Algorithm to Check whether a Given Graph is Subgraph of Other Graph

Abhinav Mishra
Indian Institute of
Information Technology
iim2014003@iiita.ac.in

Neelanjana Jaiswal
Indian Institute of
Information Technology
ism2014005@iiita.ac.in

Sannihith Kavala
Indian Institute of
Information Technology
ihm2014004@iiita.ac.in

Abstract—Finding subgraph isomorphisms is an important problem in many applications which deal with data modeled as graphs. While this problem is NP-hard, in recent years, many algorithms have been proposed to solve it in a reasonable time for real datasets using different join orders, pruning rules, and auxiliary neighborhood information. However, since they have not been empirically compared one another in most research work, it is not clear whether the later work outperforms the earlier work. Another problem is that reported comparisons were often done using the original authors' binaries which were written in different programming environments. In this paper, we address these serious problems by re-implementing five state-of-the-art subgraph isomorphism algorithms in a common code base and by comparing them using many real-world datasets and their query loads. Through our in-depth analysis of experimental results, we report surprising empirical findings.

Index Terms—Subgraph, Isomorphism, NP-complete.

I. Introduction

Many complex objects, such as chemical compounds, social networks, and biological structures are modeled as graphs. Many real applications in bioinformatics, chemistry, and software engineering require efficient and effective management of graph structured data. One of most important graph queries in graph databases is the subgraph isomorphism query. That is, given a query q and a data graph g, find all embeddings of q in g. This problem belongs to NP-hard [10] and has many important applications, such as searching chemical compound databases,querying biological pathways, and finding protein complexes in protein interaction networks. Ullmann proposes the first practical algorithm for subgraph isomorphism search for graphs. It is a backtracking algorithm which finds solutions by incrementing partial solutions or abandoning them when it determines they cannot be completed. In recent years, many algorithms such as VF2, QuickSI have been proposed to enhance the Ullmann algorithm. These algorithms exploit different join orders, pruning rules, and auxiliary information to prune out falsepositive candidates as early as possible, thereby increasing performance.

II. Motivation

The Subgraph Isomorphism algorithm is used extensively in various field and is a ongoing research problem in the field of Graph Theory. Few of its application which motivate us to pursue this problem are as follows -

1) As subgraph isomorphism has been applied in the area of cheminformatics to find similarities between chemical compounds from their structural formula; often in this area the term substructure search is used. A query structure is often defined graphically using a structure editor program; SMILES based database systems typically define queries using SMARTS, a SMILES extension.
2) The closely related problem of counting the number of isomorphic copies of a graph H in a larger graph G has been applied to pattern discovery in databases, the bioinformatics of protein-protein interaction networks, and in exponential random graph methods for mathematically modeling social networks.
3) Ohlrich et al. (1993) describe an application of subgraph isomorphism in the computer-aided design of electronic circuits. Subgraph matching is also a substep in graph rewriting (the most runtime-intensive), and thus offered by graph rewrite tools.
4) The problem is also of interest in artificial intelligence, where it is considered part of an array of pattern matching in graphs problems; an extension of subgraph isomorphism known as graph mining is also of interest in that area.

III. Algorithm

The subgraph isomorphism problem asks whether a graph G has a subgraph $G' \subset G$ that is isomorphmic to a graph P. So basically you have the picture on the box of a puzzle (G) and want to know where a particular piece (P) fits, if at all. It is NP-complete because Hamiltonian cycle is a special case.

A. Algorithm I

Given two graphs.Let the first graph be graph1 and second graph be graph2.The objective is to find if the second graph is the subgraph of the first. Consider the graph2.Take every node in graph2 and assign some unique label to every node in the range(1,graph2.size). Now for every edge between these labelled nodes,create a hash indicating if the edge between labelled nodes in present or not.Now consider the graph1 and generate all possible subgraphs from graph1.Now for every generated unlabelled subgraph try generating all possible labellings for that subgraph.for every combination check if every labelled edge in that combination is equivalent to labelled edge in graph2(i.e., check the hash). If any combination is found to satisfy this criteria,Then given graph2 is subgraph of graph1.

```
ans=false;

void main():

  scan number of nodes and
  no. of edges of reference_graph;//nodes=n1,edges=e1;

  scan the edges of referene graph;

  scan the number of nodes and
  number of edges of subgraph;  //nodes=n2,edges=e2;

  scan the edges of subgraph;

  temp_graph=NULL;
  generate_hash(subgraph);
  generateSubgraph(reference_graph,e1,e2,temp_graph,0);
  if(ans==true):
     print "YES";
  else:
     print "NO";

  return ;

void generaSubgraph(ref_graph,e1,e2,temp_graph,num):

  edge_num=temp_graph.number_of_edges;

  if(edge_num == e2 || num >= e1):

    available=[1...edge_num];
    func(available,temp_graph,0);

  else:

    generate_subgraph(ref_graph,e1,e2,temp_graph,num+1);
    add ref_graph.edge[num] in temp_graph;
    generate_subgraph(ref_graph,e1,e2,temp_graph,num+1);

void func(available,labelled_graph,num):

    if(available.size()==0):

        new_edges=generateEdges(labelled_graph);

        for every edge in new_edges:
            if(hash[edge] is absent):
                return ;

        ans=true;

    for i in range available:
        labelled_graph.vertex[num] = i;
        remove i from available;
        func(availble,labelled_graph,num+1);
        add i to available;

void generate_hash(graph subgraph):
    for every edge in subgraph:
        hash[edge]=present;
```

Time complexity:

The worst case time complexity of this method is pow(2,E)*factorial(V); where E is the number of edges and V is the number of vertices

B. Algorithm II

It is possible to encode a subgraph isomorphism as a $|V_P| \times |V_G|$ matrix M in which each row contains exactly one 1 and each column contains at most one 1. We set m_{ij} to 1 iff $v_j \in G$ corresponds to $v_i \in P$ in the isomorphism. Then $P = M(MG)^T$, where I use P and G to stand for the adjacency matrices. If we aren't looking for induced subgraphs, $P \leq M(MG)^T$ (componentwise), i.e. the subgraph of G selected by M might contain additional edges not present in P.The algorithm works by systematically enumerating possible matrices M and checking whether they actually encode an isomorphism.

We start by setting up a $|V_P| \times |V_G|$ matrix M^0 that contains a 1 at (i,j) if it is possible that $v_i \sim v_j$ in some subgraph isomorphism. For now, we only use the degree as a criterion, i.e. we can map v_i to v_j if the latter has enough neighbors: $m^0_{ij} = 1 \Leftrightarrow \deg(v_i) \leq \deg(v_j)$. If we were cleverer we could remove more 1's and reduce our runtime.

Now all we need to do is try all matrices M that can be obtained from M^0 by removing all but one 1 from each row while having at most one 1 in each column. We do that recursively.

```
recurse(used_columns, cur_row, G, P, M)
    if cur_row = num_rows(M)
        if  M is an isomorphism:
            output yes and end algorithm

    M' = M
    prune(M')

    for all unused columns c
        set column c in M' to 1 and-
        -other columns to 0
        mark c as used
        recurse(used_column,cur_row+1,G-
        -,P,M')
        mark c as unused

    output no
```

Making the M' copy of M isn't really necessary, but it will become so once we implement our pruning procedure.

C. Pruning

We want to (safely) change at least some of the 1's in our matrix to 0's to reduce the computation time. For that, we use a simple observation. If some $p \in V_P$ has neighbours $p_1, \ldots, p_l \in P$, and we map it to some $g \in V_G$, then we'd better also map $p_1, \ldots, p_l$ to neigbors of g.

Remember that a 1 at (i,j) in M means that we still think that $v_i \in P$ can correspond to $v_j \in G$. But if we already found out that there is a neighbour of $v_i \in P$ can't be mapped to any neigbour of $v_j \in G$ clearly the 1 at (i,j) is wrong and we can change it safely to a 0. This change might make more mappings impossible, so we iterate this check until nothing can be changed. If we remove all 1's from a row during this refinement, we can stop the whole process, since M can't be completed to an isomorphism anymore.

```
do
for all (i,j) where M is 1
    for all neighbors x of vi in P
        if there is no neighbor y of vj-
        -s.t. M(x,y)=1:
            M(i,j)=0
while M was changed
```

Now the effectiveness of our pruning procedure depends on the order of the vertices. The earlier in the recursion we find a 1 that can be changed to a 0, the better. Thus it is a good a idea to order the vertices such that high degree vertices of P are first.

D. Clever Implementation

This algorithm seems rather costly, since we do a matrix multplication for every leaf in the recursion tree and manipulate the matrix quite a lot in between. However, note that we're dealing with boolean matrices here. It's a good idea to encode them as bit vectors. Then multiplication can be done efficiently using bit-twiddling, even if we still use the naive $O(n^3)$ algorithm. Similarly setting a column to 1 and the other columns to 0 can be done using bit-twiddling and finding viable neighbors during the pruning step is fast too. Using these implementation tricks we can speed up the naive algorithm by some largish constant factor, depending on the word size of the CPU and whether it supports SSE or similar vector operations.

IV. Conclusion

We have given two approaches for checking whether one graph is subgraph of another or not.The first approach given here is the naive solution where we check all the permutations and in the second one we try to prune the permutations so as to reduce the number of permutations to be checked.So the second approach is better than the first one.