Graph Code Based Isomorphism Query on Graph Data

Yu Wai Hlaing
University of Computer Studies, Yangon
yuwaihlaing.1987@gmail.com

Kyaw May Oo
University of Information Technology
kmayoo19@gmail.com

***Abstract*— The field of graph datasets and graph query processing has received a lot of attention due to constantly increasing usage of graph data structures for representing data in different domains. Storing the graph in large dataset is a challenging task as it deals with efficient space and time management. To ensure the consistency of graph dataset, one of the necessary procedures required is a mechanism to checks whether two graphs are automorphic or not. Given a graph query, it is also desirable to query isomorphic graphs quickly and efficiently from a large dataset via its index. In this paper, we propose a graph isomorphism querying approach that based on graph code. Graph code is a graph representative structure that can preserve the structural information of the original graph. Graph isomorphism query can be efficiently processed via trie that is constructed on graph codes of the data graphs. The experimental results and comparisons offer a positive response to the proposed approach.**

Keywords—graph data; graph representative structure; automorphism; isomorphism; graph querying

I. Introduction

Graph is a powerful tool for representing and understanding objects and their relationships in various application domains. Conceptually, any kind of data can be represented by graphs. The power of graphs to model the complex datasets has been recognized by various researchers. In Chemical informatics and bio-informatics, scientists use graphs to represent compounds and proteins. Daylight system [11], a commercial product for compound registration, has already been used in chemical informatics. In recent years, graph datasets have become more in use and the volume of graph data increased rapidly.

A well-known representation of graph structured data is an adjacency matrix representation. Many graph datasets such as chemical graphs have more than one vertex with the same level. These graphs have more than one adjacency matrix representation based on the ordering of same vertex labels, and it becomes difficult to identify them uniquely. There are possibilities that the same graph is stored more than once in the graph dataset leading to the adverse results of mining. Also, if stored more than once in different adjacency matrices, a single graph affects the consistency of graph dataset [12].

In the core of many graph-related applications, lies a common and critical problem: how to efficiently process graph queries and retrieve related graphs. In some cases, the success of an application directly relies on the efficiency of the query processing system. A primary challenge in computing the answers of graph queries is that pair-wise comparisons of graphs are usually really hard problems. A naïve approach to compute the answer set of a graph query q is to perform a sequential scan on the dataset's graphs and to check whether each graph dataset members satisfies the conditions of q or not. However, the graph dataset can be very large and which makes the sequential scan over the dataset impractical. Thus, finding an efficient querying technique is immensely important due to the combined costs of pair-wise comparisons and the increasing size of modern graph datasets. Clearly, it is necessary to build graph indices in order to help processing graph queries. It is apparent that the success of any graph data-based application is directly dependent on the consistency of graph datasets and efficiency of the graph indexing and querying mechanisms.

The rest of the paper is organized as follows. Section II provides some definitions and introduces the notation that is used in the paper. Section III describes the related works in this area while the section IV focuses on the proposed approach. Our experimental results are reported in section V. Section VI gives the conclusion and future works.

II. Definitions and Notation

This section presents the key concepts, notations, and terminology used in this paper, which include: labeled graph, graph automorphism, graph isomorphism, canonical label and graph code. As a general data structure, labeled graph is used to model complicated structures and schemaless data. In labeled graph, vertex and edge represent entity and relationship, respectively. The chemical compound shown in Fig. 1 is an undirected labeled graph.

A. Labeled Graph

A labeled graph G_1 is a 5-tuple, $\{V, E, \Sigma V, \Sigma E, l\}$ where V is a set of vertices and E is a set of undirected edges. ΣV and ΣE are the sets of vertex labels and edge labels respectively. The labeling function l defines the mappings $V \rightarrow \Sigma V$, $E \rightarrow \Sigma E$.

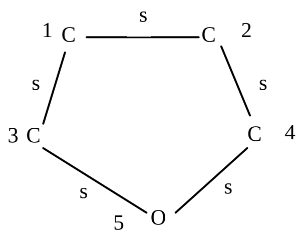

Fig. 1. A Labeled Graph(G_1)

B. Graph Automorphism

An automorphism [9] between two graphs G_1 and G_2 is an isomorphism mapping where $G_1 = G_2$. An isomorphism mapping is a mapping of the vertices of G_1 to vertices of G_2 that preserve the edge structure of the graphs. That is, it is a graph isomorphism from a graph G_1 to itself.

The graph G_2 shown in Fig. 2 is automorphic to graph G_1.

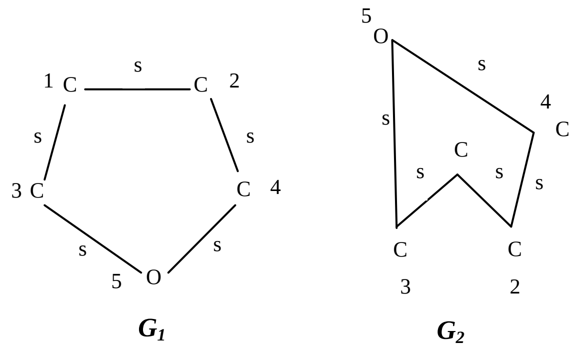

Fig. 2. Graph Automorphism($G_1 = G_2$)

C. Graph Isomorphism

An isomorphism [10] of graphs G_1 and G_2 is a bijection between the vertex sets of G_1 and G_2; such that any two vertices u and v of G_1 are adjacent in G_1 if and only if $f(u)$ and $f(v)$ are adjacent in G_2. This kind of bijection is generally called "edge-preserving bijection".

D. Canonical Label

A canonical labeling of graph G_1 is an isomorphism-invariant labeling of G_1's vertices, i.e., two graphs G_1 and G_2 have the same canonical labeling if and only if they are automorphic to each other [1], [2].

E. Graph Code

For a graph G_1, the code of G_1, denoted by $c(G_1)$ is the list of the form e_{id}\{(v), e_{id_adj},…}… depending on adjacent edges. e_{id} is the edge id, v is vertex label on which two edges are connected, e_{id_adj} is list of adjacent edge ids for this edge.

III. Related Works

The significant of using graphs to represent complex datasets has been recognized in different disciplines such as chemical domain [3], computer vision [4], and image and object retrieval [5].

To avoid the ambiguity of representation and inefficiency in the graph pattern search, canonical labeling approach that produces a unique code for each graph called canonical label has been used in Frequent Subgraph Discovery algorithm (FSG) [6]. A simple way of defining the canonical label of an undirected graph is to use the string obtained by concatenating the upper triangular elements of the graph's adjacency matrix when this matrix has been symmetrically permuted such that this string is lexicographically largest or smallest among the strings obtained from all such permutations. If a graph contains $|V|$ vertices, the worst case time complexity to compute its canonical code is O ($V!$) since $|V|!$ permutations of vertices have to be checked before selecting the minimum (or maximum) code.

Other popular attempt for detecting automorphic graph is Fast-Graph Automorphic Filter (F-GAF) [7]. F-GAF detects automorphic graphs in three phases. In preprocessing phase, edge array of the graph is computed by grid traversal technique. Then, the grid code is generated in feature extraction phase. The grid code is a feature vector consisting of the edge array, distinct vertex labels, the degrees and all vertex labels with degrees of each of its neighbors of k^{th} graph. In pattern matching phase, the grid code of G_k is compared with those of other graphs in graph database to check automorphism. For large graphs, the length of grid code can be very long. It has overhead of space and expensive processing time for detecting automorphic graphs.

A. N. Thaing and K. M. Oo propose Compact Graph Representative Structure (CGRS) [8] called edge code. It has two phases to generate edge code. In preprocessing phase, the input xml file is parsed with xml parser. And then the edge codes are generated in code generation phase. The edge code is composed of array of edge ids follows by lists of edge ids that are connected with this edge. These codes are used in detecting automorphic graphs. In querying phase, the edge code of the query is sequentially scanned over edge codes of data graphs to retrieve isomorphic graph. The structural information of the graph such as the two edges are connected on which vertex can be lost. This can have large affects on the accuracy of the result. It is possible that incorrect results will be returned. It also has time overhead due to sequential scanning for querying.

IV. Proposed Approach

In this section, our proposed graph isomorphism querying approach that based on graph code for efficiently detecting automorphic graphs and querying isomorphic graph is described. There are two main phases in proposed approach. In graph code generation phase, there are two sub-steps: preprocessing and code generation. Edge dictionary and adjacent edge information play important roles in code generation. For a new graph, generate its graph code and make automorphism test then determine to add its graph code or not. In querying phase, all of the graph codes and corresponding graph ids are inserted into trie. Then, query's graph code is probed in this trie to retrieve isomorphic graphs efficiently.

A. Preprocessing

In this phase, the graph information such as vertex information, edge information, and adjacent edge information are generated by parsing input xml files with xml parser. Then the edge information of the graph is defined as (V_{id},L,V_{id}) where V_{id} is the vertex id, L is the edge label. Then adjacent edge information is generated. Fig. 3 shows graph information for graph G_1 in Fig. 1.

Vertex id :	1	2	3	4	5
Vertex Info :	C	C	C	C	O

(a)

***V_{id},L,V_i*:**	<1,s,2>	<1,s,3>	<2,s,4>	<3,s,5>	<4,s,5>
Edge Info:	<C,s,C>	<C,s,C>	<C,s,C>	<C,s,O>	<C,s,O>

(b)

Edge:	**Adjacent Edges:**
<1,s,2>	<1,s,3>, <2,s,4>
<1,s,3>	<1,s,2>, <3,s,5>
<2,s,4>	<1,s,2>, <4,s,5>
<3,s,5>	<1,s,3>, <4,s,5>
<4,s,5>	<2,s,4>, <3,s,5>

(c)

Fig. 3. Graph Information of G_1 (a) Vertex Information (b) Edge Information (c) Adjacent Edge Information

For each edge from the graph's edge information, check the edge dictionary to determine whether the edge is already existed in edge dictionary or not. If not, insert new edge into edge dictionary. Then, the edge ids are associated with their corresponding edges in graph's edge information. Edge dictionary is shown in Fig. 4. The procedure processing edge dictionary is shown as follow.

Id	Edge
1	<C,s,C>
2	<C,s,O>
…	…

Fig. 4. Edge Dictionary

```
Procedure InsertEdgeDict(edge_info(G), Edge_Dictionary)
-----------------------------------------------------------
For each edge $e_i \in$ edge_info(G) do
    If $\nexists$ $e_i \in$ Edge_Dictionary then
        Add new edge information
return Edge_Dictionary
```

B. Code Generation

A graph is represented holistically into a graph code that preserves the structural information 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 of the edge 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 graph code.
- Thirdly, using the edge id in the code reduces the time for finding automorphic or isomorphic graphs.

Most of the chemical graphs have a lot of common edges. So, edge dictionary uses little memory space. Edge dictionary and adjacent edge information are used to generate graph code. Graph code for graph G_1 is as follows:

$$c(G_1)=1\{(c)1,1,1,1,2,2\}\ 2\{(c)1,1,(0)2,2\}$$

The procedure for graph code generation is described as follow.

```
Procedure GenGraphCode(adj_edgeinfo (G), Edge_Dictionary)
-----------------------------------------------------------
c(G) := Ø
For each $e_{adj} \in$ adj_edgeinfo (G) do
    Get id( $e_{adj}$ ) from Edge_Dictionary
    Find connected vertex v of $e_{adj}$
    Add these information to c(G)
return c(G)
```

C. Automorphic Detection

After computing the graph code of G_k, compares it with each graph code G_i in code store (*CS*), $1 <= i < k$, to check automorphism. If the graph code of G_k has the same code as that of G_i, concludes that the graphs are automorphic and append id of G_k to corresponding graph code. Otherwise, add the graph code of G_k to *CS* assuming as G_k is a new graph. Procedure for automorphic detection is as follow.

```
Procedure DetectAutomorphism(c($G_k$), CS)
-----------------------------------------------------------
If k = 1 then
    Add c($G_k$) to CS
    return CS
Else
For each c($G_i$) $\in$ CS
    If(c($G_i$) = c($G_k$)) then
        Append id of $G_k$ to c($G_i$)
    Else
        Add c($G_k$) to CS
return CS
```

D. Graph Isomorphism Query

Graph isomorphism query retrieves the graph in the dataset that is isomorphic to the given query graph. It can be responsively answered without candidate verification by using proposed graph code. When the query is entered, it is transformed into query's graph code. This code is probed in trie of dataset graphs' codes. Trie is used as an index to narrow down the query response time and search space. Then, the dataset graph is returned that is isomorphic to query graph. Procedure for graph isomorphism query is shown as follow.

```
Procedure GraphIsomorphismQuery(q_graph, CS)

For each c($G_i$) ∈ $CS$
    Insert graph codes and ids into Trie
Generate q_graph code
Probe q_graph code in Trie
return result
```

The following algorithm describes the step-by-step process of our proposed approach.

```
Algorithm 1 Proposed Approach

Input: $G_k$
Output: $CS$, Edge_Dictionary
For each $G_k$
  Generate graph information
  Edge_Dictionary:=InsertEdgeDict(edge_info($G$),Edge_Dictionary)
  c($G_k$) := GenGraphCode(adj_edgeinfo($G_k$), Edge_Dictionary)
  $CS$ := DetectAutomorphism(c($G_k$), $CS$)
q_result := GraphIsomorphismQuery(q_graph, $CS$)
return
```

V. Experimental Results

In this section, we present our experimental studies that validate the effectiveness and efficiency of our approach by comparing it with other existing methods. AIDS antiviral screen dataset is used in the evaluation to test the effectiveness of our proposed approach. All the experiments are performed on an Intel(R) core(TM) i3-4010u 3.0GHz PC with 2GB RAM, running the windows 32-bit operating system. All of the methods are implemented in java.

A. *AIDS Antiviral Screen Dataset*

The experiments described use the AIDS antiviral screen dataset (AIDS for short). This dataset can be available publicly. AIDS contains around 42,000 chemical compounds. The graphs have average number of 25 vertices and 27 edges and maximum number of 222 vertices and 251 edges. The total number of distinct vertex labels is 62. The major portions of vertices are C, O and N. All of the graphs are sparse graphs. Most of the chemical compound datasets are in the form of xml files and image files. So, the required storage space between proposed graph code, xml file and image file are evaluated. In Fig. 5, it can be seen that the storage space of our graph code is significantly less than the other two well-known storage formats.

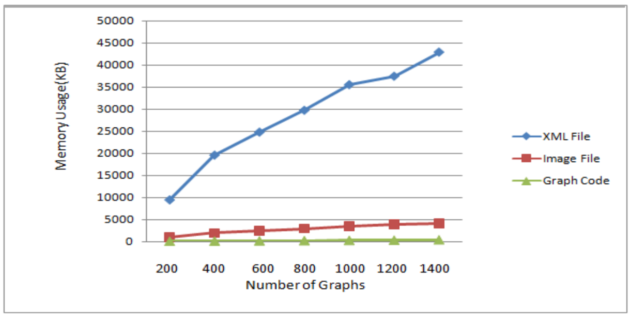

Fig. 5. Analysis of Storage Space between Graph Codes, XML Files and Image Files

In proposed approach, computational time complexity for generating graph code takes O($NE+(E*(E-1))/2$) where V is the number of vertices, E is the number of edges and N is the size of edge dictionary. Proposed approach takes NE comparisons to insert edges into edge dictionary and get edge id from it. For getting adjacent edge information, $(E*(E-1))/2$ comparisons are required. Table. I describes the computational time complexity between three methods for generating graph representative structure such as canonical label, grid code by F-GAF and our proposed graph code. Table. II shows the analysis of computational time complexity for code generation between three methods. 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 in code generation.

TABLE I. Computational Time Complexity for Code Generation between Three Methods

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

TABLE II. Analysis of Computational Time Complexity for Code Generation between Three Methods

No. of vertex	No. of edge	Canonical Labeling	F-GAF	Graph Code(N=19)
10	10	$3.63x10^6$	4.00x10	$2.35x10^2$
40	40	$8.16x10^{47}$	$1.60x10^2$	$1.54x10^3$
80	80	$7.20x10^{118}$	$3.20x10^2$	$4.68x10^3$
120	120	$6.70x10^{198}$	$4.80x10^2$	$9.42x10^3$
160	160	$4.70x10^{284}$	$6.40x10^2$	$1.58x10^4$
200	200	$7.89x 10^{374}$	$8.00x10^2$	$2.37x10^4$

Table. III shows the computational time complexity for the whole process of code generation and automorphic detection between canonical labeling, F-GAF and proposed approach where k is number of graphs and assumes k = 100. The analysis of comparisons required by three methods is shown in Table.

IV. Although our proposed graph code consumes a little more time in code generation than F-GAF, it can significantly seen that proposed approach can reduce at least 10^2 times computational complexity for the whole process of code generation and automorphic detection when compared to canonical labeling and F-GAF.

TABLE III. COMPUTATIONAL TIME COMPLEXITY FOR WHOLE PROCESS OF CODE GENERATION AND AUTOMORPHIC DETECTION

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

TABLE IV. ANALYSIS OF COMPUTATIONAL TIME COMPLEXITY FOR WHOLE PROCESS OF CODE GENERATION AND AUTOMORPHIC DETECTION

No. of vertex	**No. of edge**	**Canonical Labeling**	**F-GAF**	**Graph Code(N=19)**
10	10	$3.63x10^6$	$6.25x10^6$	$1.62x10^4$
40	40	$8.16\ x10^{47}$	$1.00x10^8$	$3.06x10^5$
80	80	$7.20x10^{118}$	$4.00x10^8$	$1.25x10^6$
120	120	$6.70x10^{198}$	$9.00x10^8$	$2.84x10^6$
160	160	$4.70x10^{284}$	$1.60\ x10^9$	$5.07x10^6$
200	200	$7.89x\ 10^{374}$	$2.50x10^9$	$7.94x10^6$

Table V. reports the analysis of code generation time between proposed approach and CGRS. From the empirical analysis, the code generation time varies depending on the number of graphs. Number of graphs between 200 to 1000 graphs is tested and takes the average code generation for comparison. Proposed approach consumes a little more time for preserving the structural information of the original graph such as two edges are connected on which vertex. CGRS consumes less time but their edge code can't preserve the structural information of original graph. It may be possible that many incorrect results will be returned.

TABLE V. ANALYSIS OF CODE GENERATION TIME IN MILLISECONDS

No. of graphs	**Proposed Approach**		**CGRS**	
	Total code generation time	Average code generation time	Total code generation time	Average code generation time
200	26516	132.58	25422	127.11
400	73274	183.19	70327	175.82
600	123322	205.54	121140	201.90
800	179680	224.60	176528	220.66
1000	256071	256.10	253791	253.79

Fig. 6 shows the analysis of the graph isomorphism query response time between proposed approach and CGRS. Both of the proposed approach and CGRS are implemented in java. Average query size is 100 and we apply chemical datasets with various sizes containing 100 to 40000 graphs. The experimental analysis of AIDS dataset reveals an optimistic performance of proposed approach over CGRS in graph isomorphism query.

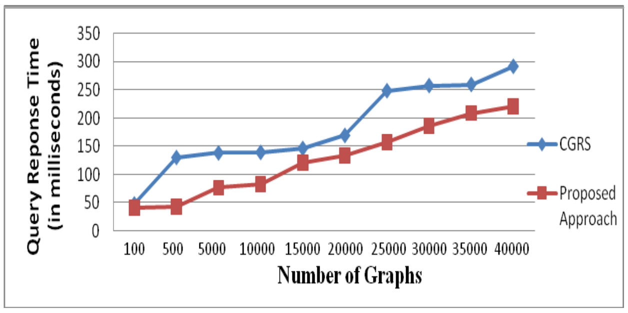

Fig. 6. Analysis of Graph Isomorphism Query Respose Time between Proposed Approach and CGRS

VI. CONCLUSION AND FUTURE WORKS

Instead of generating all possible permutation matrices or long grid code, proposed approach uses trie of data graphs' codes to detect automorphic graphs and retrieve isomorphic graph. The edge dictionary is used to narrow down the search space of graph code. Trie is also efficiently used to reduce search space and query response time. From our experimental results, proposed approach outperforms the existing methods in automorphic detecting and graph isomorphism querying. Subgraph, supergraph and similarity query are going to be observed as future works.

Acknowledgment

This research was supported by University of Computer Studies, Yangon and University of Information Technology. We thank our colleagues who provided insight and expertise that greatly assisted the research.

References

[1] A. Inokuchi, T. Washio, and H. Motoda, "Complete mining of frequent patterns from graphs", Mining graph data, Machine Learning 50(3), 2003, pp 321-354.

[2] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation", In Proc. of ACM SIGMOD Int. Conf. on Management of Data, Dallas, Taxas, May 2000.

[3] R. N. Chittimoori, L. B. Holder, and D. J. Cook, "Applying the SUBDUE substructure discovery system to the chemical toxicity domain", In Proc. of the 12th international Florida AI, Research Society Conference, 2003, pp 90-94.

[4] D. A. Piriyakumar, and P. Levi, "An Efficient A* based algorithm for optimal graph matching applied to computer vision", In GRWSIA-98, Munich, 1998.

[5] D. Dupplaw, and P. H. Lewis, "Content-based image retrieval with scale-spaced object trees", Proc. of SPIE: Storage and Retrieval for Media Databases, Volume 3972, 2000, pp 253-261.

[6] M. Kuramochi, and G. Karypis, “Frequent subgraph discovery”, Proc. 1st IEEE Int. Conf. on Data Mining (ICDM 2001), San Jose, CA, IEEE Press, Piscataway, NJ, USA, 2001, pp 313-320.

[7] 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.

[8] A. N. Thaing, and K. M. Oo, “CGRS: compact graph representative structure for efficient graph querying in chemical compound graph databases”, in Proceeding of the 12th International Conference on Computer Applications, 2014.

[9] https://en.wikipedia.org/wiki/Graph_Automorphism

[10] https://en.wikipedia.org/wiki/Graph_isomorphism

[11] C. A. James, D. Weininger, and J. Delany, “Daylight theory manual daylight version 4.82. daylight chemical information system”, Inc, 2003.

[12] R. C. Read and D. G. Corneil, “The graph isomorph disease”, Journal of Graph Theory, 1977, pp 339-363.