# Graph Querying Using Graph Code and GC_Trie

Yu Wai Hlaing
*University of Computer Studies (Yangon)*
*yuwaihlaing@ucsy.edu.mm*

## Abstract

*Graph datasets face a great challenge arising from a massive increasing volume of new structural graphs in bio-informatics, chem-informatics, business processes, etc. One of the essential functions in graph dataset is to query graph effectively and efficiently. Given a graph query, it is desirable to retrieve relevant graphs quickly from a graph dataset via efficient graph indices. In our proposed system, there are two phases: index constructing and graph querying. In index constructing phase, a graph is represented holistically via graph code that is constructed based on adjacent edge information and edge dictionary. The GC_Trie is constructed as index with graph codes of data graphs. In graph querying phase, automorphic(duplicate) graphs and isomorphic graphs of the query graph are queried by using GC_Trie. AIDS antiviral screen compound dataset is used to test the effectiveness of proposed approach. The experimental results offer a positive response to our newly proposed approach.*

## 1. Introduction

Graphs have become increasingly important in modeling complicated structures and schemaless data such as proteins, circuits, images, Web, and XML documents. Conceptually, any kind of data can be represented by graphs. It is inefficient to perform a sequential scan on a large graph dataset and check each graph to find answers of a query graph. Sequential scan is costly because one has to not only access the whole graph dataset but also check automorphic and isomorphic graphs. Therefore, high performance graph indexing is needed to quickly prune graphs that obviously violate the query requirements.

It is apparent that the success of any graph database application is directly dependent on the efficiency of the graph indexing and query processing mechanisms. Recently, there are many approaches that have been proposed to tackle these problems. The conditions for an efficient graph indexing approach are: not too expensive (in terms of space and time), high prune capability, and absence of false negatives. Various indexing approaches have been developed to process graph queries.

A common and critical issue lies in many graph-related applications is the efficient process of graph queries and retrieval of related graphs. In principle, queries in graph datasets can be broadly classified into the following four main categories:

- Graph isomorphism query
- Graph automorphism query
- Supergraph query, and
- Similarity query

In our proposed approach, automorphic and isomorphic graphs can be queried on AIDS antiviral screen compound dataset, where queries use the graph representation of chemical entities as XML format.

The remainder of the paper is organized as follows: related work is presented in section 2. Section 3 defines the preliminary concepts, section 4 describes our index constructing phase, query processing phase is presented in section 5, section 6 describes performance analysis of our approach and final conclusions are drawn in section 7.

## 2. Related Work

Over the years, a number of different representative structures have been developed to represent graphs more and more efficiently. 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.

In [1], canonical code is proposed for representing graph. Canonical label of a graph is as the string obtained by concatenating the upper triangular entries of the graph’s adjacency matrix when this matrix has been symmetrically permuted such that this string is the lexicographically largest (or smallest) among the strings obtained from all such permutations. 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.

Fast-Graph Automorphic Filter (F-GAF)[2] was proposed for detection and elimination of automorphic graphs in graph dataset. F-GAF uses edge-based graph representation called grid code. It involves three main phases: preprocessing, feature extraction and pattern matching. For large graphs, the length of grid code can be very long. It has overhead of space and expensive processing time.

GraphGrep[3] was proposed that is a path-based technique to index graph datasets. It has three basic components: building the index to represent graphs as sets of paths, filtering dataset based on query and computing exact matching. GraphGrep enumerates paths up to a threshold length from each graph. An index table is constructed and each entry in the table is the number of occurrences of the path in the graph. The “path-representation” of a graph is the set of label-paths in the graph, where each label-path has a set of id-paths. The keys of the hash table are the hash values of the label-paths. Each row contains the number of id-paths associated with a key (hash value) in each graph. This hash table is referred to as fingerprint of the dataset. Filtering phase generates a set of candidate graphs for which the count of each path is at least that of the query. Verification phase verifies each candidate graph by subgraph matching. However, the graph dataset contains huge amount of paths and can have an effect on the performance of the index.

He and Singh develop a tree-based index called Closure-Tree, or C-tree[4], in which each node of the tree contains discriminative information about its descendants to facilitate effective pruning work. The summary information is like a “bounding box” of the structural information of the constituent graphs and it is represented as a graph closure. In C-tree indexing approach, the summary graph in the index structure is a generalized graph which is a structural union of the underlying database graphs. A pair wise graph comparison performed through heuristic techniques. For subgraph, the subgraph isomorphism problem is handled by an approximation technique called pseudo subgraph isomorphism. And for similarity queries, graph similarity is defined based on edit distance and then compute it using heuristic graph mapping methods.

Our graph code[5] is developed by using edge dictionary and adjacent edge information to retain original graph structure, to save storage space and to facilitate the efficient processing of graph queries in graph dataset[6]. GC_Trie is proposed as an index structure to reduce expensive path or subgraph decomposition which could result in structural information lost and expensive enumerating time and to support graph queries efficiently with no additional indexing works for each type of graph queries[7]. Based on the graph code and GC_Trie, algorithms for graph isomorphism query, automorphism query are proposed to quickly identify graph relationships between the query graph and dataset graphs.

## 3. Preliminary Concept

For simplicity, the key concepts, notations, and terminology used in our proposed approach which includes labeled undirected graph, subgraph isomorphism, and graph automorphism are presented according to graph theory[8].

**Definition 1**. Labeled Undirected Graph

A labeled undirected graph $G$ is defined as 5-tuple, $(V, E, L_V, L_E, l)$ where $V$ is the non-empty finite vertex set called vertices, and $E$ is the unordered pairs of vertices called edges. $L_V$ and $L_E$ are the set of labels of vertices and edges and $l$ is a labelling function assigning a label to a vertex $l{:}V \rightarrow L_V$ or an edge $l{:}E \rightarrow L_E$.

**Definition 2.** Subgraph 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

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

## 4. Index Constructing Phase

In index construction phase, there are four steps: preprocessing, code generation, automorphic graph detection, and index construction. In preprocessing, the graph information such as vertex information, edge information and adjacent edge information are generated from XML files. In code generation step, a graph is represented into a graph code that preserves the structural information of the graph. In graph automorphism detection step, duplicate graphs are detected by comparing graph code of the newly graph with already inserted graph codes of the dataset graph in code store(*CS*). GC_Trie is constructed in index construction step using the graph codes of dataset graph is *CS*.

## 4.1 Preprocessing

The edge in the graph is defined as ($V_{id}$, $l$, $V_{id}$) where $V_{id}$ is vertex id such as 1,2,3 and so on, $l$ is edge label such as single bond (s), double bond (d) and triple bond (t). Then the adjacent edge information is generated from vertex information and edge information. Figure 1 shows potassium;1,2,4-dithiazolidine-3,5-dione compound. Vertex and edge information of this compound is shown in Figure 2.

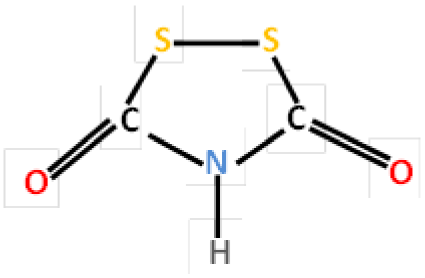
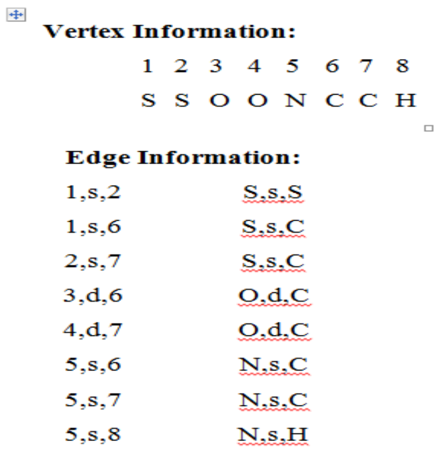


**Figure 1. Potassium;1,2,4-dithiazolidine-3,5-dione Compound**



**Vertex Information:**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| S | S | O | O | N | C | C | H |

**Edge Information:**

| | |
|---|---|
| 1,s,2 | S,s,S |
| 1,s,6 | S,s,C |
| 2,s,7 | S,s,C |
| 3,d,6 | O,d,C |
| 4,d,7 | O,d,C |
| 5,s,6 | N,s,C |
| 5,s,7 | N,s,C |
| 5,s,8 | N,s,H |

**Figure 2. Vertex and Edge Information of Potassium;1,2,4-dithiazolidine-3,5-dione Compound**

When most of the chemical structures share common edges, storing the edges of the chemical structures into edge dictionary is more efficient. The storage space of edge dictionary is more compact when most of the edges in the chemical structures are identical. The edge dictionary contains unique edge with global unique id to represent all the edges of all chemical structures in dataset. Potassium;1,2,4-dithiazolidine-3,5-dione compound shown in Figure 1 contains eight edges such as <S,s,S>, <S,s,C>, <S,s,C>, <O,d,C>, <O,d,C>, <N,s,C>, <N,s,C>, and <N,s,H> as edge information. If the edge is already defined in edge dictionary, it is filtered out. In this way, the edge dictionary can save the storage space. There are five distinct edges in this compound. These edges are <S,s,S>, <S,s,C>, <O,d,C>, <N,s,C>, and <N,s,H>. These distinct five edges are inserted into edge dictionary. Table 1 shows the edge dictionary.

**Table 1. Edge Dictionary**

| id | Edge |
|---|---|
| 1 | <S,s,S> |
| 2 | <S,s,C> |
| 3 | <O,d,C> |
| 4 | <N,s,C> |
| 5 | <N,s,H> |

Figure 3 shows procedure 1 which is the step by step procedure of processing edge dictionary. The input to the procedure is edge information. The distinct edge is inserted into the edge dictionary according to the steps in the following procedure.

```
Procedure1
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
```

**Figure 3. Procedure of Processing Edge Dictionary**

## 4.2 Code Generation

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 of graph querying.

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. For a graph $G$, the graph code of $G$, denoted by $c(G)$ is the list of the form $e_{id}[(v), e_{id}], \ldots$ depending on adjacent edges and edge dictionary where $e_{id}$ is the edge id, $v$ is vertex label on which two edges are connected. Figure 4 shows the procedure 2 for generating graph codes of the graphs is described follows:

```
Procedure2
GenGraphCode(adj_edgeInfo(G),Edge_Dictionary)
c(G) := Ø
For each e_adj ∈ adj_edgeInfo (G) do
    Get id( e_adj ) from Edge_Dictionary
    Find connected vertex label v of e_adj
    Add these information to c(G)
return c(G)
```

**Figure 4. Procedure of Generating Graph Code**

According to the procedure in Figure 4, the graph code of the potassium;1,2,4-dithiazolidine-3,5-dione compound is [1][s,2][1][s,2][2][s,1][2][c,3][2][c,4][2][s,1][2][c,3][2][c,4][3][c,2][3][c,4][3][c,2][3][c,4][4][c,2][4][c,3][4][n,4][4][n,5][4][c,2][4][c,3][4][n,4][4][n,5][5][n,4][5][n,4].

## 4.3 Graph Automorphism Detection

After computing the graph code of $G_k$, compares it with each graph code $G_i$ in $CS$, $1 <= i < k$, to check graph automorphism. If the graph code of $G_k$ has the same code as that of $G_i$, concludes that the two graphs are automorphic and append id of $G_k$ to corresponding graph code of $G_i$. Otherwise, add the graph code of $G_k$ to $CS$ assuming as $G_k$ is a new graph. We have already described automorphism detecting using graph code in [9]. Procedure 3 is for automorphic detection and is shown in Figure 5 as follow.

```
Procedure 3
DetectAutomorphism(c(G_k), CS)
If k = 1 then
    Add c(G_k) to CS
return CS
Else
    For each c(G_i) ∈ CS
    If(c(G_i) == c(G_k)) then
        Append id of G_k to id of c(G_i)
    Else
        Add c(G_k) to CS
return CS
```

**Figure 5. Procedure of Automorphic Detection**

## 4.4 Index Construction

Instead of using path or subgraph decomposition, GC_Trie is proposed. The graph codes of all dataset graphs are put in GC_Trie. A GC_Trie is a trie where each node except the root node is a string array that represents an edge id or a vertex label on which two edges are connected. There are five levels in the GC_Trie. The second and fourth level is for edge ids. The third level is for vertex labels and the last is for leaves which are implemented by hashmaps of graph ids and their frequencies. Procedure 4 for index construction is shown in Figure 6. We represent one edge's adjacent code, e.g; 1[o,2], as feature $f$.

```
Procedure 4
IndexConstruction(CS)
For each c(G_i) ∈ CS
    For each feature f ∈ c(G_i)
        Put f in GC_Trie
return GC_Trie
```

**Figure 6. Procedure of Index Construction**

## 5. Graph Querying Phase

In this phase, if any isomorphic graphs of the query graph are retrieved by using GC_Trie as an index. Graph isomorphism query retrieves the data graphs from the dataset that is equal to the given query graph. It can be responsively answered without candidate verification through the use of proposed graph code. When the query graph enters, the query is transformed into a query graph code. The query's graph code is probed into the GC_Trie to find the exact match structures. Then, the data graph is returned that matches with features of the query graph. The algorithm 1 in Figure 7 describes the step-by-step process for graph isomorphism query.

**Algorithm 1 GraphIsomorphismQuery**

```
Input : GC_Trie, and Query $q$
Output : Answer set $D_q$.
    1.  Generate graph code for query $q$, c($q$).
    2.  Let $D_q = D$
    3.  For each feature $qf \in$ c($q$)
    4.  Probe $qf$ in GC_Trie.
    5.  If $qf \in$ GC_Trie
    6.  $D_q = D_q \cap D_{qf}$
    7.  For each $G_i \in D_q$
    8.      If size($G_i$) != size($q$)
    9.          Remove $G_i$ from $D_q$
    10. Return $D_q$;
```

**Figure 7. Proposed Algorithm for Graph Isomorphism Query**

## 6. Performance Analysis

A comprehensive study of storage space for proposed graph code and other existing graph representative structures were conducted on AIDS antiviral screen compound dataset. This dataset is available in the format of Text ASN.1 files, XML files, SDF files, and image (.png) files via PubChem BioassaysDatabase(http://pubchem.ncbi.nlm.nih.gov/ ). All of the chemical compound graphs from this dataset are large number of sparse graphs. A sparse graph is a graph in which the number of edges is closely the same as the number of vertices. Storage space on Text ASN.1 files, XML files, SDF files, image (.png) files, OrientDB, GraphGerp and our proposed graph codes are analyzed. For GraphGrep, two values 4 and 10 are used for maximum path length parameter lp. Figure 8 shows the analysis of storage space to allocate different graph storage structures for AIDS antiviral screen compound dataset.

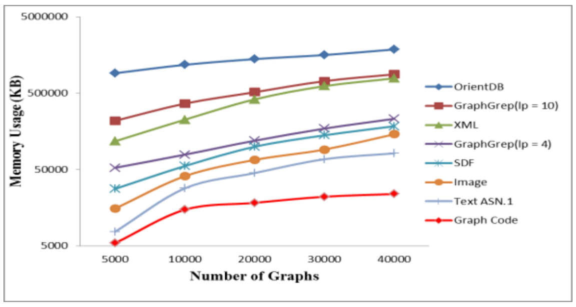


**Figure 8. Analysis of Storage Space between Different Formats on AIDS Antiviral Screen Compound Dataset**

For proposed graph representative structure, the computational time complexity for generating graph code takes O(*NE+(E*(E-1))/2*) where *V* is the number of vertices in the graph, *E* is the number of edges in the graph and *N* is the size of edge dictionary. Our approach takes *NE* comparisons to insert edges into edge dictionary and get edge id from it. For getting adjacent edge information and generating graph codes, *(E*(E-1))/2* comparisons are required.

Table 2 describes the computational time complexity between three methods for generating graph representative structure such as canonical labeling by FSG, grid code by F-GAF and our proposed graph code.

**Table 2. Computational Time Complexity for Code Generation**

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

Table 3 shows the analysis of computational time complexity for code generation between three methods. Graph code can reduce at least $10^{44}$ times computational complexity when compared to canonical labeling. But it consumes a little more time than F-GAF in code generation.

**Table 3. Analysis of Computational Time Complexity for Code Generation**

| No. of vertex | No. of edge | Canonical Labeling | F-GAF | Graph Code with *N*=19 |
|---|---|---|---|---|
| 40 | 40 | $8.16 \times 10^{47}$ | $1.60 \times 10^{2}$ | $1.54 \times 10^{3}$ |
| 80 | 80 | $7.20 \times 10^{118}$ | $3.20 \times 10^{2}$ | $4.68 \times 10^{3}$ |
| 120 | 120 | $6.70 \times 10^{198}$ | $4.80 \times 10^{2}$ | $9.42 \times 10^{3}$ |
| 160 | 160 | $4.70 \times 10^{284}$ | $6.40 \times 10^{2}$ | $1.58 \times 10^{4}$ |
| 200 | 200 | $7.89 \times 10^{374}$ | $8.00 \times 10^{2}$ | $2.37 \times 10^{4}$ |

The query response time of proposed approach with GraphGrep is also evaluated in figure 9. Since GraphGrep only support graph isomorphism query, only graph isomorphism query response time can be evaluated with it. For GraphGrep, two values 4 and

10 for parameter; the length of path (lp) are used. Figure shows the analysis of graph isomorphism query response time over AIDS antiviral screen compound dataset. It can be seen that proposed approach significantly reduces at least $10^2$ times for graph isomorphism query response time when compare to GraphGrep.

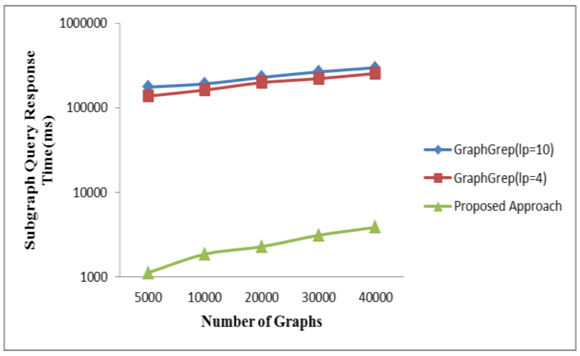


**Figure 9. Analysis of graph isomorphism Query Response Time on AIDS Antiviral Screen Compound Dataset**

## 7. Conclusion

Graph code used edge dictionary and adjacent edges information to preserve the structural information of the original graph. Instead of expensive pair-wise comparisons, graph code can be efficiently used to detect automorphic graphs. Instead of path or subgraph decomposition process which could result in structural information lost and exhausted enumeration time, GC_Trie is used. An efficient graph querying approach using graph representative structure called graph code and trie structure called graph code_trie (GC_Trie) is implemented. The experimental results and comparisons offer a positive response to our approach.

## Reference

**[1]** M. Kuramochi, and G. Karypis, “An Efficient Algorithm for Discoverying Frequent Subgraphs”, In Proceeding of 2002 International Conference on Data Mining (ICDM’02), San Jose, CA, November, 2002, pp. 313-320.

**[2]** R. Vijayalakshmi, R. Nadarajan, P. Nirmala, and M. Thilaga, “A Novel Approach for Detection and Elimination of Automorphic Graphs in Graph Database”, International Journal of Open Problems Compt. Math., Vol. 3, No. 1, March, 2010.

**[3]** R. Giugno, and D. Shasha, “A Fast and Universal Method for Querying Graphs”, In Proceeding of International Conference on Pattern Recognition, 2004.

**[4]** H. He, and A. K. Singh, “Closure-tree: An Indexing Structure for Graph Queries”, In Proceeding of the 22th IEEE International Conference on Data Engineering (ICDE’06), Atlanta, pp. 38-49, 2006.

**[5]** Y. W. Hlaing, and K. M. Oo, “A Graph Representative Structure for Efficient Querying”, In Proceeding of 13th International Conference on Computer Application (ICCA’15), Yangon, Myanmar, February, 2015, pp. 111-114.

**[6]** X. Yan, P. Yu, and J. Han, "Mining, Indexing and Similarity Search in Large Graph Data Sets", 2006.

**[7]** S. Sakr, and G. Al-Naymat, "Graph Indexing and Querying:a review", International Journal of Web Information Systems, Vol. 6, No. 2, pp. 101-120, 2010.

**[8]** K. Ruohonen, "Graph Theory", 2013.

**[9]** Y. W. Hlaing, and K. M. Oo, “A Graph Representative Structure for Detecting Automorphic Graphs”, In Proceeding of the 9th International Conference on Genetic and Evolutionary Computing (ICGEC’15), Yangon, Myanmar, August, 2015, pp. 189-197.