OPTIMIZED BACKTRACKING FOR SUBGRAPH ISOMORPHISM

Lixin Fu and Shruthi Chandra

Department of Computer Science, 167 Petty Building,
University of North Carolina, Greensboro, NC 27412, USA
lfu@uncg.edu

ABSTRACT

Subgraph isomorphism is a fundamental graph problem with many important applications. Given two graphs G and SG, the subgraph isomorphism problem is to determine whether G contains a subgraph that is isomorphic to SG. It is well known that the problem is NP complete in the worst case. In this paper, we present two new algorithms for subgraph isomorphism problem for labeled graphs. If the graphs have unique vertex labels, we designed a new algorithm based on modified adjacency list that has achieved linear performance. For general graphs we present another algorithm using optimized backtracking search. Though this algorithm doesn't guarantee polynomial time, it reduces the search space by applying several pruning techniques. Simulation results show that our new algorithms are competitive with classic Ullman's algorithm and more recent VF2.

KEYWORDS

Subgraph isomorphism, backtracking, algorithm, graph matching

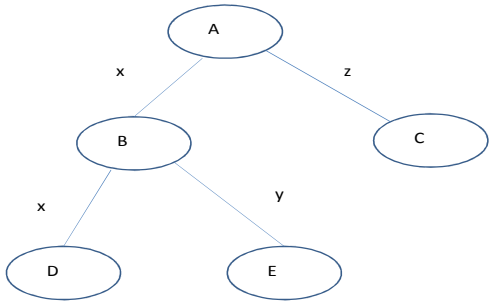
1. INTRODUCTION

A given graph G1 is isomorphic to a subgraph G2 if there exists a one-to-one mapping of the nodes of G1 onto the nodes of G2 such that all corresponding edge adjacencies are preserved. [7]. Given two graphs G and SG, the subgraph isomorphism problem is to determine whether G contains a subgraph that is isomorphic to SG. Subgraph isomorphism has wide applications in cheminformatics, bioinformatics, social networks, software and data engineering, World Wide Web, etc. In these applications, graphs are used to model complex structures and relationships. The large sizes of graphs in the real world applications pose great challenges for designing efficient isomorphism algorithms. Subgraph isomorphism is a generalization of both the maximum clique problem and the problem of testing whether a graph contains a Hamiltonian cycle, and is therefore NP-complete [9]. However with certain constraints, it is possible that subgraph isomorphism may be solved in polynomial time. For example, if the node labels are unique, we design a new isomorphism algorithm that runs in linear time.

Definition: A labeled graph G = (V, E, α , β, L), where
• V is the set of vertices or nodes,
• E is the set of edges,
• $\alpha : V \rightarrow L$, is the node labeling function;
• $\beta : V \times V \rightarrow L$ is the edge labeling function
• L is a finite alphabet of labels for nodes and edges.

DOI: 10.5121/ijdms.2012.4601

Definition: The adjacency matrix of a finite labeled graph G of *n* vertices is the n × n matrix where the non-diagonal entry *aij* represents an edge from vertex *i* to vertex *j* with label e. Let *aij* =e if there exists an edge from vertex *i* to vertex *j*, *a*ij =0 otherwise. Diagonal entries represent the node labels. We randomly assign unique Node Ids to the nodes.



	1	2	3	4	5
1	A	x	z	0	0
2	x	B	0	x	Y
3	z	0	C	0	0
4	0	x	0	D	0
5	0	y	0	0	E

Figure 1: A graph with edge labels and adjacency matrix

Definition: In adjacency list representation, for each vertex in the graph we keep a linked list of vertices adjacent to it. Depending on the requirement we can have other attributes of vertices like edge labels, Node Ids or degree along with the vertices label in the list. Following is the adjacency list representation of a simple undirected graph in Figure 1 with edge label attribute along with the vertices. Adjacency matrix representation is suitable for dense graphs while adjacency lists for sparse graphs.

2. RELATED WORK

Subgraph isomorphism problem have been researched extensively due to its fundamental significance in graph theory and recent important applications such as social media and world wide web. The algorithms roughly fall into two categories.

1. A brute force approach which directly compare two graphs using classical DFS or Backtracking approach
2. Approaches based on pre-computing before comparing the graphs

A natural way to tackle the graph isomorphism is to use direct back-tracking and brute force methods to construct a search tree. In these algorithms, the approach would be to classify the nodes according to defined invariants down the search space. Then invoke a function that explores the possible matching of the nodes of one graph against the other. Backtrack if a branch does not reach a valid solution. Examples of this class of algorithms are Ullman[1] and VF2[2].

These algorithms work very well at best and average cases, but in worst cases the whole search tree and each branch needs to be backtracked which considerably degrades the algorithm's performance.

Ullman algorithm is a classic early backtrack based algorithms. It uses a refinement procedure based on matrix of possible future matched node pairs to prune unfruitful matches. It is a simple enumeration algorithm for the isomorphisms between a graph G and a subgraph of another graph H with the adjacency matrices AG and AH. A matrix M' with |VG| rows and |VH| columns can be used to permute the rows and columns of AH to produce a further matrix P. In this algorithm backtracking procedure can only be applied to two graphs at a time. The enumeration algorithm is designed to find all of the possible mappings between a graph and a subgraph.

VF/VF2 reduces the memory requirement from $O(N^2)$ to O(N). It extends assignment by growing two connected graphs inside the pattern and the target graph respectively, and by performing the checking of conditions of connectivity of current subgraphs. It finds the first best match and then finds all the matches between two given graphs. The VF is tested with a graph dataset called as ARG database. Zong Ling proposes an algorithm that extracts a centered spanning tree from pattern graph and partition model graph in the pre-processing stage [7]. It then carries out the matching procedure by increasing the size of the pattern graphs obtained in preprocessing stage. From the experimental results shown in the paper, for most of the patterns, this algorithm runs in near-linear execution time.

3. ADJACENCY LIST MATCHING FOR UNIQUE LABELED GRAPHS

Many real world applications of SGI problem, the datasets contain unique node labels. For example, the node labels of social media graphs e.g. Facebook and those in roadmap graphs are unique. Under this constraint we introduce a new linear algorithm that matches the sorted adjacency lists using the approach similar to merging sorting. We call this algorithm MLC (Merging Lexicographic Chain).

3.1. MLC Algorithm

Throughout this paper we use ***G*** and ***SG*** to refer to our main graph (model graph) and sub graph (query graph) respectively. Input can be taken in the form of adjacency list (for sparse graph) or adjacency matrix (for dense graphs).

Step 1: Preprocessing: Construct adjacency list of nodes including all node properties
for each node $u \in SG$
    sgAdjList ← adjacency list of SG node u
for each node $v \in G$
    gAdjList ← adjacency list of G node v

Normalize the adjacency lists
for each node $u \in SG$
    sgLexChain[u] ← **Sort** the nodes in sgAdjList[u] according to the lexicographic order of their node labels of the neighbors of u
sgLexChainList ← **Sort** the list of *sgLexChain* according to the lexicographic order of their head node labels
for each node $v \in G$
    gLexChain[v] ← **Sort** the nodes in gAdjList[v] according to the lexicographic order of their node labels of the neighbors of v
gLexChainList ← **Sort** the list of *gLexChain* according to the lexicographic order of their head node labels

Step 2: Adjacency list comparison. For each node in the lexical chain list of SG, try to find a match in the lexical chain list of G. If matching is found for all nodes in SG, then search is successful otherwise no matching is found.

u : the head node of the first *sgLexChain* in *sgLexChainList*
v : the head node of the first *gLexChain* in *gLexChainList*
while (*sgLexChainList* is not empty) {
while(Label(*u*) > Label(*v*) **and** *gLexChainList* is not empty) remove first list of *gLexChainList*
// u is not found in G
if(Label(*u*) < Label(*v*) **or** *gLexChainList* is empty)) **return** "Matching is not found."

// Now Label (u) = Label (v), matching the Lex Chain of u and v
m: the first neighbor of u in *sgLexChain;*
n: the first neighbor of v in *gLexChain;*
while (*sgLexChain* is not empty) {
while(Label(*m*) > Label(*n*) **and** *gLexChain* is not empty) Remove n from *gLexChain*
if(Label(m) < Label(n) **or** *gLexChain* is empty)) **return** "Matching is not found."
if (edge_label(m) ≠ edge_label(n)) **return** "Matching is not found."
Remove m from *sgLexChain*;
Remove n from *gLexChain;*
}
Remove u from *sgLexChainList*;
Remove v from *gLexChainList;*
}
return "Matching is found."

Let's take the following example and trace it with the algorithm discussed.
Step-1: Construct normalized adjacency list for both the graphs.
Format of normalized adjacency list is: [node_id | node_label | connected_edge_label]
Here, connected_edge_label will be marked as "FIRST_EDGE" if it is a head node.
Suppose *gLexChainList:*
[1| A | FIRST_EDGE]--> | 2 | B | x |---- | 3 | C | z |---- | 4 | D | w |---- | 5 | E | u |
[2| B | FIRST_EDGE]--> | 1 | A | x | ---- | 3 | C | y | ---- | 4 | D | v | ---- | 5 | E | x |
[3| C | FIRST_EDGE]--> | 1 | A | z | ---- | 2 | B | y | ---- | 4 | D | x | ---- | 5 | E | v |
[4| D | FIRST_EDGE]--> | 1 | A | w| ---- | 2 | B | v |----| 3 | C | x |---- | 5 | E | x |
[5| E | FIRST_EDGE]--> | 1 | A | u |---- | 2 | B | x | ---- | 3 | C | v |------| 4 | D | x |
sgLexChainList:
[2| A | FIRST_EDGE]--> | 3 | B | x |---- | 1 | C | z |
[3| B | FIRST_EDGE]--> | 2 | A | x |---- | 1 | C | y |
[1| C | FIRST_EDGE]--> | 2 | A | z |---- | 3 | B | y |
Step-2: Starting from first node in the normalized adjacency list of sub graph SG, pick each list and compare with compatible lists in main graphs. If a matching is found, mark the list as visited and continue to map next node in the subgraph. If all the nodes in the subgraphs are mapped, search is successful, otherwise no match found.

We will start comparing each item in sub graph with each item in main graph
Node "A" in subgraph is compared with Node "A" in main graph.
Node labels are matched. In this case, subgraph node "A" has neighbors "B" and "C" with degree 2, and main graph node "A" has neighbors "B", "C", "D" and "E" with degree equal or greater than 2. Hence sub graph node "A" can be mapped to main graph node "A". Similarly compare and Match nodes "B" and "C" of the subgraph. So in this example a matching is found.

3.2 Advantages and Limitation

MLC algorithm works because when the node labels are unique and the chain lists of the subgraph is contained in those of the main graph, the subgraph will be uniquely matched. The following simple and powerful features make this algorithm an efficient and a scalable solution for matching unique node labeled graphs:

1. Uses the lexicographic ordering to speed up the sorting of the lists. We can use data structures such as tries to accomplish this in linear time
2. Works faster with early detection and quitting for unmatched nodes or edges. The algorithm continues to the next step only when the current node is successfully matched in the lex chain

Although MLC is linear and extremely fast, unfortunately it does not work on graphs with duplicate node labels in general. We give the following counter example, which motivated to design a new algorithm with completely different approach for general graphs in the next section.

In the *Figure 2*, we see that both Main Graph and Sub Graph have the same node label and same edge label and all the nodes have the same degree. According to our MLC algorithm, a matching should be found but this is obviously not correct.

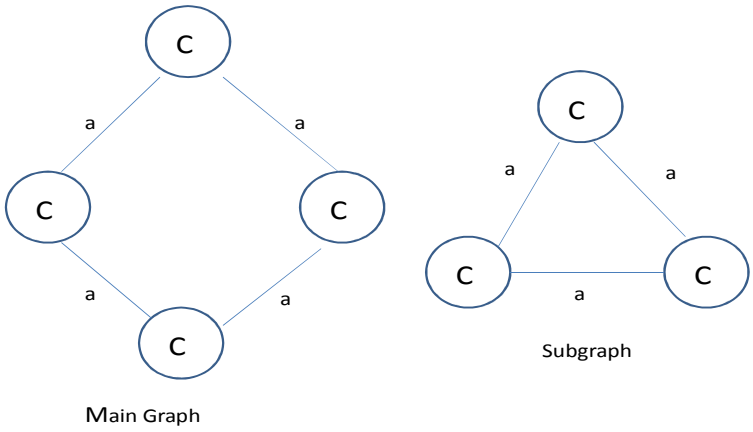


Figure 2: A counter example

4. OPTIMIZED BACKTRACKING RECURSIVE ALGORITHM

For general graph with duplicate node labels, we give a new optimized backtracking algorithm that incorporates several pruning techniques which decrease the search space of matching. We find the mapping for each node in the subgraph in the main graph using recursive comparison of neighboring nodes. The recursion goes on until we find correct mapping for all the nodes in the subgraph. The algorithm halts either if all the subgraph nodes are mapped or if the graph is exhausted without complete mapping. In the course of the algorithm we apply several pruning techniques to reduce the search paths.

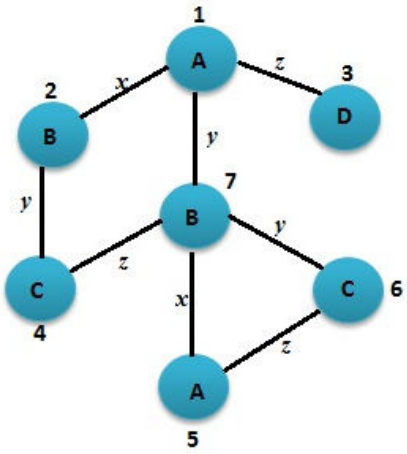
4.1 Optimized Backtracking Algorithm

Step 1: Early detection and Quit
SG_NLS ← Node Label Set of SG

```
G_NLS ← Node Label Set of G
SG_ELS ← Edge Label Set of SG
G_ELS ← Edge Label Set of SG
if (SG_NLS ⊄ G_NLS OR SG_ELS ⊄ G_ELS)
print "Subgraph not found"

Step 2: Preprocessing
for each node v ∈ SG
SG_adjList ← Construct adjacency list of nodes including all node properties;
for each node v ∈ G
G_adjList ← Construct adjacency list of nodes including all node properties;

Step 3: Recursive Search and Backtracking
CHECK-SGI() {
        u is first node in SG;
        for each node v ∈ G {
                FIND-MAPPING(u,v) ;
                if(Mapping is complete)
                        Set found=true;
                        print "Subgraph found"
        }
        if(found is not true)
                print "Subgraph Not found";
}

FIND-MAPPING(u,v) {
        if(Mapping is full)
                Set found=true;
                return true;
        if( COMPATIBLE-NODES (u, v)) {
                compatiblePairs = GET-COMPATIBLE-NEIGHBORS(u,v);
                if(compatiblePairs is null) return false;
                else {
                        Add (u,v) to Mapping;
                        for each pair(u',v') in compatiblePairs {
                                if(found is true) break; /* Search successful */
                                if(endOfBranch)
                                        reset endOfBranch ;
                                        continue;
                                FIND-MAPPING(u',v');
                        }
                        if(Mapping is full)
                                Remove last added (u,v) from Mapping
                                return false;
                        else return true;
                }
        }
}

GET-COMPATIBLE NEIGHBORS(u,v) {
        for each neighbor u in SG_adjList
                for each neighbor v in G_adjList
```

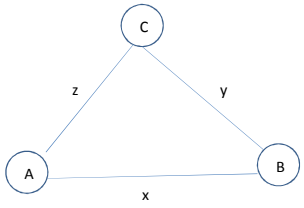
```
                    if(COMPATIBLE-NODES(u,v)) {
                        if(u OR v is not already in Mapping)
                            Add (u,v) to compatiblePairs;
                        if(∀u in SG_adjList (u,v) is in Mapping)
                            Set endOfBranch to true;
                            return compatiblePairs;
                    }
}

COMPATIBLE-NODES(u,v) {
    if((Label(u) = Label(v)) and (Degree(u) <= Degree(v)) return true;
    else return false;
}
```

We trace an example graph with the above algorithm.



	1	2	3	4	5	6	7
1	A	x	z	0	0	0	y
2	x	B	0	y	0	0	0
3	z	0	D	0	0	0	0
4	0	y	0	C	0	0	z
5	0	0	0	0	A	z	x
6	0	0	0	0	z	C	y
7	y	0	0	z	x	y	B

Figure 3: Main Graph



	1	2	3
1	C	z	y
2	z	A	x
3	y	x	B

Figure 4: Sub Graph

**********Main Graph Data**********

Pruned Maingraph Nodes

Node [id,label,degree] = [1,A,3] Neighbors: {[2,B,x][3,D,z][7,B,y]}

Node [id,label,degree] = [2,B,2] Neighbors: {[1,A,x][4,C,y]}

Node [id,label,degree] = [3,D,1] Neighbors: {[1,A,z]}
Node [id,label,degree] = [4,C,2] Neighbors: {[2,B,y][7,B,z]}
Node [id,label,degree] = [5,A,2] Neighbors: {[6,C,z][7,B,x]}
Node [id,label,degree] = [6,C,2] Neighbors: {[5,A,z][7,B,y]}
Node [id,label,degree] = [7,B,4] Neighbors: {[1,A,y][4,C,z][5,A,x][6,C,y]}

**********Sub Graph Data**********
Actual Subgraph Nodes
Node [id,label,degree] = [1,A,2] Neighbors: {[2,B,x][3,C,z]}
Node [id,label,degree] = [2,B,2] Neighbors: {[1,A,x][3,C,y]}
Node [id,label,degree] = [3,C,2] Neighbors: {[1,A,z][2,B,y]}

Initially Mapping: **[[] <--> []]**
Pass 1: Starts with (SG, G) = (1,1) => ([1,A,2], [1,A,3])
Nodes are compatible to compare. So check Neighbor compatibility
For SG node 1 neighbors are [2,B,x] [3,C,z]
For G node 1 neighbors are [2,B,x] [3,D,z] [7,B,y]
Neighbors are not compatible to compare. Pick the next node in G.
Pass 2: Starts with (SG, G) = (1,2) => ([1,A,2], [2,B,2])
Label mismatch. Nodes are NOT compatible to compare. Pick the next node in G.
Pass 3: Starts with (SG, G) = (1,4) => ([1,A,2], [4,C,2])
Label mismatch. Nodes are NOT compatible to compare. Pick the next node in G.
Pass 4: Starts with (SG, G) = (1,5) => ([1,A,2], [5,A,2])
Nodes are compatible to compare. So check Neighbor compatibility
For SG node 1 neighbors are [2,B,x] [3,C,z]
For MG node 5 neighbors are [6,C,z] [7,B,x]
Compatible Neighbor pair list (3,6) (2,7)
Mapping : [1 <--> 5]
Pass 4-Recursive Call 1: Starts with (SG, G) = (3,6)
(SG, G) = (3,6) => ([3,C,2], [6,C,2])
Nodes are compatible to compare. So check Neighbor compatibility
Compatible Neighbor pair list (2,7)
Mapping : [1 <--> 5] [3 <--> 6]
Pass 4-Recursive Call 2**:** Starts with (SG, G) = (2,7)
For SG node 2 neighbors are [1,A,x] [3,C,y]
For MG node 7 neighbors are [1,A,y] [4,C,z] [5,A,x] [6,C,y]
All SG Neighbors are already in Mapping and corresponding G nodes in Mapping matches with neighbors of 7.
Mapping : [1 <--> 5] [3 <--> 6] [2 <--> 7]
Subgraph is found in Main Graph.

4.2 Distinctive features of this algorithm

- Early Detection and Quit
  We ensure if the two graphs are compatible to compare by checking the number of nodes and edges in the subgraph and main graph at the first step. Most importantly we check whether the nodes and edges in the subgraph is the subset of that of the main graph to stop the search and quit at early stage.
- Efficient Pruning: At each stage we eliminate most of incompatible node pairs from being added to the mapping using label pruning, edge pruning, degree pruning and neighbor pruning. This avoids later backtracking.

- Optimized (short-circuited) Backtracking: As we see in the classical SGI algorithms, we see that the backtracking happens at $\mathbf{n}^{th}$ state in worst case scenario. We have a short circuit backtracking, where we check the neighbor compatibility well ahead of entering into the matching routine. In case the decision turns to be false, then we have staged break points, and we backtrack to these points immediately.
- Neighbor Compatibility: Every time a mapping pair is decided, we check whether the pair satisfies the relationship with the pairs already in mapping and whether the pair has neighbors which are compatible with each other. In this method we eliminate already compared incompatible pairs and already mapped pairs. This is a recursive call which follows with the subgraph path and ensures valid mapping reducing the search path

5. SIMULATION RESULTS

5.1 Execution Environment and Data Sets

We implement the algorithms with Java 1.6 and run the experiments on Windows system with a 2GB main memory. We have carried our testing and analysis using some of the datasets obtained from various websites in the internet. As this alone is not sufficient, we also design a random graph generator. The graph generator generates the different types of undirected graphs. This Graph generator works on the principle of pseudo random numbers. When we generate the different nodes and edge connections between nodes, a random generator procedure is invoked, which picks the labels randomly from the given inputs and generates the graph. So the simple logic behind this generator is to pick up the labels given by the user randomly and create a graph in the form of adjacency matrix.

5.2 Simulation Results

We implement our Optimized Backtracking algorithm, Ullman and VF2 for this comparison study. In our experiments we ran all the three algorithms for different kinds of graphs with different number of nodes. The experiment results show that the proposed algorithm correctly identifies whether a given subgraph exists in the main graph. That is, for all the test cases it was found that all the three algorithms gave the same result (Subgraph found/not found), with varying runtimes. Figure 5and Figure 6 shows the results of the comparison of Optimized Subgraph Isomorphism algorithm with Ullman and VF2 respectively. From these results we find that our algorithm is faster than Ullman and VF2 for some input graphs but overall has similar performance.

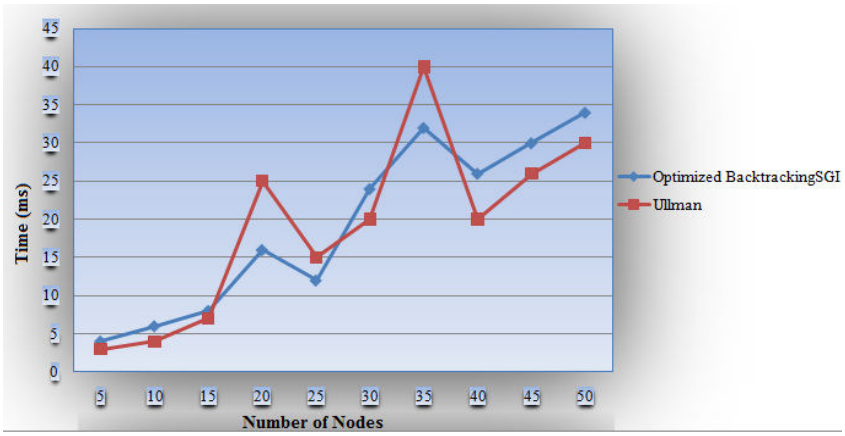


Figure 5

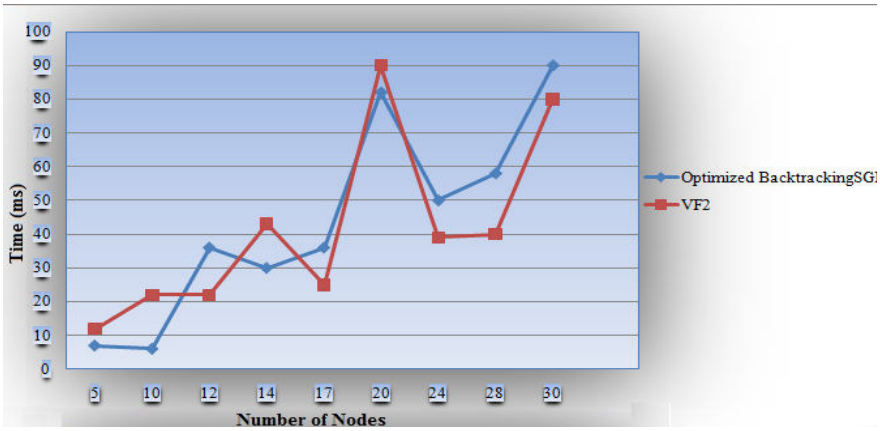


Figure 6

6. CONCLUSION AND FUTURE WORK

Our algorithm proposes an efficient backtracking mechanism, early detection of dissimilar graphs, several pruning techniques to reduce the search paths, thereby improving the performance for a large class of graphs. In particular, our new MLC algorithm achieves linear performance for graphs with unique node labels. Our optimized backtracking search reduces the search space by applying several pruning techniques. Simulation results show that our new algorithms are competitive with classic Ullman's algorithm and more recent VF2.

REFERENCES

[1] J. R. Ullmann. An algorithm for subgraph isomorphism. Journal of the ACM, 23(1):31–42,1976.

[2] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In Proceedings of the 3rd IAPR-TC-15 International Workshop on Graph based Representations, pages 149–159, Ischia, Italy, May 2001

[3] An Effective Approach for Solving Subgraph Isomorphism Problem by Zong Ling, Department of Electrical Engineering, University of Hawaii (UH)

[4] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. Performance evaluation of the VF graph matching algorithm. In Proceedings of the 10th International Conference on Image Analysis and Processing, pages 1172–1177, Venice (Italy), September 1999. IEEE Computer Society.

[5] P.Foggia, C. Sansone, and M.Vento. A performance comparison of five algorithms for graph isomorphism. In Proceedings of the 3rd IAPR-TC15 Workshop on Graph-based Representations, pages 188–199, Ischia, Italy, May 2001

[6] X.Liu and D.J.Klein. The graph isomorphism problem. Journal of Computational Chemistry, 12(10):1243–1251, 1991.

[7] An Effective Approach for Solving Subgraph Isomorphism Problem by Zong Ling Department of Electrical Engineering, University of Hawaii (UH)

[8] Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism by Haichuan Shang Ying Zhang Xuemin Lin and Jeffrey Xu Yu

[9] M.R.Garey and D.S.Johnson. Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., 1990.

[10] Subgraph Isomorphism in Polynomial Time, B.T Messmer and H.Bunke, University of Bern, Neubruckstr. 10, Bern, Switzerland

[11]Fast Algorithm for Graph Isomorphism Testing, Jos´e Luis L´opez-Presa1 and Antonio Fern´andez Anta2, Universidad Polit´ecnica de Madrid, 28031 Madrid, Spain, Universidad Rey Juan Carlos, 28933 M´ostoles, Spain.