A Graph Representative Structurefor Efficient Querying

Yu WaiHlaing, Kyaw May Oo
University of Computer Studies, Yangon
yuwaihlaing.1987@gmail.com

Abstract

Graphs are prevalently used to model the relationships between objects in various domains. With the increasing usage of graph data, it has become more and more demanding to efficiently process graph queries. Querying graph data is costly since it involves exhausted graph isomorphism testing. In Ph.D research work, we propose a graph representation and querying approach. There are 2 main phases in our approach: code generation phase and query phase. In this paper, an efficient graph representative structure that is generated by code generation phase is presented. In code generation phase, there are two main processes: preprocessing of graph data and code generation. Our proposed graph representative structure is called graph code.Edge dictionary is efficiently used to narrow down the search space of graph codes. To generate graph code, edge dictionary and adjacent edge information are used.Our experiment also shows that for storage space required by our graph code is smaller than other formats such as XML file and image file.

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.

Structures that can be represented as graphs are based on graph theory. 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*×*n adjacency matrix A. Ai j* is 1 if (*i*, *j*) ∈*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. Our graph code is a new way of representing graph data to process graph queries without verifying 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 discusses about related work. Preliminaries are presented in section 3. In section 4, graph representative structures including our graph code are explained.Experimental results are shown in section 5. And then conclusion is in section 6.

2. Related Work

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.

MichihiroKuramochi et al. proposed a canonical labeling [5] 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 [6].

R Vijayalakshmi et al. [7] 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 disadvantage of this structure is it requires exhausted enumeration to generate grid code.

3. Preliminaries

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

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

In this paper, the size of a graph is defined as the number of its edges. An example labeled graph is shown in figure 1.

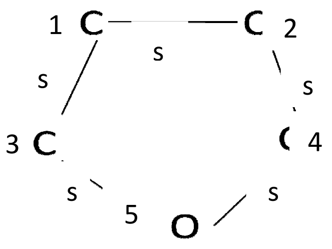


Figure 1. A Labeled Graph

Definition 2(Graph Isomorphism) Two graphs G1 = (V1, E1) and G2 = (V2, E2) are isomorphic if they are topologically identical to each other, that is, there is a mapping from V1 to V2 such that each edge in E1 is mapped to a single edge in E2 and vice versa.

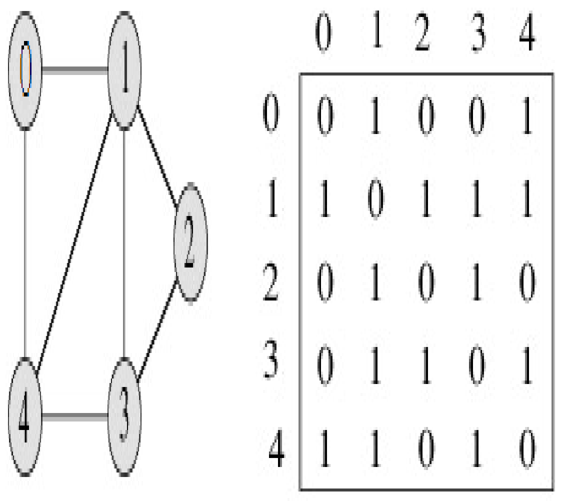
Definition 3(Graph Code Representation) For a graph G, the code of G, denoted by code(G) is in the form e_{id}\{(v),eid_{adj},…\}… depending on adjacent edge information of preprocessing. e_{id} is the edge id, v is vertex label on which two edges are connected, eid_{adj} is list of adjacent edge ids for this edge.

4. Graph Representative Structures

A graph data structure consists of a finite set of vertices, together with a set of ordered pairs of these nodes. These pairs are known as edges.Traditionally, the two most commonly used graph representative structures are adjacency matrix, and adjacency list. Graph can also be represented as XML representation and image representation (.png,.jpg,.gif).

4.1. Adjacency Matrix

An adjacency matrix is a means of representing which vertices of a graph are adjacent to which other vertices. the adjacency matrix of a finite graph G on *n* vertices is the n × n matrix where the non-diagonal entry a_{ij} is the number of edges from vertex i to vertex j, and the diagonal entry a_{ii}, depending on the convention, is either once or twice the number of edges from vertex i to itself. Figure 2 shows an example graph and its adjacency matrix.



$$\begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 2 & 0 & 1 & 0 & 1 & 0 \\ 3 & 0 & 1 & 1 & 0 & 1 \\ 4 & 1 & 1 & 0 & 1 & 0 \end{array}$$

(a) **(**b)

Figure 2. (a) An Example Graph G (b) Its Adjacency Matrix

4.2. Adjacency List

An array of linked lists is used. 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 theith vertex.
Figure 3 shows an adjacency list representation of G.

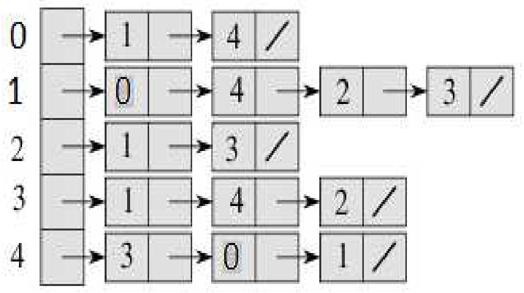


Figure 3. Adjacency List Representation of G

4.3. XML Representation and Image Representation

Most of the chemical compound graph dataset are available in XML filesand image files. Figure 4shows an XML representation for graph G1.

```
<?xmlversion="1.0"encoding="UTF-8"?>

<graph mode="static"defaultedgetype="directed">

<nodes><node id="1" label="C" />

         <node id="2" label="C" />

         <node id="3" label="C" />

         <node id="4" label="C" />

         <node id="5" label="O" /></nodes>

<edges><edge source="1" target="2" label=”s”/><

         <edge source="1" target="3" label=”s”/>

         <edge source="2" target="4" label=”s”/>

         <edge source="3" target="5" label=”s”/>

         <edge source="4" target="5" label=”s”/></edges>

</graph>
```

Figure 4. XML Representation for Graph G1

4.4. Our Graph Code

In this section, we explain step-by-step procedure of generating our graph code in code generation phase. It involves two main processes: preprocessing and code generation.

4.4.1. Preprocessing

In this step, the input xml file is parsed using xml parser. 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

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

label.

4.4.2. 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 graph processing. Figure 5 shows the example edge dictionary for input graphs.

Id	Edge
1	<C,s,C>
2	<C,s,O>
...	…

Figure 5. Edge Dictionary

4.4.3. 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. For each edge e, the adjacent edges of *e* are investigated in the graph where the identifiers of the adjacent edges are the global edge identifiers 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.

Edge dictionary and adjacent edge information are used to generate graph code. Graph codes of the

input graphs are stored in graph code store. An example graph code of G1 is:
code(G1)=1{(c)1,(c)1}1{(c)1,(c)2} 1{(c)1,(c)2} 2{(c)1,(0)2} 2{(c)1,(0)2}.

5. Experimental Result

The experiment described in this section use the AIDS Antiviral Screen Dataset. The experiment is done on the Intel Core i5 with 2GB main memory.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.The total number of distinct vertex labels is 62. The major portions of vertices are C, O, N. 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 figure 6, it can be seen that the storage space of our graph codes is significantly less than the other two formats.

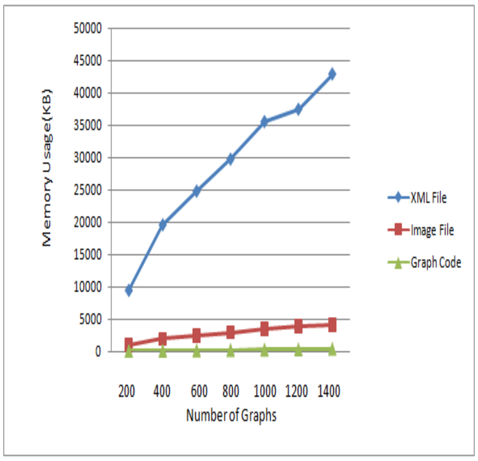


Figure 6. Analysis of Storage Space for Various Numbers of Graphs between Graph Code, XML File, Image File

6. Conclusion and Future Work

Our goal is to find graph isomorphism query in graph dataset efficiently.The edge dictionary is used to narrow down the search space. Because most of the graphs in graph dataset mostly contain similar edges. By representing graphs as graph codes, the storage space can be greatly reduced. Our approach can be extended to labeled directed graphs and graph querying 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 8th 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*, December 1991, Vol. 32, No. 12, pp. 1979-1998.

[4] S. Raghavan, and H. Garcia-Molina, “Representing Web Graphs”, In Proceeding of IEEE International Conference on Data Engineering, 2003.

[5] 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.

[6] S. Fortin, “The graph isomorphism problem”, Technical Report TR 96-20, Department of Computing Science, University of Alberta, 1996.

[7] 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.