An Efficient Indexing Mechanism for Graph Queries in Graph Databases

Aye Nwe Thaing
University of Computer Studies, Yangon, Myanmar
ayenwethaing@gmail.com

Abstract

In recent years, graph has become a powerful tool for representing and modeling objects and their relationships in various application domains such as protein interactions, chemical compounds, social networks, XML documents and so on. The volume of graph data increases rapidly and the graph database becomes an essential role to store graph data. However, the performance of query processing on graph databases is still inadequate due to the high complexity of processing graph data. As a result, it is important to develop efficient indexing structure for query processing on the graph databases. In this paper, we propose both algorithms for graph indexing and subgraph isomorphism query for subgraph query processing in the graph database. We also propose a proficient index structure (AdE) to support both algorithms to quickly index and identify the isomorphic graphs for the given query graph. Like canonical code, AdE checks whether the query graph is a subgraph isomorphic to the database graph. Our proposed index structure significantly reduces the computational time complexity compared to the DGIndex structure.

Keywords

Graph Database, Graph Decomposition, Graph Indexing, Graph Query Processing, Chemical Compound

1. Introduction

Graphs have appeared in many applications to represent social networks, web site link structures and other. 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. The graph database contains large amount of graphs and a graph handles billions of nodes and relationships.

Existing research has been carried out on two types of graph databases. The first type of graph database consists of very large graphs, such as the Web graph and social networks. Typical querying tasks for such graph databases include finding the best connection between a given set of query nodes[4,8] and finding subgraphs that match a given query pattern. The second type graph database consists of a large set of small graphs such as chemical compounds [2,3]. This kind of databases is most useful in scientific domains such as chemistry and bio-informatics. The querying tasks for this type of database include subgraph queries and similarity queries. In this paper, we also focus on this type of database.

For subgraph queries, one looks for a specific pattern in the graph database. A matching condition is used to decide if a pattern subgraph occurs in a graph. Subgraph isomorphism test is one of the most commonly used matching condition which determines that a pattern P matches a graph G if and only if P is a subgraph of G. In this way, subgraph query identifies the occurrences of the query subgraph in the database graph. Subgraph queries are useful in a number of applications such as finding structural motifs in protein 3D structures, and pathway discovery in protein interaction graphs [1].

Second, one looks for graphs in a graph database that are similar to a query graph. This type of query is known as similarity query. For similarity queries, graph edit distance [6] is a common way to measure the similarity between two graphs. There are generally two kinds of similarity queries: K-NN query and range query[11,6]. K-NN query finds K nearest graphs to the query graph. Range query finds graphs within a specified distance to the query graph. Similarity queries are useful for applications such as schema matching and classification [1].

Query processing on graphs is challenging for a number of reasons. For subgraph queries, one faces the subgraph isomorphism problem, known to be NP-Complete. For similarity queries, it is difficult to give a meaningful definition of graph similarity.

In this paper, we propose an efficient indexing structure for structured graph decomposition and both algorithms for graph indexing and subgraph query. The indexing structure consists of two data structures: the directed acyclic graph (DAG) and the adjacent edge structure (AdE) . Our proposed AdE indexing structure considerably reduces the computational time complexity compared to GDIndex structure[11].

The rest of the paper is organized as follows. Section 2 represents the formal definitions and notations used in this paper. Section 3 discusses about the related work of graph isomorphism. Section 4

discusses about the GDIndex structure. Section 5 explains Fast-GDIndex structure. Section 6 discusses about the analysis and illustration of two index structures and section 9 discusses about conclusion.

2. Preliminaries Concepts

This section describes the formal graph definitions and notation used in this work.

Definition 1. Labeled Graph

A labeled graph G is defined as a 4-tuple $(\mathrm{V}, \mathrm{E}, \sum, l)$ where V is the set of vertices, $\mathrm{E} \in \mathrm{V} \times \mathrm{V}$ is the set of edges, $\sum$ is the set of labels of vertices and edges and $l\colon \mathrm{V} \cup \mathrm{E} \rightarrow \sum$ is a labeling function assigning a label to a vertex or an edge.

Definition 2. Graph Isomorphism

Let $\mathrm{G} = (\mathrm{V}, \mathrm{E}, \sum, \mathrm{l})$ and $\mathrm{G}' = (\mathrm{V}', \mathrm{E}', \sum', \mathrm{l}')$ be two graphs. A graph isomorphism from G to G' is an injective function $\mathrm{f}\colon \mathrm{V} \rightarrow \mathrm{V}'$ such that (1) $\forall \mathrm{u} \in \mathrm{V}$, $\mathrm{l}(\mathrm{u}) = \mathrm{l}'(\mathrm{f}(\mathrm{u}))$ (2) $\forall (\mathrm{u}, \mathrm{v}) \in \mathrm{E}, (\mathrm{f}(\mathrm{u}), \mathrm{f}(\mathrm{v})) \in \mathrm{E}'$ and $\mathrm{l}(\mathrm{u}, \mathrm{v}) = \mathrm{l}'\big(\mathrm{f}(\mathrm{u}), \mathrm{f}(\mathrm{v})\big)$.

Definition 3. Subgraph Isomorphism

Let $\mathrm{G} = (\mathrm{V}, \mathrm{E}, \sum, \mathrm{l})$ and $\mathrm{G}' = (\mathrm{V}', \mathrm{E}', \sum', \mathrm{l}')$ be two graphs. A subgraph isomorphism from G to G' is an injective function $\mathrm{f}\colon \mathrm{V} \rightarrow \mathrm{V}'$ such that (1) $\forall (\mathrm{u}, \mathrm{v}) \in \mathrm{E}, (\mathrm{f}(\mathrm{u}), \mathrm{f}(\mathrm{v})) \in \mathrm{E}'$ and (2) $\mathrm{l}(\mathrm{u}) = \mathrm{l}'(\mathrm{f}(\mathrm{u}))$ and (3) $\mathrm{l}(\mathrm{u}, \mathrm{v}) = \mathrm{l}'\big(\mathrm{f}(\mathrm{u}), \mathrm{f}(\mathrm{v})\big)$. G is subgraph isomorphic to G' is denoted as $\mathrm{G} \subset \mathrm{G}'$.

3. Related Work

Several indexing techniques have been developed for graph queries. H. He and A. K. Singh proposed the concept of a graph closure, a generalized graph that represents a number of graphs. C-Tree organizes graphs hierarchically where each node summarizes its descendants by a graph closure [6]. C-Tree can efficiently support both subgraph queries and similarity queries. For subgraph queries, pseudo subgraph isomorphism approximates subgraph isomorphism with high accuracy [6]. For similarity queries, graph similarity is measured through edit distance using heuristic graph mapping methods [6].

D. W. Williams, J. Huan and W. Wang introduced a method of indexing graph databases using Structured Graph Decomposition [11]. The index consists of two data structures: directed acyclic graph (DAG) and a hash table. DAG contains nodes and each of these nodes is the unique subgraph of the database graphs. The hash table contains the hash codes of canonical forms of the database graphs.

Isomorphism between graphs becomes active research area in graph query processing and automorphic graphs filtering [9]. Most frequent subgraphs mining and graph isomorphism problems employ canonical code of a graph to test whether two graphs are isomorphic or not [11,7]. A canonical form is to construct a code word that uniquely identifies a graph up to automorphisms. 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.

S. Zhang, J. Yang and W. Jin proposed subgraph indexing and approximate matching in large graphs (SAPPER). SAPPER method is introduced to find the occurrences of a query graph in a large database graph with noise (e.g., missing edge). Utilizing the hybrid neighborhood unit structures in the index, SAPPER takes the advantage of pre-generated random spanning trees and a carefully designed graph enumeration order.

4. GDIndex

An efficient indexing structure GDIndex using structured graph decomposition is presented in [11]. Comparing to the previous indexing methods for query processing on the graph databases, GDIndex employs the smart idea of graph decomposition, therefore, no candidate verification is needed. GDIndex consists of two data structures: directed acyclic graph (DAG) and a hash table using code-based canonical representation. In this section, we review the ideas of GDIndex and point out the bottlenecks in the computing of canonical code in terms of the computational time complexity.

4.1. Graph Decomposition DAG

The graph decomposition of a graph G is the representation of all possible subgraphs of G which are induced subgraphs of G. The graph decomposition of the directed acyclic graph (DAG) contains nodes and each node represents subgraph or graph in the graph database. For two nodes A and B, there is a directed link from node A to node B if $\mathrm{A} \subset \mathrm{B}$. The graph decomposition DAG contains both the original graph and a null graph.

4.2. Canonical Code

For the canonical code of an undirected graph, the graph is first represented by an adjacency matrix, M. Every diagonal entry of the adjacency matrix is filled up with the label of the corresponding node and every off-diagonal entry is filled up with the label of

the corresponding edge, if there is an edge between two vertices or zero if there is no edge.

Given an n x n adjacency matrix of a graph G with n nodes, the canonical code of M is defined as the sequence of lower triangular entries of M (including entries on the diagonal) in the order: $m_{1,1}$ $m_{1,2}m_{2,2}...m_{n,1}m_{n,2}...m_{n,n}$ where $m_{i,j}$ represents the entry at the ith row and j[th] column in M. The resulting codes are sorted lexicographically. Then, the maximal canonical code is chosen from all possible codes of a given graph. This maximal canonical code is the canonical representation of the graph.

For canonical code representation, if the graph contains |V| vertexes, the worst case time complexity to compute canonical code is O(|V|!). To overcome the computation of finding all possible canonical code of a graph, an efficient adjacent edge index structure(AdE) is proposed for structured graph decomposition.

5. Fast-GDIndex

The Fast-GDIndex contains two data structure: DAG and AdE. The DAG contains the decomposition of all graphs in the database. We model chemical structures as graphs and a chemical compound graph database is shown in figure 1. The graph decomposition DAG for the given graph database is shown in figure 2.

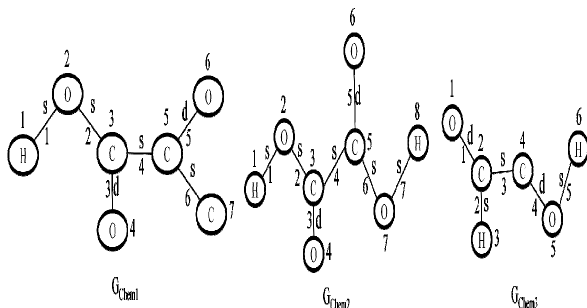

Figure 1. A chemical compound database

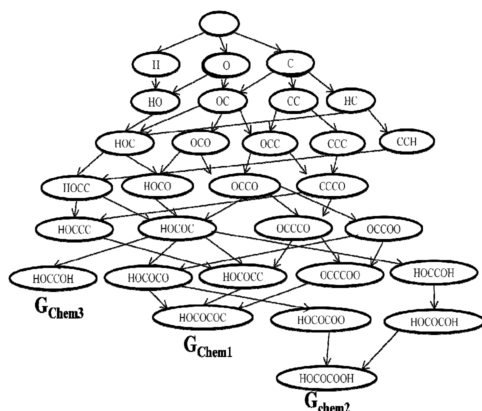

Figure 2. DAG of chemical compound database

Algorithm 1 demonstrates the construction of Fast-GDIndex of the database graphs. The notations used in this work is summarized in table 1.

Table 1. Notations used in Fast-DGI Algorithm

Notation	Definition
GDB	Graph Database
GEI	Global edge information
LAEI	Local adjacent edge information
AdE[DAG]	Adjacent edge information of graph contained in DAG
N[DAG]	Node contained in DAG
E[DAG]	Edge contained in DAG
G	Graph
e⟵((*l*(u),*l*(u,v),*l*(v))	Edge represents in the edge table
DAG	Directed acyclic graph
G_q	Subgraph query
N	No. of vertices in the graph

Algorithm 1 : Fast-GDI Construction

```
Input :GDB
Output:DAG, AdE(DAGn1,DAGn2,...,DAGnn)

Construct(GDB)
AdE:=Φ
DAG:=Φ
For each G Є GDB do
N[DAG]:= N[DAG] ∪ {G}
AdE[G]=G
AdE[DAG]:=AdE[DAG] ∪ AdE[G]
Decompose (G, DAG, AdE)
End do
Return (DAG,AdE)

Decompose(G,DAG,AdE)
For each v Є V[G] do
Gi:=G-v
N[DAG]:=N[DAG] ∪ {Gi}
E[DAG]:=E[DAG] ∪ {(Gi,G)}
AdE[Gi]=Gi
AdE[DAG]:=AdE[DAG] ∪ AdE[Gi]
Decompose (Gi, DAG,AdE)
End do
```

5.1. Adjacent Edge structure (AdE)

The adjacent edge structure is used to describe adjacent edge information for each node contained in the DAG. AdE structure consists of two steps: (1)

constructing global edge information table (GEI) for graph database and (2) creating local adjacency edge information (LAEI) for each graph in the database.

5.1.1. Global Edge Information (GEI)

The global edge information (GEI) contains all distinct edges of graphs in a graph database. Each edge in GEI is assigned with the global unique identifier. Therefore, it is efficient to retrieve the equivalent edge came along in the graph. In GEI, An edge e = (u,v) is defined as a tuple (l(u), l(u,v), l(v)) in the edge table where l(u) and l(v) are the labels of the vertices and l(u,v) is the label of the edge. Each edge appears only once in the global edge table, no matter how many times it appears in the graphs.

When a graph introduces in the graph database, we pick out all distinguishable edges in the graph and then we check whether these edges already existed or not in GEI. If the GEI contains these edges, we look up the corresponding identifier of that edge and use the identifier for further processing. If the edge does not already exist in the GEI, then put the edge into the GEI serially.

5.1.2. Local Adjacent Edge Information (LAEI)

The local adjacent edge information (LAEI) contains adjacent edges information for each edge appeared in the graph. LAEI uses the edge information from the GEI. From GEI, the edge identifier of the distinct edges contained in the graph is looked up. For each edge e= (l(u),l(u,v),l(v)), we finds the adjacent edges of e in the graph where the identifiers of the adjacent edges are the global edge identifiers in the GEI. Moreover, the local edge information for each graph is transformed into AdE (**Ad**jacent **E**dge) representation for further string comparisons efficiently.

5.2. Subgraph Isomorphism Queries

Subgraph isomorphism queries can be responsively answered through the use of fast-GDI structure. When the subgraph query enters, the query is transformed into AdE structure. Then the query's AdE structure is matched with AdEs of nodes in the DAG. When there is match to the query, then locates the nodes in the DAG and reports all descendents of the node which correspond to database graphs. If any of these parametric quantities in AdE are different, then the graphs are not isomorphic to each other, the answer set is empty.

Algorithm 2: Subgraph Isomorphism Query

```
Input  :Gq
Output :G1,G2,…,Gi which are supergraphs of Gq

Subgraph Isomorphism Search (Gq)
ans:=Φ
visited:=Φ
ade:=AdE(Gq)
if ade exists then Visit(ade, ans, visited)
return ans
Visit(ade, ans, visited)
Visited:=visited ∪ {ade}
If ade represents G then
ans:=ans ∪ {G}
end if
for each child ade' do
if ade' ∉ visited then Visit(ade',ans,visited)
end for
```

6. Analysis and Illustration

For a given graph database, the space requirement of DAG is the same for both techniques. The space O($k2^n$) is needed to represent the total number of nodes in the GDI where n is the maximum size of graph in graph database and k is the number of graphs in the database. A node in the GDI requires the space of O(n) for the worst case. Therefore, the space requirement for the DAG is O($kn(2^n)$).

6.1. Illustration of Canonical Code

The adjacency matrix of graph G_{Chem1} is shown in figure 3 and the corresponding canonical code is "HsO0sC00dO00s0C0000dO00000s0C".

All possible canonical codes of G_{Chem1} are 5040 distinct codes and each code contains 28 lengths. Therefore, the number of comparisons to find unique code is 141120.

Id	1	2	3	4	5	6	7
Label	H	O	C	O	C	O	C
1 H	H	s	0	0	0	0	0
2 O	S	O	s	0	0	0	0
3 C	0	s	C	d	s	0	0
4 O	0	0	d	O	0	0	0
5 C	0	0	s	0	C	d	s
6 O	0	0	0	0	d	O	0
7 C	0	0	0	0	s	0	C

Figure 3. Adjacency Matrix (G_{chem1})

Consider a graph database with k=100 graphs and n=7, the number of nodes in DAG is $O(k2^n)$= $(100x2^7)$= 12800 for the worst case. For this case, the computational time complexity to compute all canonical codes for 12800 nodes is 141120x12800=1806336000.

6.2. Illustration of AdE

The GEI contains many distinct edges in a graph database DB. The GEI of G_{Chem1} is shown in table 2. Assume G_{Chem1} is the first graph entering into the database.

Table 2. GEI of Graph G_{Chem1}

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

The LAEI contains adjacent edges information for each edge appeared in the graph. LAEI uses edge's information from the GEI. Table 3 shows LAEI of G_{Chem1}.

Table 3. LAEI of Graph G_{Chem1}

Edge ID	Edge Neighbour
1	1 < 3 >
2	3 5 < 3, 4 >, < 4,4 >
3	2 < 1, 2, 4 >
4	4 6 < 2, 2, 3, 4>, < 2, 4 >

We represent above LAEI of G_{Chem1} in term of AdE representation.

$AdE(G_{Chem1})$= {1{3}, 2 {{3, 4},{4,4}}, 3{1, 2, 4}, 4 {{2,2,3,4}, {2,4}}}

The analysis of the computational time of AdE structure is described as follow. At the worst case, construction of GEI for each graph takes |E| computational time. For LAEI, the number of comparison to compute local information is also |E|. Therefore, the total time complexity of AdE(G) is 2|E|. The total number of comparisons for $AdE(G_{Chem1})$ in figure is 2x6=12.

For the same DAG, we need to compute the AdE for 12800 nodes in DAG. Therefore, the worst case computational time complexity to compute AdEs for 12800 nodes is 12x12800=153600.

6.3. Analysis Results

Table 4 and 5 shows the analysis of the computational time complexity of two techniques and AdE performs extremely well compared to canonical code. It is found that the canonical code is based on the number of vertices. On the other hand, AdE based on edge-based representation. Our work explores the need for a generic representation of graphs, more specifically, chemical graphs. This is because almost all chemical graphs are non-sparse but no complete. However, our approach requires storage space for locating GEI if the graphs in DB contain significant distinct edges.

Table 4. Analysis for 50 graphs

No. of graphs in GDB=50				
Complete Graphs				
No. of vertices	No. of edges	No. of node in DAG $O(k2^n)$	No. of comparisons	
			Canonical code	AdE
5	10	1600	2,880,000	32,000
7	21	6400	17,203,200	268,800
9	28	25600	471,859,200	1,433,600
Average no. of comparison			163,980,800	578,133
Dense Graphs				
5	8	1600	2,880,000	25,600
7	18	6400	17,203,200	230,400
9	25	25600	471,859,200	1,280,000
Average no. of comparison			163,980,800	512,000
Sparse Graphs				
5	4	1600	2,880,000	12,800
7	7	6400	17,203,200	89,600
9	8	25600	471,859,200	409,600
Average no. of comparison			163,980,800	170,667

Table 5. Analysis for 100 graphs

No. of graphs in GDB=100				
Complete Graphs				
No. of vertices	No. of edges	No. of node in DAG $O(k2^n)$	No. of comparisons	
			Canonical code	AdE
5	10	3200	5,760,000	64,000
6	15	6400	96,768,000	153,600
7	21	12800	1,806,336,000	637,600
Average no. of comparison			636,288,000	285,067
Dense Graphs				
5	6	3200	5,760,000	38,400
6	12	6400	96,768,000	153,600
7	19	12800	1,806,336,000	486,400
Average no. of comparison			636,288,000	226,133
Sparse Graphs				
5	4	3200	5,760,000	25,600
6	5	6400	96,768,000	64,000
7	12	12800	1,806,336,000	307,200
Average no. of comparison			636,288,000	132,267

7. Conclusion

In this paper, we propose a proficient AdE indexing structure and also propose both algorithms for Fast-GDIndex and subgraph isomorphism queries and also propose Fast-GDIndex by using AdE structure. From analysis results, AdE indexing structure significantly reduces the computational time complexity compared to the original DGIndex.

References

[1]J. Cheng, Y. KE and N. Wilfred. “Efficient Query Processing on Graph Database”, September 2008.

[2] 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

[3] 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.

[4] Faloutsos, C., McCurley, K. S., and Tomkins, A. “Fast discovery of connection subgraphs” In KDD. 118–127,2004.

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

[6]H. He, and A. K. Singh, “Closure-Tree: An Index Structure for Graph Queries”, ICDE’06, Atlanta, Georgia, 2006

[7] 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.

[8] Koren, Y., North, S. C., and Volinsky, C. “Measuring and extracting proximity in networks”. In KDD. 245–255. 2006.

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

[10] C. Wang, W. Wang, J. Pei, Y. Zhu and B. Shi. “Scalable Mining of Large Disk-based Graph Databases”. ACM, 2004.

[11]D. W. Williams, J. Huan, W. Wang. “Graph Database Indexing Using Structured Graph Decomposition”, 2007.