# A Graph Representative Structure for Detecting Automorphic Graphs

Yu Wai Hlaing[1], Kyaw May Oo[2]

[1] University of Computer Studies, Yangon, [2] University of Information Technology
yuwaihlaing.1987@gmail.com, kmayoo19@gmail.com

**Abstract.** Graphs are prevalently used to model the relationships between objects in various domains. Storing the graphs into large databases is a challenging task as it deals with efficient space and time management. Unlike item sets in huge transactional databases, it becomes essential to ensure the consistency of graph databases since relationships among edges of a graph are predominant. One of the necessary procedures required is a mechanism to check whether two graphs are automorphic(duplicated) or not. Difficulty in identifying and eliminating the automorphic graphs is a challenging problem to the research community. In this paper, we propose a graph representative structure that is called graph code. There are three main phases: preprocessing, code generation and code matching. In preprocessing phase, vertex list, edge list and adjacent edge information are generated for input graph. In code generation, edge dictionary plays an important role. The edge dictionary and adjacent edge information are used to generate graph codes. In code matching, the new graph code is compared with those of other graphs in graph dataset to determine whether they are automorphic or not. The experimental results and comparisons offer a positive response to the proposed structure.

**Keywords:** edge dictionary, graph automorphism, graph code, graph representative structure.

## 1 Introduction

A graph describes relationships over a set of entities. With node and edge labels, a graph can describe the attributes of both the entity set and the relation. Labeled graphs appear in many research domains such as drug design [1], protein structure comparison [2], video indexing [3], and web information [4]. In addition, rapidly increasing Web sites and XML documents can also be modeled as graphs. Therefore, it is evident that graph data will become more and more prevalently used in the near future.

Various conferences over the past few years on mining graphs have motivated researchers to focus on the importance of mining graph data. One of the major perceptions concerned in graph mining is discovering frequent patterns [5], [6], [7], and [8]. The key operation required by any frequent subgraph discovery algorithm is a mechanism to check whether two subgraphs are identical or not. Many graph databases such

as chemical graphs have more than one vertex with the same label. There are possibilities that the same graph is stored more than once in the graph database leading to adverse results of mining. So, there is a need to represent graphs efficiently in order to identify automorphic graphs.

Structures that can be represented as graphs are based on graph theory [9]. Graph databases apply graph theory to store information about the relationships between entities in terms of graphs. There are many different structures that can be represented as graphs. Perhaps the simplest graph representation of a graph is as an unordered edge sequences. Each edge contains a pair of node indices and, possibly, associated information such as an edge weight. Adjacency array support easy access to the edges leaving any particular node, we can store the edges leaving any node in an array. If no additional information is stored with the edges, this array will just contain the indices of the target nodes. Next one is adjacency list. Size of the array is equal to number of vertices. Let the array be array []. An entry array[i] represents the linked list of vertices adjacent to the $i^{th}$ vertex. An $n$-node graph can also be represented by an $n \times n$ adjacency matrix $A$. $A_{i,j}$ is 1 if $(i, j) \in E$ and 0 otherwise. Among them the two most popular graph representative structures are adjacency list and adjacency matrix.

The most important one is to be compact and less time needed in generating representative structures for graph. Our graph code is a new way of representing graph data to process graph queries without verification between graph structures. It is developed to process labeled, undirected chemical compound graphs in the area of chemical informatics.

The rest of the paper is organized as follows. Section 2 reviews the related works in this area. Section 3 presents the formal definitions and notations used for the proposed research work. Section 4 explains proposed graph representative structure. Section 5 shows the experimental results of proposed structure using AIDS Antiviral Screen Compound Dataset. In section 6 concludes our paper.

## 2 Related Works

Over the years, a number of different structures have been developed to represent graphs more and more efficiently. Developing such structures is particularly challenging in terms of storage space and computational time.

Michihiro Kuramochi et al. proposed a canonical labeling [10] for representing graphs. Canonical label of a graph is nothing more than a code that uniquely identifies the graph such that if two graphs are isomorphic to each other, they will be assigned the same code. Canonical label of a graph is as the string obtained by concatenating the upper triangular entries of the graph's adjacency matrix. Once the canonical label has been obtained, the adjacency matrix representation is discarded. The disadvantage of this structure is if a graph has |V| vertices, the complexity of determining its canonical label is in $O(|V!|)$ making it impractical even for modern size graphs.

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 [11].

R Vijayalakshmi et al. [12] propose F-GAF: a novel approach for detection and elimination of automorphic graphs in graph database. F-GAF uses edge-based graph representation. It involves three main phases: preprocessing, feature extraction and pattern matching. In preprocessing, edge list of the input graph is generated. In feature extraction phase, grid code is generated. In pattern matching the new grid code is compared with those of other graphs in graph database. The weak points of this structure are it requires exhausted enumeration to generate grid code and it needs large number of comparisons to determine automorphic graphs.

## 3 Definitions and Notations

This section presents the key concepts, notations and terminology used in this paper, which include: labeled graphs, graph isomorphism, graph automorphism, and graph code representation.

**Definition 1** (Labeled Graph) A labeled graph is a four-element tuple $G = (V, E, \Sigma, f)$where $V$ is a set of vertices and $E$ is a set undirected edges joining two distinct vertices. $\Sigma$ is the set of vertex and edge labels and f maps vertices and edges to their labels.

**Definition 2** (Graph Isomorphism) An isomorphism of graphs $G$ and $H$ is a bijection between the vertex sets of $G$ and $H$; $f: V(G) \rightarrow V(H)$such that any two vertices $u$ and $v$ of $G$ are adjacent in $G$ if and only if $f(u)$ and $f(v)$ are adjacent in $H$.

**Definition 3** (Graph Automorphism) An automorphism of a graph $G = (V, E, \Sigma, f)$ is a permutation $\sigma$ of the vertex set $V$, such that the pair of vertices $(u, v)$ form an edge if and only if the pair $(\sigma(u), \sigma(v))$ also form an edge. That is, it is a graph isomorphism from $G$ to itself.

**Definition 4** (Graph Code Representation) For a graph $G$, the code of $G$, denoted by $c(G)$ is in the form $e_{id}\{(v), eid_{adj}, \ldots\}\ldots$ depending on adjacent edge information of preprocessing. $e_{id}$ is the edge id, v is vertex label on which two edges are connected, $eid_{adj}$ is list of adjacent edge ids for this edge.

An example labeled graph is shown in Fig. 1.

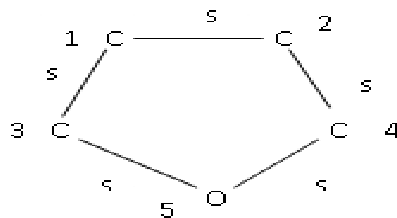


**Fig. 1**. A Labeled Graph ($G$)

An example graph code of G is $c(G)$=1{(c)1,1} 1{(c)1,2} 1{(c)1,2} 2{(c)1,(0)2} 2{(c)1,(0)2}.

## 4 Our Graph Representative Structure

In this section, our proposed graph representative structure is described that applied on a chemical compound graph dataset for efficient detection of automorphic chemical graphs. There are three main phases to generate graph code. The first phase is preprocessing. And the other is code generation. Edge Dictionary plays an important role in generating graph code. The final phase is code matching phase. The overall structure is described in Fig. 2.

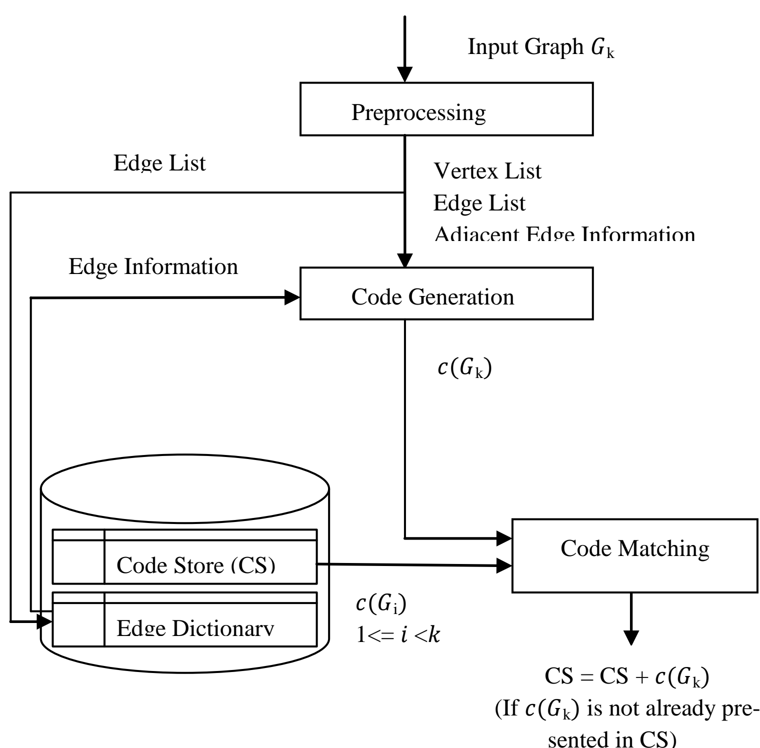


**Fig. 2**. Overall Structure of Proposed Work

### 4.1 Preprocessing

In this step, for a graph the vertex list, edge list, and adjacent edge information are generated. Each vertex in the graph is assigned with unique id.

| Vertex id : | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Vertex List : | C | C | C | C | O |

Then the edge list of the graph is defined as ($V_{id}$,L,$V_{id}$) where $V_{id}$ is the vertex id, L is the edge label. The output of this step is as follows.

| $V_{id}$,L,$V_{id}$ : | <1,s,2> | <1,s,3> | <2,s,4> | <3,s,5> | <4,s,5> |
|---|---|---|---|---|---|
| Edge List: | <C,s,C> | <C,s,C> | <C,s,C> | <C,s,O> | <C,s,O> |

### 4.2 Edge Dictionary

When a graph introduced to the database, the edges in the graph are added into edge dictionary if these edges are not already existed in edge dictionary. In this dictionary, these edges are assigned with global unique ids for further processing. Fig. 3 shows the example edge dictionary for input graphs.

| Id | Edge |
|---|---|
| 1 | <C,s,C> |
| 2 | <C,s,O> |
| 3 | <C,s,C> |
| ... | … |

**Fig. 3**. Edge Dictionary

### 4.3 Code Generation

A graph is represented holistically into a graph code that captures the structural representation of the graph. Every edge in the graph is assigned with global unique identifier already defined in the edge dictionary. Instead of using the edge itself, using the edge id in the dictionary can have advantages in three ways:

- Firstly, using the edge id in the code saves the amount of storage space.
- Secondly, using the same id for the duplicated edge is effective when constructing the edge code.
- Thirdly, using the edge id in the code reduces the time of matching graphs.

Most of the chemical graphs have a lot of common edges. So, edge dictionary uses a little memory space. Edge dictionary and adjacent edge information are used to generate graph code. Graph code of the input graph is stored in code store (CS) if it is not already existed in CS.

### 4.4 Code Matching

After generating the graph code of the $k^{th}$ graph, Code of $G_k$ is compared with each graph code of $G_i$, $1 \leq i < k$ in code store (CS). If the graph code of $G_k$ is not the

same code as one of $G_i$, concludes that the graph is not automorphic graph and its graph code is stored in CS. Otherwise, the graphs are automorphic and then terminate the process. In this process of comparison in the specified order, if any of these codes are different, the algorithm immediately concludes that the graphs are different.

# 5 Experimental Results

The experiment described in this section use the AIDS Antiviral Screen Dataset. The AIDS Antiviral Screen Dataset from Development Therapeutics Program NCI/NIH is available publicly. The dataset contains 43,906 chemical compounds. Each compound has 32 vertices and 34 edges in average. The maximum one has 188 vertices and 196 edges. In this section, we evaluate the storage space of our graph codes with the storage space required by other two formats: xml file and image file (.png). In Fig. 4, it can be seen that the storage space of our graph codes is significantly less than the other two formats.

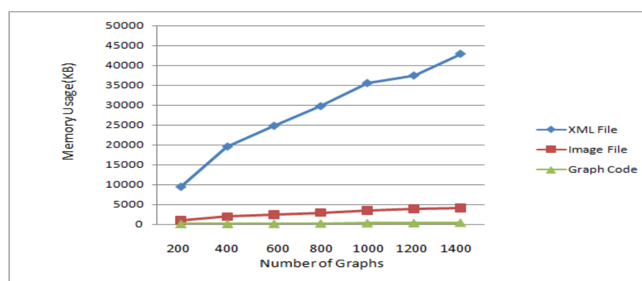


**Fig. 4**. Analysis of Storage Space for Various Numbers of Graphs between Graph Code, XML File, Image File

The time complexity is commonly estimated by counting the number of elementary operations performed by an algorithm where an elementary operation takes a fixed amount of time to perform. In proposed approach, computational time complexity for generating graph code takes $O(NE + (E * (E - 1))/2)$ where $E$ is the number of edges, $V$ is the number of vertices and $N$ is the size of edge dictionary. Proposed approach takes $NE$ comparisons is required to check edge dictionary whether the edges in the input graph already exist in it or not. For getting adjacent edge list, $(E * (E - 1))/2$ comparisons are required. Table.1 describes the computational time complexity between three techniques for generating graph representative structure. Tabel.2 shows the analysis of computational time complexity for code generation between three techniques. Graph code can reduce at least $10^4$ times computational complexity when compared to canonical labeling. But it consumes a little more time than F-GAF for code generation.

**Table 1**. Computational Time Complexity for Code Generation between Three Techniques

| Technique | Computational Complexity |
|---|---|
| Canonical Labeling | $O(V!)$ |
| F-GAF | $O(4E)$ |
| Graph Code | $O(NE+(E*(E-1))/2)$ |

**Table 2**. Analysis of Computational Time Complexity for Code Generation between Three Techniques

| No. of vertices | No. of edges | Canonical Labeling | F-GAF | GraphCode(N=19) |
|---|---|---|---|---|
| 10 | 10 | $3.63 \times 10^{6}$ | 4.0 x10 | $1.45 \times 10^{2}$ |
| 20 | 20 | $2.43 \times 10^{18}$ | 8.0 x10 | $5.90 \times 10^{2}$ |
| 40 | 40 | $8.16 \times 10^{47}$ | $1.60 \times 10^{2}$ | $2.38 \times 10^{3}$ |
| 60 | 60 | $8.32 \times 10^{81}$ | $2.40 \times 10^{2}$ | $5.37 \times 10^{3}$ |
| 80 | 80 | $7.16 \times 10^{118}$ | $3.20 \times 10^{2}$ | $9.56 \times 10^{3}$ |

Table. 3 shows the analysis of computational time complexity for detecting automorphic graphs between canonical labeling, F-GAF and proposed approach. The analysis of comparisons for detecting automorphic graphs required by three techniques is shown in table. 4. Proposed approach can reduce at least $10^{2}$ times computational complexity when compared to canonical labeling. It can lessen at least 1 time computational time complexity than F-GAF.

**Tabel 3.** Computational Time Complexity for Detecting Automorphic Graphs between Three Techniques

| Technique | Computational Complexity |
|---|---|
| Canonical Labeling | $O\ ((V!)\ +\ k\ *\ ((V2)-V)/2))$ |
| F-GAF | $O\ ((4E)\ +\ k\ *\ (2+3E+(5V)^{2}-V))$ |
| Graph Code | $O\ ((NE+(E*(E-1))/2)\ +\ k\ *\ 2E*(V-2))$ |

Where,
$V$ = number of vertices
$E$ = number of edges
$N$ = size of edge dictionary
$k$ = number of graphs

**Tabel 4.** Analysis of Computational Time Complexity for Detecting Automorphic Graphs between Three Techniques in 100 Graphs

| No. of vertices | No. of edges | Canonical Labeling | F-GAF | GraphCode(N=19) |
|---|---|---|---|---|
| 10 | 10 | $3.63 \times 10^{6}$ | $2.52 \times 10^{5}$ | $1.61 \times 10^{4}$ |
| 20 | 20 | $2.43 \times 10^{18}$ | $1.00 \times 10^{6}$ | $7.26 \times 10^{4}$ |
| 40 | 40 | $8.16 \times 10^{47}$ | $4.01 \times 10^{6}$ | $3.06 \times 10^{5}$ |
| 60 | 60 | $8.32 \times 10^{81}$ | $9.01 \times 10^{6}$ | $7.01 \times 10^{5}$ |
| 80 | 80 | $7.16 \times 10^{118}$ | $1.60 \times 10^{7}$ | $1.26 \times 10^{6}$ |

# 6 Conclusion and Future Work

We propose a new way of representing a graph holistically via its graph code. The edge dictionary is efficiently used to narrow down the search space of graph code. Automorphic graphs can be detected by only comparing graph codes instead of expensive automorphism testing. From our experimental results, our graph code outperforms the existing storage structures and methods. Efficient graph querying using graph codes is going to be observed as a future work.

# References


1. C. Borgelt, and M.R. Berthold, "Mining Molecular Fragments: Finding Relevant Substructures of Molecues", *ICDM*, pp. 51-58, 2002.
2. J. Huan, W. Wang, D. Bandyopadhyay, J. Snoeyink, J. Prins and A. Tropsha, "Mining Protein Family Specific Residue Packing Patterns from Protein Structure Graphs", In Proceedings of the 8$^{th}$ Annual International Conference on Research in Computational Molecular Biology (RECOMB), pp. 308-315, 2004.
3. B.T. Messmer, and H. Bunke, "A Decision Tree Approach to Graph and Subgraph Isomorphism Detection", *Pattern Recoginition*, Vol. 32, No. 12, pp. 1979-1998, December 1991.
4. S. Raghavan, and H. Garcia-Molina, "Representing Web Graphs", In Proceeding of IEEE International Conference on Data Engineering, 2003.
5. R. Agrawal, and R. srikant, "Mining Sequential Patterns", Proceeding of the 11$^{th}$ International Conference on Data Engineering (ICDE), pp 3-14, IEEE Press, 1995.
6. R. Agrawal, and R. srikant, "Fast Algorithms for Mining Associations Rules", Proceeding of 20$^{th}$ International Conference on Very Large Databases (VLDB), pp. 487-499, September 1994.
7. C. W. K. Chen, and D. Y. Y. Yun, "Unifying graph-matching problem with a practical solution", In Proceeding of International Conference on Systems, Signals, Control, Computers, September 1998.
8. R. N. Chittimoori, L. B. Holder, and D. J. Cook, "Applying the SUBDUE Structure discovery system to the chemical toxicity domain", In Proceeding of the 12$^{th}$ International Florida AI Research Society Conference, pp. 90-94, 1999.
9. J. A. Bondy, and U. S. R. Murty, "Graph Theory With Applications", Elsevier Science Publishing Co., Inc, New York, 1976.
10. M. Kuramochi, and G. Karypis, "An Efficient Algorithm for Discoverying Frequent Subgraphs", In Proc. 2001 International Conference on Data Mining (ICDM' 01), pp. 313-320, San Jose, CA, Nov. 2001.
11. S. Fortin, "The graph isomorphism problem", Technical Report TR 96-20, Department of Computing Science, University of Alberta, 1996.
12. R. Vijayalakshmi, R. Nadarajan, P. Nirmala, and M. Thilaga, "A Novel Approach for Detection and Elimination of Automorphic Graphs in Graph Database", Int. J. Open Problems Compt. Math., Vol. 3, No. 1, March 2010.