atoms such as O, H and C are defined as nodes in the graph and bonds such as single (s) and double (d) represented edges.

$SG(G_{shikimic}) = \{ H = 4, O = 5, C = 7\}$

The edge dictionary can be obtained using the edge list information from the preprocessing step. The dictionary of $G_{shikimic}$ is shown in table 2. Assume that initially the edge dictionary contains no edge and $G_{shikimic}$ is the first graph entering into the GDB.

The AdE structure contains adjacent edges information for each edge appeared in the graph. To obtain AdE structure of a graph, AdE engine obtains edge information from the preprocessing stage and the edge dictionary. Table 3 shows the nearest neighbor edges for each edge in the $G_{shikimic}$.

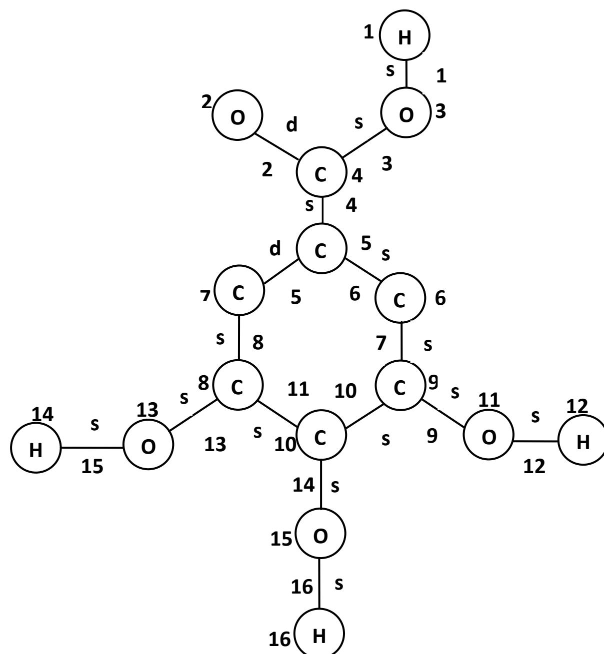

Fig 4: A chemical compound graph of shikimic acid ($G_{shikimic}$)

Table 2. EDict of graph $G_{shikimic}$

ID	Edge
1	<H, s, O>
2	<O, d, C>
3	<O, s, C>
4	<C, s, C>
5	<C, d, C>

Table 3. Adjacent edge information of $G_{shikimic}$

ID	Adjacent Edge
1	1 12 15 16 <3>, <3>, <3>,<3>
2	2 <3,4>
3	3 9 13 14 <1,2,4>, <1,4,4>, <1,4,4>, <1,4,4>
4	4 6 7 8 10 11 <2,3,4,5>,<4,4,5>,<3,4,4>,<3,4,5>,<3,3,4,4>,<3,3,4,4>
5	5 <4,4,4>

Then the adjacent edge information of $G_{shikimic}$ can be represented in term of AdE structure.

$AdE(G_{shikimic})=\{1\{\{3\},\{3\},\{3\},\{3\}\},2\{3,4\},3\{\{1,2,4\},\{1,4,4\},\{1,4,4\},\{1,4,4\}\},4\{\{2,3,4,5\},\{4,4,5\},\{3,4,4\},\{3,4,5\},\{3,3,4,4\},\{3,3,4,4\}\},5\{4,4,4\}\}$

The analysis of the computational time of DAGC algorithm is described as follows. To find the AdE structure for each graph, the algorithm takes E comparisons to generate distinct edge list of each graph. The same numbers of E comparisons are required to check the dictionary where the edges in the incoming graph are already in it. Then, E comparisons are needed to get the adjacent edge information of edges in the graph. Therefore, the total number of comparisons needed for constructing the AdE structure is*(E+E+E) =3E.*

Assume there are k graphs already in the GDB. We need to check whether the new input graph k+1 is automorphic to any of the k graphs. In the preprocessing step, the computational time to compare the number of vertices in the graph is 1. It requires *N* comparisons to group the same label vertices for a graph. The number of comparisons needed to compare the same group vertices is also *N*. At the worst case, comparison between AdE structures is *E(E-1)*. Therefore, the maximum number of comparisons needs to check out k graphs would be $k(1+N+N+E^2-E)$. Therefore, at the worst case, the total number of comparisons for DAGC algorithm would *be* $3E+k(1+2N+E^2-E)$.

The detail analysis of computational time of our propose DAGC algorithm for chemical compound graph $G_{shikimic}$ is described as follow. There are 16 vertices and 16 edges in $G_{shikimic}$. To generate the distinct edge list for this graph, we need to take 16 comparisons. To check whether the edges in the graph already exist in the edge dictionary, the number of comparison needed is 16. To compute the adjacent edge information for each edge, 16 comparisons are needed. Therefore, the total number of comparisons for AdE structure is (16+16+16)=48.

To check whether $G_{shikimic}$ exists in GDB or not, we need to compare the AdE structure of $G_{shikimic}$ with other AdE structures of AdE storage. To reduce the computational time complexity, we first check the total number of vertices in $G_{shikimic}$. The time complexity to check vertices count is 1. The number of comparisons to group same label vertices is 16 (no. of vertices in $G_{shikimic}$). To test the same group vertices, the number of comparisons required is 16. If the vertices of $G_{shikimic}$ match with one of the graphs in GDB, the AdE structure of $G_{shikimic}$ is needed to compare with those structures of the graphs in GDB. The number of comparisons necessitates to comparing AdE structure is 240(E^2-E). If the database contains 100 graphs, the total time complexity of DAGC algorithm is 48+100(1+16+16+240) =27,348.

8. EXPERIMENTAL RESULTS

A study on three techniques to test various types of graphs such as sparse, dense and complete graphs was conducted. Table 4 describes the total time complexity of three techniques: Canonical Code, Fast-Graph Automorphic Filter (F-GAF) algorithm, and DAGC algorithm. In canonical code, we need to compute the factorial of total number of vertices (V!) for a graph because it is vertex-based representation. Then maximum or minimum canonical code must be selected from the number of V! canonical codes. Assume the length of each canonical code is l. Therefore, if the number of graphs in GDB is k, we need to compare $k(V!\times l)$ times for each graph to check whether the input graph is automorphic or not. For F-GAF algorithm, the grid code representation of each graph requires 4E comparisons and it is based on edge-based representation. Then the worst case number of comparisons for F-AGF $4E+k(2+3E+(5N)^2-N)$.

Table 5 and 6 shows the analysis of computational time complexity of three techniques. Our proposed work significantly reduces the number of comparisons needed for identifying automorphic graphs than canonical code representation. Our proposed DAGC algorithm can mostly reduce at least 15 times computational time complexity for sparse graph when compared to F-GAF algorithm. It also reduces the number of comparisons more commonly than F-GAF for dense graphs and complete graphs. As a result, the study shows that our method performs proficiently well compared to canonical code representation. Moreover, our method reduces the computational time complexity more efficiently than F-GAF algorithm. DAGC algorithm outperforms other two techniques when the graphs in GDB are chemical compound datasets because the graphs in chemical compound database are sparse and dense graphs.

Table 4. Time complexity to check graph automorphism of three techniques

Techniques	Total Time Complexity
Canonical Code	$k(V!\times l)$
F-GAF	$4E+k(2+3E+(5N)^2-N)$
DAGC	$3E+k(1+2N+E^2-E)$

Table 5. Comparisons between three techniques for (50) graphs

No. of graphs in GDB=50				
Sparse graphs				
No. of vertices	No. of Edges	Techniques		
		Canonical Code	F-GAF	DAGC
8	9	20,160,000	81,086	4,477
9	11	181,440,000	102,594	6,483
10	12	1,995,840,000	126,448	7,686
Dense graphs				
8	20	20,160,000	82,780	19,910
9	28	181,440,000	105,212	38,834
10	35	1,995,840,000	129,990	60,655
Complete graphs				
8	28	20,160,000	84,012	38,734
9	36	181,440,000	106,144	64,058
10	45	1,995,840,000	131,530	100,185

Table 6. Comparisons between three techniques for (100) graphs

No. of graphs in GDB=100				
Sparse graphs				
No. of vertices	No. of Edges	Techniques		
		Canonical Code	F-GAF	DAGC
8	9	40,320,000	162,136	8,927
9	11	362,880,000	205,144	12,933
10	12	3,991,680,000	252,848	15,336
Dense graphs				
8	20	40,320,000	154,680	39,760
9	28	362,880,000	210,412	77,584
10	35	3,991,680,000	259,840	121,205
Complete graphs				
8	28	40,320,000	167,912	77,384
9	36	362,880,000	212,744	128,008
10	45	3,991,680,000	262,880	200,235

9. CONCLUSION

In this paper, we propose an efficient algorithm to identify and removal of automorphic graph storing into the graph database using the edge dictionary for overall graph database and adjacent edge structure (AdE) for each graph. The edge dictionary is efficiently used to narrow down the search space of adjacent edge information. The proposed approach eliminates the duplicated graphs storing in DB using the AdE structure. The AdE structure is described in term of string and therefore it is more efficient for further query processing. The computational time complexity is significantly reduced compared to canonical code and F-GAF algorithm. This approach requires more space for constructing the edge dictionary when the graphs in the graph database contain mostly distinct edges. However, the storage space can be reduced when the edges in graphs are most similar such as graphs in chemical compound database.

10. ACKNOWLEDGMENTS

I would like to thank Dr. Ni Lar Thein, Rector of the University of Computer Studies, Yangon, for giving me the opportunity to attend the doctoral degree course in the University of Computer Studies, Yangon. I would like to express my gratitude to my supervisor Dr. Kyaw May Oo, Associate Professor of the University of Computer Studies, Yangon, for allowing me to develop this research and for her valuable advice during period of my study. I also pay respect her for giving enough consideration to my ideas and views. I am extremely fortunate to work under her supervision for my research. I also dedicate this research to my respectful parents for their kindness to fulfill my ambitions.

11. REFERENCES

[1] M. Kuramochi and G. Karypis. Frequent Subgraph Discovery. Proc. 1st IEEE Int. Conf. on Data Mining (ICDM 2001), San Jose, CA), 313–320. IEEE Press, Piscataway, NJ, USA 2001

[2] R Vijayalakshmi, R Nadarajan, P Nirmala, and M Thilaga. A Novel Approach for Detection and Elimination of Automorphic Graphs in Graph Databases. Int. J. Open Problems Compt. Math., Vol. 3, No. 1, March 2010

[3] R. N. Chittimoori, L. B. Holder, and D. J. Cook. Applying the SUBDUE substructure discovery system tothe chemical toxicity domain. In Proc. of the 12th International Florida AI Research Society Conference, pages 90–94, 1999

[4] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In R. Agrawal, P. Stolorz, and G. Piatetsky-Shapiro, editors, Proc. of the 4th International Conference on Knowledge Discovery and Data Mining, pages 30–36. AAAI Press, 1998.

[5] H. K¨alvi¨ainen and E. Oja. Comparisons of attributed graph matching algorithms for computer vision. In Proc. of STEP-90, Finnish Artificial Intelligence Symposium, pages 354– 368, Oulu, Finland, June 1990.

[6] D. A. L. Piriyakumar and P. Levi. An efficient A* based algorithm for optimal graph matching appliedto computer vision. In GRWSIA-98, Munich, 1998.

[7] D. W. Williams, J. Huan, W. Wang. "Graph Database Indexing Using Structured Graph Decomposition", 2007.

[8] R. Agrawal and R. Srikant. Mining sequential patterns. In P. S. Yu and A. L. P. Chen, editors, Proc. Of the 11th Int. Conf. on Data Engineering (ICDE), pages 3–14. IEEE Press, 1995.

[9] S. Fortin. The graph isomorphism problem. Technical Report TR96-20, Department of Computing Science, University of Alberta, 1996.

[10] J. Huan, W. Wang, and J. Prins, "Comparing Graph Representations of Protein Structure for Mining Family-Specific Residue-Based Packing Motifs", Journal of Computational Biology (JCB), Vol.12, No.6, pp.6576671,2005.

12. AUTHORS PROFILE

Aye Nwe Thaing received her **B.C.Sc** in computer science from Government Computer College, Sittwe, Myanmar in 2003, her **B.C.Sc(Hons:)** and **Master** degrees from University of Computer Studies, Yangon, Myanmar in 2004 and 2007. Currently, she is a **Ph.D** candidate at University of Computer Studies, Yangon, Myanmar. Her research interests include graph database technology, efficient query processing of graph database and database management system.