I/O Efficient Algorithm for Graph Pattern Matching Problem

Pushpi Rani, Abhishek Srivastava

Abstract— This paper presents an I/O efficient algorithm for graph pattern matching problem. It is based on decision tree approach proposed by B. T. Messmer and H. Bunke. In that paper, if the time needed for preprocessing is neglected, the computational complexity of their approach is only polynomial in the number of input graph vertices. However, the decision tree is of exponential size. It's not practical for the graphs with large size.. In the new algorithm, we increases the preprocessing time as well as space complexity, it can remarkably reduces the number of I/Os and keeps the same time complexity. The algorithm is improved here to reduce its I/O complexity and to achieve a better performance on large graphs.

Keywords- Decision tree; Graph isomorphism; Subgraph isomorphism; I/O complexity; Adjacency matrix; Permutation matrix;

1 INTRODUCTION

Graph pattern matching problem is a problem to find the patterns in a large data graph [1] that match a user-given graph pattern. It is one of the most profound areas of computer science. Because of it's widely applications, it is a very active area with intensive researches for many years. It is based on graph or subgraph isomorphism. The graph isomorphism problem [2] is the problem of determining whether two graphs are isomorphic.

Graph is very expressive and effective data structure to represent the relationship among data objects, so it is widely used to represent the large volume of data. With the rapid growth of the Internet and new data analyzing techniques, there exists a huge volume of data.

When we represent these data in form of graph, the size of graph will be also very large. If, the size of graph is very large, it may be possible that whole data graph will be not fit in main memory while processing. In that case some data must be placed in secondary memory and transfer in main memory while processing. The transfer of data between main memory and secondary memory is known as I/O communication [3].

This communication between internal and external memory becomes bottleneck in case of large volume of data. Therefore, an I/O efficient algorithm is needed; this can manage the disk access as a part of algorithm. The aim of I/O efficient algorithm is to design an algorithm that minimizes
the transfer of data between internal and external memory.

The I/O algorithm also known as EM algorithm (or out-of-core algorithms) design was effectively started in the late eighties [1] by Aggarwal and Vitter. For designing I/O algorithms they proposed an important model called Parallel Disk Model (PDM) [3] in 1986. This model proposed that a good I/O algorithm should transfer data between main memory and disk in a blocked manner, and should use all of the available disks concurrently. An optimal I/O algorithm under this model minimizes the number of such blocked, parallel I/O operations it performs. So, the goal of I/O efficient algorithm is to eliminate or minimize the I/O bottleneck through better algorithm design.

- *Pushpi Rani is currently pursuing masters of technology program in computer science engineering in Jaypee University of Engineering and Technology,Guna, India, PH-7828788769*
  E-mail: pushpi.05.wit@gmail.com
- *Abhishek Srivastava is currently working in department of computer science engineering in Jaypee University of Engineering and Technology,Guna, India, PH-8103584248*
  E-mail: abhishek.srivastava@juet.ac.in

2 PROBLEM DEFINITION

Graph Pattern matching problem is also known as subgraph isomorphism problem.

There are two variations of this problem.

The first one is to detect graph/subgraph isomorphism between two unknown graphs.

In this case the problem is to find a subgraph of an input graph, called the target, such that the subgraph is isomorphic to another input graph, called the pattern.

Two Graphs G1(V1, E1, L1) and G2(V2, E2, L2) are isomorphic, if

a) There exists a bijective mapping [2] between the vertices in V1 and V2.

b) There is an edge between two vertices of one graph if and only if there is an edge between the two corresponding vertices in the other graph, and

c) The labels on the vertices and edges are preserved by the mapping.

The other one is to detect graph/subgraph isomorphism from an input graph to a database of model graphs [4]. There is often a database of graphs, so-called model graphs, and a single input graph that must be tested. It means we already have a graph database, and then test if a new input graph is graph or subgraph isomorphic to a model graph in the database or not.

3 DECISION TREE ALGORITHM

The new algorithm is based on decision tree based graph isomorphism algorithm. The basic idea of the isomorphism algorithm is that for each model graph, first computation of all

possible permutations of its adjacency matrix has been done and then the permutation matrices were transformed into a decision tree. At run time, the matrix of the input graph that is pattern graph is matched to those adjacency matrices in the decision tree which are identical to it. The permutation matrices that correspond to these adjacency matrices represent the graph or subgraph isomorphisms that we are looking for.

We will briefly describe the decision tree construction procedure. For this, first we will introduce the row-column elements [4].

A row-column element ai of a n X n matrix m is a vector

ai = (m1i, m2i , … , mii, mi(i-1) , …, mi1)

Figure 1 illustrates this representation.

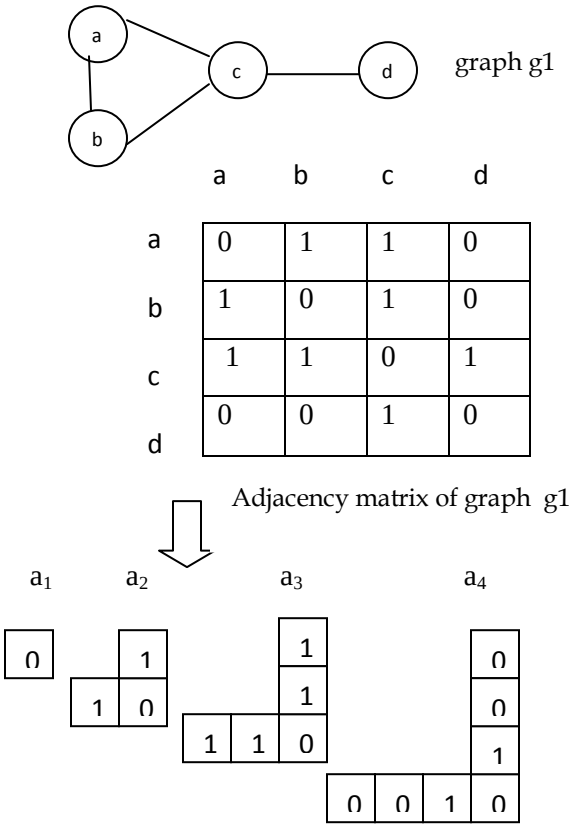


Figure 1: Row column representation of matrix

With row-column representation of matrix, we organize the model graphs into a decision tree. A decision tree sample is shown in Fig. 2 [4].

At the top of the decision tree there is a root node (dummy node). Its direct successors constitute the first level of the decision tree. On the first level, the classification is done by comparing the first row-column element of the input graph to the first row-column element ai of each permutation matrix. Likewise at nth level of the decision tree the classification is done by comparing the row-column an of permutation matrices.

The graph g1 has 4 vertices, so it has 4! =24 permutation matrices. The row-column element of these permutation matrices is used to construct the decision tree. Here, we are considering only six permutations matrix of node at which constructs only a part of decision tree. Similar procedure will be repeated for node b, c, and d also. Figure 2 illustrate the permutation matrix and decision tree for node a in the graph g1.

4 I/O EFFICIENT ALGORITHM

In the graph isomorphism and subgraph isomorphism algorithm discussed above the decision tree is of exponential size. If the vertices increase, it needs a huge amount of storage, as the size of the decision tree is directly associated with the permutation matrices of the model graphs. A graph with n

as the size of the decision tree is directly associated with the permutation matrices of the model graphs. A graph with n vertices has n! permutation matrices. To find the graph or sub-

graph isomorphism we will have to traverse the complete decision tree. Therefore, the number of I/O depends on the height of the decision tree. The maximum height of decision tree is log N. So, total number of I/O in this case will be O(logN/B) [5]. So, if we can reduce this number of I/O, the I/O complexity of this algorithm will be reduced subsequently. We use the invariant property of graph to do this job. An invariant is a property such that if a graph has it all isomorphic graphs have it.

We use the sum of adjacency matrix and number of nodes in the pattern graph to find the graph or subgraph isomorphim. The basic idea of new algorithm is, rather than traversing the whole decision tree, we will traverse up to level n, where n is the number of nodes in the pattern graph. Since, the number

of node in the pattern graph and its isomorphic graph must be same, so, there is no need to traverse the decision tree beyond that level. Since, we are using all the possible permutation of adjacency matrix; it gives us all the possible combination of nodes in the graph. Therefore, if any isomorphism present at the bottom level of tree, then it must be present at the top level also. Now, we use an array of list. This list stores the sum of adjacency matrix at each level. The array contains the first address of the list. For example, the sum of adjacency matrix at level one is stored in the list at index one of the array, the sum of adjacency matrix at level two is stored in the list at index two of the array and so on. This representation is shown in figure 3.

At run time, first we find the number of node in the pattern graph and traverse the list at that index only. For example, if the graph has five nodes, then the list at index five in array will be traversed only. While traversing, if the sum of adjacency matrix is equal to the sum of pattern graph, then we transfer only that nodes in the main memory and perform the matching operation. If both adjacency matrices are equal, then we get the graph isomorphism of the pattern graph.

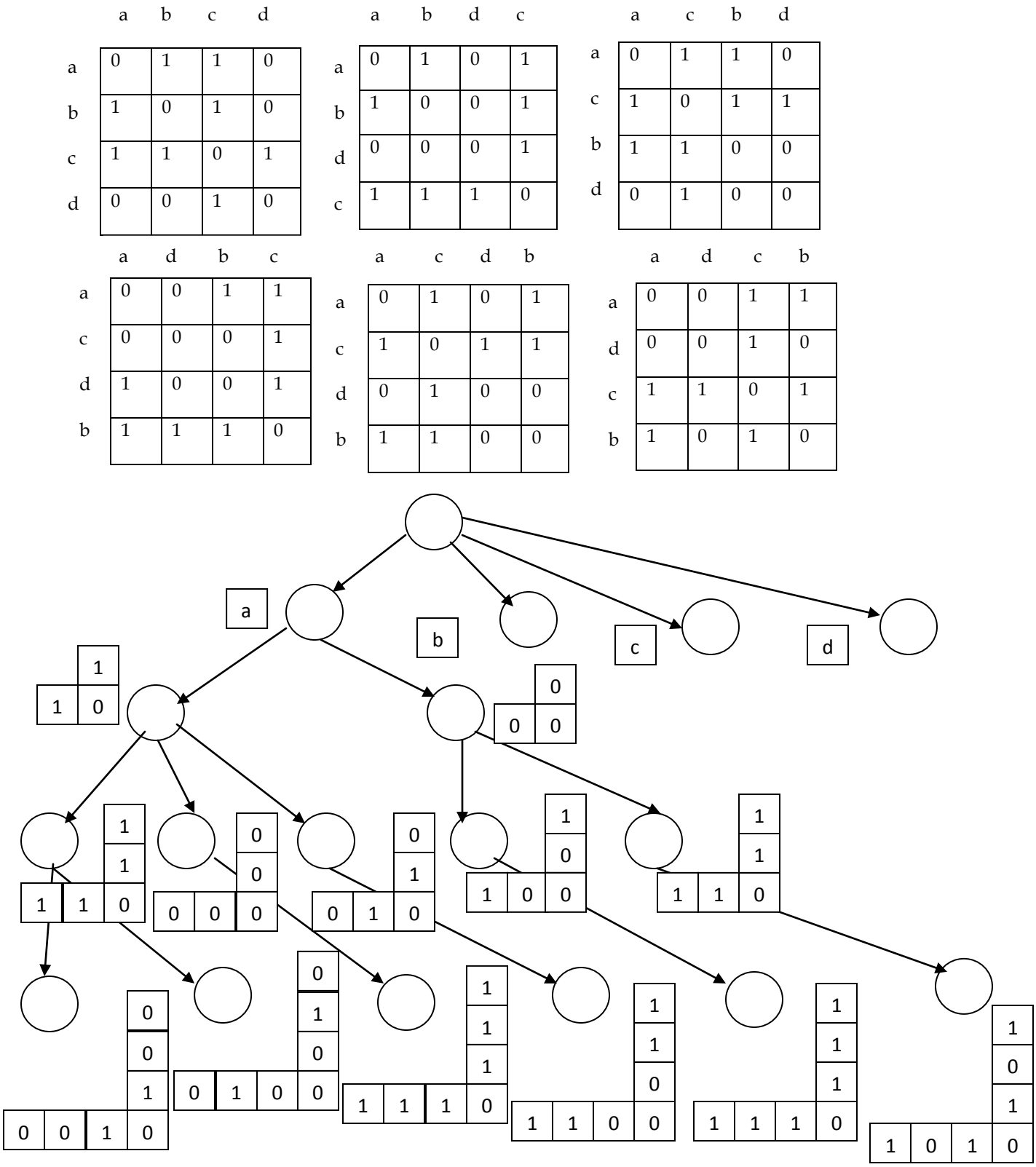


Figure 2: Decision tree construction for graph g1.

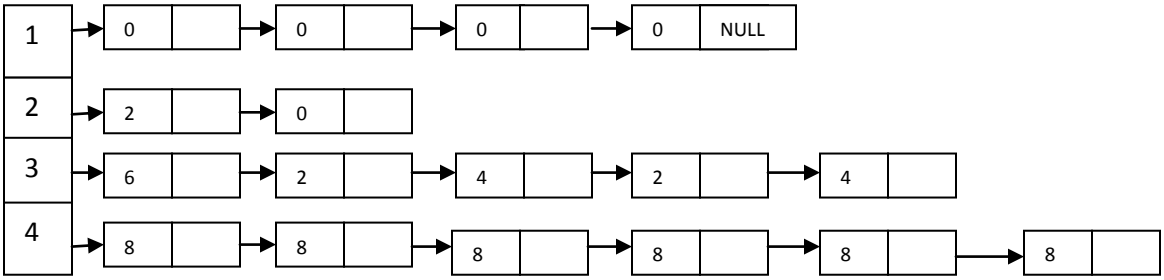
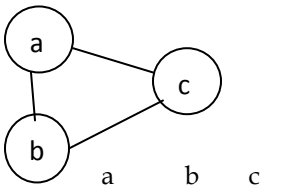


Figure 3 : Memory Representation of Array list

The row-column elements associated with predecessor row-column element and the sum of row-column element are organized in a data dictionary.

In the next section, the algorithm is explained with the help of an example. For this we consider the model graph shown in figure 1. Figure 2 represents its decision tree, which is not complete.

In example as shown in figure 4 sum of adjacency matrix is 6 and the number of nodes in the pattern graph is 3. So, we traverse the array list at index 3 only and the nodes having sum equal to 6 will be transferred into main memory to perform the matching operation.



	a	b	c
a	0	1	1
b	1	0	1
c	1	1	0

Figure 4: Pattern graph and its adjacency matrix.

The algorithm is divided into two parts.

- **Decision tree construction**: It is preprocessing step and contains two procedures: DECISION_TREE and SUM. First one constructs the decision tree and later one store the sum of decision tree in array.
- **Matching algorithm:** This is run time algorithm which finds the isomorphism of pattern graph into the database of model graph.

```
1.  DECISION_TREE(R, M, A, S)
2.  Create a dummy node.
3.  for each R
4.  for each M
5.  Rnew = R[M[i]]
6.  Go to parent of Rnew
7.  for j=2 to M[i]
8.  { count=0
9.  if(R[j]=Rnew)
10. count++
11. if(count>0)
12. no operation
13. else
14. create a new node.
15. Call procedure SUM
16. M++   //Update M
```

Figure 5: DECISION_TREE Algorithm

Here, M Adjacency matrices and R row-column element.

In detail, the procedure works as follows: line 1 create the root of decision tree, line 2 and 3 ensures that for each row- column element all permutation of adjacency matrix are traversed. If the same row column element is present in the adjacency matrix then, there is no need to add new node in decision tree (line 11), it means for every unique row-column element a new node is created in the decision tree (line 13). Line 4-13 ensures this property. Whenever a new node is created its sum is stored in the array list. For this procedure SUM is called in line 14. In line 15 we update the value of adjacency matrix which gives another row-column element. After this line 5 invoked, which assign this value into Rnew and repeat the whole procedure again until the entire row-column element has been traversed.

```
SUM(S, R, A)
1.  if (A[R[i]]=NULL)
2.  sum= S[R[i-1]]+R[i]
3.  Create a new node LIST in the linked list
4.  first=last= LIST
5.  LIST[DATA]=sum
6.  LIST[LINK}=NULL
7.  A[R[i]]=first
8.  else
9.  sum= S[R[i-1]]+R[i]
10. Create a new node LIST in the linked list
11. LIST[DATA]=sum
12. LIST[LINK}=NULL
13. last[LINK]=LIST
14. last=LIST
```

Figure 6: SUM procedure

The procedure SUM works as follows: If the first node is added into the array list, only then if condition in line 1 will be true, otherwise control goes in line 8. The line 2 and 9 update the sum by adding the previous sum and the new row column element value. Line 3 and 10 create a new node in the list and line 5 and 11 store the sum into the node. Line 6 and 12 assign the NULL value in the last node which ensures the list is ended. In line 13 the address of newly created node is assigned into the last node to create a link between nodes. Line 4 and 14 update the value of last node. Line 7 store the starting address of list into the first node.The algorithm DECISION TREE creates a decision tree of one model graph in the database. We will call this algorithm for the entire model graph in the database.

4.1 I/O Complexity of Matching Algorithm

First we are explaining the MATCH algorithm, then its I/O complexity has been analyzed Symbols used in this algorithm:
M= adjacency matrix of pattern graph.
S= Sum of adjacency matrix.
N=No of nodes in the pattern graph

In the algorithm MATCH, line 1 assign the starting address ofarray list at index N into P. line 2 ensure that, this list is traversed until it reach at the end. In line 3-5, it checks whether the sum of pattern graph is equal to the sum stored inthe list. If this condition is true, then the adjacency matrix corresponding to that node is used for matrix matching purpose. Line 5 updates the link after each node. Total no. of I/Os to transfer pattern graph into main memory will be n/B. No. of I/Os performed by while loop is P/B + n/B. By mathematical induction we can prove that the P will always be less than or equal to I * n^2, where n is the level no. of decision tree and I is the maximum number of nodes in the model graph. Hence the I/O complexity of the MATCH algorithm will be O (n/B (In)).

```
MATCH (M,S,N)
1.  P=A[N]
2.  While (P!=NULL)
3.  if(P[DATA]=S)
4.  //code for adjacency matrix matching
5.  P=P [LINK]
```

Figure 7: MATCH Algorithm

5. CONCLUSION

We have presented an I/O efficient algorithm for graph pattern matching matching algorithm, whose I/O complexity is reduced, due to the use of array list. Since, this list in created in preprocessing time, so time time complexity remains the same. The algorithm is tailored for dealing with large graphs without making particular assumptions on the nature of the graphs to be matched and can be used for both isomorphism and graph-subgraph isomorphism. The achievement seems large area of interest as many real applications such as web modeling, GIS demands new graph processing techniques to access large data graphs effectively and efficiently.

REFERENCES

[1] A.Aggarwal and J. S.Vitter (1988). The input/output complexity of sorting and related problems.Commun. ACM, 31(9):1116–1127.

[2] Ashay Dharwadker and Shariefuddin Pirzada (2008.) Graph Theory, Orient Longman and Universities Press of India,

[3] Jeffrey Scott Vitter (2008) Algorithms and Data Structures for External Memory

[4] B.T. Messmer*, H. Bunke (1999). A decision tree approach to graph and subgraph isomorphism detection, Pattern Recognition 32 1979-1998.

[5] J. S. Vitter and E. A. M. Shriver, [1994] "Algorithms for parallel memory I: Two-level memories," Algorithmica, vol. 12, no. 2–3, pp. 110–147

[6] John-Tagore Tevet (2008), Constructive Representation of Graphs: A Selection of Examples, S.E.R.R., Tallinn.