Sub-graph Isomorphism in GPU

A Project Report

submitted by

GEORGE JOSEPH

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

May 2017

THESIS CERTIFICATE

This is to certify that the thesis entitled **Sub-graph Isomorphism in GPU**, submitted by **George Joseph**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bonafide record of the research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Rupesh Nasre
Research Guide
Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date:

ACKNOWLEDGEMENTS

I would like to to express my sincere thanks and deep sense of indebtedness to my guide Dr. Rupesh Nasre for his guidance and motivation throughout my work. His inspiring suggestions motivated me to solve problems efficiently. I am also grateful to my guide also for providing access to the Libra servers without which none of my experiments could have been performed.

I would also thank Vinod Raju, Jithin K M, Nikhil Stephen,Hasit Bhatt,Vivek V P and all my colleagues who always helped me whenever needed in my research.

Lastly, I am thankful to god for giving me enough luck to get into IIT Madras and I am thankful to my parents for all the moral support and the amazing oppurtunities they have given me over the years.

ABSTRACT

In Sub-Graph Isomorphism we are trying to find whether a subgraph of Data Graph is isomorphic to the query graph. Sub-Graph Isomorphism is a NP-Hard problem. Because of the lots of applications of the problem in many Data mining and pattern matching, the problem is well studied and different methods are proposed in the past years. We studied many of the state-of-art techniques used to solve the problem. Then parallelised one of them in GPU. We then tried to solve the dynamic version of the problem. In the dynamic version we remove or add edges on the go and we would like to get the output on the current problem.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

NEC	Neighbourhood Equivalence Class
CVS	Candidate vertex Set
NDS	Neighbouring Discriminating Substructure
BFS	Breadth First Search

CHAPTER 1

INTRODUCTION

1.1 Overview

Consider that you have pattern and you want to know where all the pattern coming in the data. The pattern can be easily represented as a graph. The actual data can be image video, network of people, etc. The graphs can be used to represent any form of data. The pattern matching has a wide range of application. It is not an easy problem. Since it need a lots of checking at each node. It is similar to placing the pattern at each node in the graph and trying to rotate,flip, etc to find a similar representation. There are lots of possibilities that can match but there may be only small number of actual matching. We need to find them.

1.2 Motivation

As I mentioned before this problem is having lots of applications since the current era is trying to extract lots of features from images, videos,audios,etc using many pattern matching techniques. Since the problem is NP Hard researchers have focussed on efficiently solving the problem in practice. The numerous cores of GPUs may help us to solve this problem faster. Each node search is independent so they can be done in parallel. This is the primary motivation on trying to do the sub-graph isomorphism in GPUs.

1.3 Major Contribution

- Implemented a state-of-Art solution in GPU
- Dynamic SubGraph Queries are handled in parallel

1.4 Organization of Thesis

2 discuss the problem and the state-of-art algorithms. A detailed comparison of various algorithms in terms of different pruning techniques is performed. 3 discuss the parallel version of the TurboIso algorithm. 4 discuss the implementation of the dynamic version of the problem. The challenges involved in its implementation are discussed in depth.

CHAPTER 2

Sub-Graph Isomorphism

2.1 Problem

Input: A data Graph D, and Query Graph Q. The graphs D and Q are undirected with nodes and edges having label.See Figure 1.

Graphs given as adjacency list.

Output: Give all the matching mapping of each node in Q to node in D

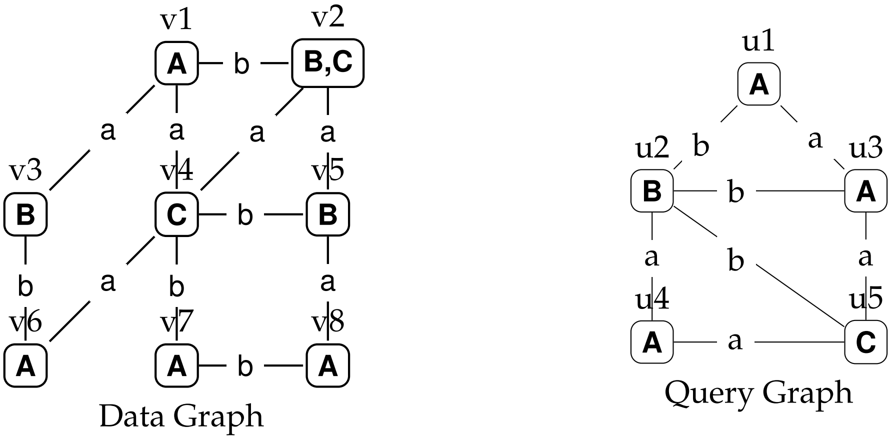


Figure 2.1 Data and Query Graph

2.2 Related Work

2.2.1 Generic Algorithm

The generic algorithmJinsoo Lee [2012] for subgraph isomorphism will help us to study the aspects of state-of-art algorithms in deep. It is presented in Algorithm 1.

Algorithm 1 Subgraph Search

```
Input: Data Graph D,Query Graph Q.
Output: Mapping of vertices from Q to D.
1. for each vertex v of Q
    (a) C(v)=FindCandidates(v,D)
    (b) If C(v) is empty return
2. SUBGRAPHMATCHING(C,Q,D,$\phi$)
Procedure SUBGRAPHMATCHING:
Input: Candidates C,Data Graph D,Query Graph Q Current Map M.
Output: Mapping of vertices from Q to D.
1. if $|M| = |V(q)|$ report M
2. else
    (a) u=NextVertex()
    (b) $C_r$ =RefinedSet(M,u,C(u))
    (c) for each $v \in C_r$
        i. if IsJoinable(M,v)
            A. UpdateState(M,v)
            B. SUBGRAPHMATCHING(q,d,C,M)
            C. RestoreState(M,v)
```

The procedure *FindCandidates* finds the vertices in datagraph which can be mapped to query vertex. The procedure *NextVertex* finds the next vertex in querygraph which should be tried to be mapped.

The $\mathsf{RefinedSet}$ prunes out some nodes in the candidate set. The $\mathsf{IsJoinable}$ checks whether the map is right. The $\mathsf{UpdateState}$ moves to next state (adds new vertex to map).The $\mathsf{RestoreState}$ removes the vertex from map and thus restores the state.

If you consider the graphs in Figure 1. $C(u1) = \{v1, v6, v7, v8\}$ pruned by node label and degree. Similiarly $C(u4) = \{v1, v6, v7, v8\}$ and $C(u2) = \{v3, v2, v5\}$. Procedure $\mathsf{NextGraph}$ will give the vertices on query graph in some order like $\{u1, u2, u3, u4, u5\}$. It can be even $\{u1, u3, u4, u2, u5\}$. Once $u1$ is mapped to $v1$,the procedure $\mathsf{RefinedSet}$ will remove $v1$ from $C(u4)$. If Map has these values $\{(u1, v1)\}$,$\mathsf{NextGraph}$ returned $u2$, $\mathsf{RefinedSet}$ returned $\{v3, v2, v5\}$ and current v is $v5$ then procedure $\mathsf{IsJoinable}$ check whether there is an edge between $v1$ and $v5$ like the one between $u1$ and $u2$.

2.2.2 Ullmann Algorithm

This algorithmUllmann [1976] is simple.The $\mathsf{FindCandidates}$ finds same degree nodes. The $\mathsf{NextVertex}$ takes the next node in input.The $\mathsf{RefinedSet}$ removes nodes already mapped. The $\mathsf{IsJoinable}$ iterate over the neighborhood and checks corresponding edge exists. The $\mathsf{UpdateState}$ and $\mathsf{RestoreState}$ adds and removes the vertex from map respectively.

2.2.3 VF2 Algorithm

VF2 algorithm was proposed in L P Cordella [2004].The $\mathsf{NextVertex}$ takes the next connected vertex. The $\mathsf{RefinedSet}$ uses these rules

1. Prune out v if not connected from already mapped vertices.
2. The count of unmatched vertices of neighbors of v in Q must be greater than unmatched vertices of neighbors of u in D
3. The count of unmatched vertices of neighbors of v in Q must be greater than unmatched vertices of neighbors of u in D

2.2.4 QucikSi Algorithm

QuickSi algorithm was proposed in H SHang [2008].The $NextVertex$ takes vertices in the most infrequent vertex first order.The $RefinedSet$ uses connectivity to mapped vertices to prune.The $RefinedSet$ only iterates over mapped adjacent vertices.

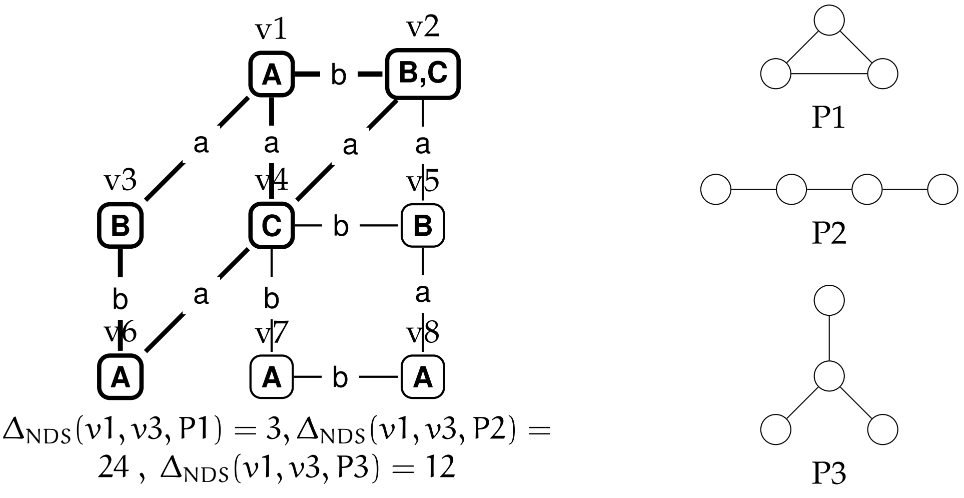
2.2.5 GADDI Algorithm

GADDI was proposed in S Zhang [2009]. They use the neighboring discriminating substructure(NDS).$\Delta_{NDS}(u, v, P)$ is the number of occurrences of P in induced subgraph $N_k(v) \cap N_k(u)$. $N_k(u)$ is the graph having all the edges in k distance from u.A matrix L is created such that each row corresponds to an induced graph g and each column represent each pattern.See the NDS calculation in Figure 2.The dark lines represent the $N_k(v1) \cap N_k(v3)$ with k=2.

The $NextVertex$ takes the one next in the DFS Tree from the vertex. The $RefinedSet$ prune based on these conditions.

If for each $u^{'} \in N_k(u)$ there is no data vertex $v^{'} \in N_k(v)$ having

1. $L(u^{'}) \subseteq L(v^{'})$
2. The shortest distance between $v^{'}$ and v must be greater than or equal to distance between u and $u^{'}$.



$\Delta_{\text{NDS}}(v1, v3, P1) = 3, \Delta_{\text{NDS}}(v1, v3, P2) = 24$, $\Delta_{\text{NDS}}(v1, v3, P3) = 12$

Figure 2.2 GADDI NDS Calculation

2.2.6 GraphQL Algorithm

The GraphWl was proposed in H He [2008]. They use neighborhood signatures. The neighbor of the vertex is encoded as the collection of labels of its neighbors. The $RefinedSet$ pruning is based on this signature. This is a one hop signature. For example the $sig(u1) = \{B, A\}$ in Figure 1.

2.2.7 SPath Algorithm

The SPath Algorithm was proposed in P Zhao [2010]. They use signatures till k hop. They store the signature in the form (d, l, c) where d is distance to the neighbor, l label,c count.The $RefinedSet$ pruning is based on these signatures.For example the $sig(u1, 2) = \{(1, B, 1), (1, A, 1), (2, A, 1), (2, C, 1)\}$ in Figure 1.

2.2.8 STWig Algorithm

The STWig Algorithm was proposed in Xiaojie Lin [2012]. Here the query graph is divided into smaller graphs. These smaller graphs are searched in the data graph first. Their results are combined to get the final result. The graphs are divided such that the root of g_j must be of the children of any of the graphs g_i such that $i < j$.All STWigs are two level trees.See Figure 3 for the STWigs generated for query graph in Figure 1.

The candidates can be started from the least frequent pattern and then building up.The splitting of the graphs,matching the small STWigs and then combining can be done in GPU. But the combining of STWig results in the troublesome task. This can lead to the need of large amount of memory too since the candidate set can increase exponentialy.

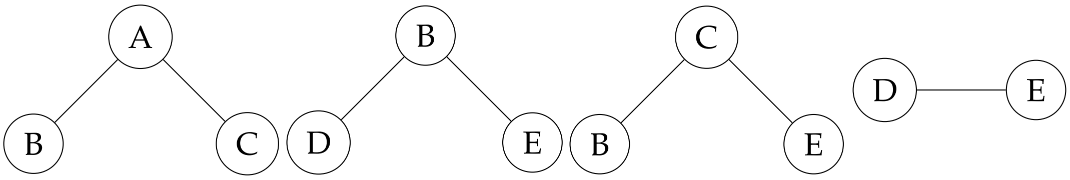


Figure 2.3 STWig Decomposition

2.3 Implemented Algorithm

2.3.1 TurboIso

It was proposed in Wook-Shin Han [2013].$\mathtt{Turbo}_{\mathrm{iso}}$ uses neighborhood equivalence class(NEC). Here they make a tree out of the query graph. In this tree they create the NEC. Each node will be part of a unique NEC. Later this tree is searched in the data graph. Then the graph edges are checked.

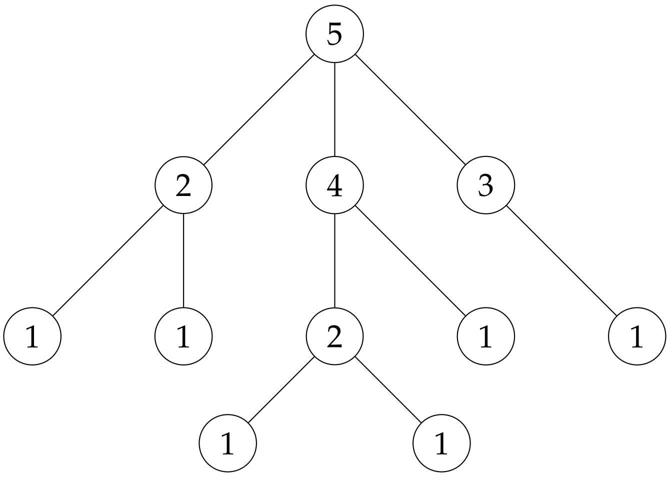


Figure 2.4 NEC Numbering

Algorithm 2 NEC creation

```
Input: Data Graph D,Query Graph Q.
Output: Mapping of vertices from Q to D.
  1. The leaf nodes are given NEC 1
  2. for each level upward
     (a) Each new neighborhood will get a new NEC
```

Then CVS for each NEC is found in the data graph. Then for each combination the actual graph is tested for a match.

2.3.2 Inferences

All of the above algorithms tried to decrease the total candidate vertices set(CVS) for a vertex in query graph. The permutation and combination of these vertices will result in the final answer. More the CVS more will be the combinations. Since the answer requires all possible permutations we can't avoid this calculation. So we need to prune out the false candidate as early as possible. This is the reason why intermediate pruning steps are added in `UpdateState` also. The neighborhood signature is the way seen so far to prune the CVS initialy better.

CHAPTER 3

Prallel Implementation

3.1 Introduction

In the parallel implementation the unique NEC numbering, CVS generation in data graph, and checking whether the query graph exists in the subgraph are done in GPU. The combination generation is done in CPU.

3.2 Algorithm

[H] **Algorithm 3** Parallel $\mathtt{Turbo}_{\mathtt{iso}}$

```
Procedure: NECGen()
Parallel NEC generation on each node.
Input: Query Graph Q.
Output: NEC.
1. repeat until all nodes got NEC
2. Run parallel on all nodes
    (a) running on node v
    (b) iterate over all neighbors of vertex v
    (c) if not all neighbors have NEC return
    (d) find the hash of neighborhood.set it hash location to 1.
3. assign unique numbering to all 1′s in the hash array
4. Run parallel on all nodes
    (a) running on node v
    (b) iterate over all neighbors of vertex v
    (c) if not all neighbors have NEC return
    (d) find the hash of neighborhood.Find the unique NEC in hash location
    (e) assign it to the vertex
```

Procedure: CVSGen()

Parallel CVS generation for each NEC.

Input: Data Graph Q,NEC.

Output: CVS.

```
1. all nodes in data graph is in NEC 1
2. for each NEC from 2 to last
3. Run parallel on all nodes
    (a) running on node v
    (b) iterate over all neighbors of vertex v.
    (c) check the existence of the neighbourhood of NEC on the node v.
    (d) if found set the flag 1.
```

Procedure: PermandComb()

Parallel check all possible permutations and combinations.

Input: Data Graph D, Query Graph Q, Current vertex index(i),Possibilities(P),MaximumPossibilit

Output: Mapping of nodes from query graph to data graph.

```
1. if i= $|V(q)|$
2. Report all values in P and return
3. else
    (a) find u,the NEC of the vertex
    (b) Multiply P with CVSGen(u)
    (c) while $P >= MP$
          i. call CheckMap() on MP elements of P
         ii. call Exclusive-scan on Checkmap output
        iii. Save valid possibilies to newP
         iv. P-=MP (numbers)
    (d) move newP to P.
```

Procedure: CheckMap()

Parallel check of existence of query graph.

```
**Input**: Data Graph D,Map m,Query Graph Q,till vertex $v$ in query graph.
**Output**: true/false.
  1. running on all nodes u if $u <= v$.
  2. iterate over all neighbors of vertex u.
  3. check the existence of all edges in data graph corresponding to one in query graph.
  4. if not all edges present set false.
```

The procedure $NECGen$ gives unique ids to one tree in the query graph similar to Figure 4. We do a level order traversal on the tree. So at some nodes all its children may not have NEC given we process those nodes in the next iterations. The step 2 finds the neighbourhoods that can be processed at the current iteration and finds a hash of the neighbourhood. To these unique hashes we assign numbering by scan algorithm. Then these numberings are assigned back to each nodes.

The procedure $CVSGen$ finds the candidates for a particular NEC. They check on each node on datagraph and checks the neighbouring nodes for matching negibours of the particular NEC in querygraph.

The procedure $PermandComb$ finds all possibilites of the query graph. For eachvertex of query graph it finds all possibilites(current possibilities x CvsGen(u)). If the possibilies is more than we can store(MP), $CheckMap$ is called on all current possibilities and the wrong ones are removed. This will make the P ¡MP. At last when i= $|V(q)|$, P has all valid possibilites. The procedure $CheckMap$ checks the existence on non tree edges are actually present in the current mapping. If $CheckMap$ retruns true it is reported as a correct mapping.

3.3 Inferences

The parallel version needs to store the mapped NEC for each node in the data graph. This is asking for a space of $O(n * N(q))$ where n is number of nodes in datagraph and $N(q)$ is number of NEC in query graph. The time taken for executing complete graphs on various test scenarios are given in table below. These results are obtained on a NVIDIA CUDA supported GPU with 580 MHz speed. The number of nodes in query graph and data graph are given in the 1st column and 1st row.(*) Random Graphs.

Data Query	10	100
3	2.8	3
5	3.6	3*
10	4.8	60*

Table 3.1 Execution Time(in sec)

3.4 Failed Approach

We tried to make the the NEC numbering more informative by using primes and composites. A prime will be assigned to a class if that graph has no other embeddings of any previous graphs we came across. If it has the embeddings we give product of the prime numbers of the embeddings. This method helps to know that if the NEC has a composite number it has some smaller graphs embedded in it. So we won't be needed to search the subgraphs in this node.

It actually captures all subgraphs at the root. See the figure below.

$v4$ is getting 2 since it has only one child 1. $v7$ and $v3$ gets 4 since it has two same subgraphs(subgraph 2) inside them. $v2$ is getting 10 since it has subgraph 2

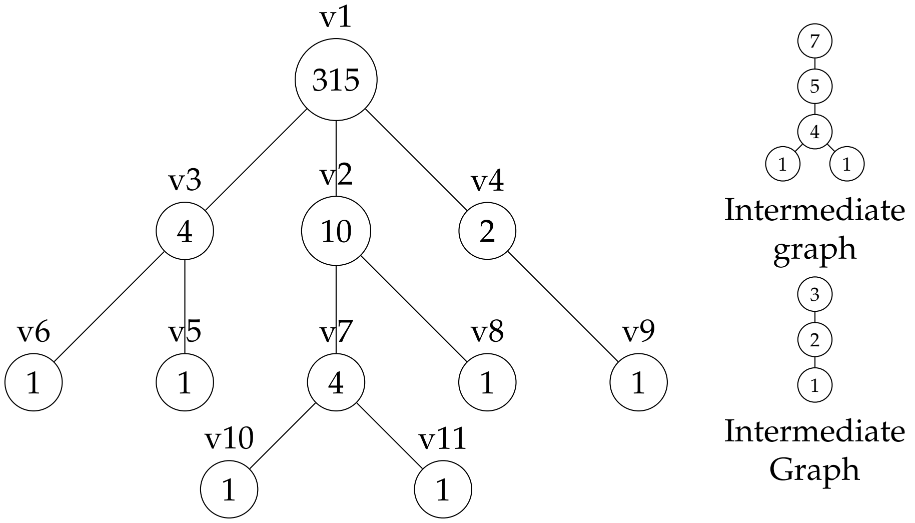


Figure 3.1 NEC based on primes

and 5 (see the numbering shown on right). Similiarly $v1$ has two 3s($v2$ contributes one 3) ,one 7 and one 5.

But this was not effective.When we consider the data graph we will need to store only one integer the product of all the graphs inside that node. The first problem we faced was the value of composite number can go beyond the long integer limit. So we tried only storing only primes. But that also didn't make much difference. By our propagation algorithm in Data Graph, we start by giving id 1 to all nodes in the data graph in the first iteration. In the second iteration every node will get id 2 since every node will have a child of id 1. Then in third every node will get 3 and so on. So every node gets every id present in query graph.

It only helped as in knowing whether there exist a path of length matching the largest length path in query graph. This will lead to all nodes becoming a candidate for the final search we atleast one graph existed in the connected component. So it is not making the CVS tight.

CHAPTER 4

Dynamic Operations

4.1 Introduction

The dynamic operations are add/delete operation on either query or data graph. The dynamic version is also having large applications. The dynamic processing will help to generate the isomorphic mappings without computing the whole answer again. This will give a great improvement in time. The dynamic changes are allowed with one condition that the query or graph will remain connected at any point of time. The dynamic queries are processed non-deterministically meaning the order of execution of the dynamic queries is undefined. The dynamic operations and its difficulties are shown below.

	Query Add Edge	**Query Remove edge**	**Query Unchanged**
Data Add Edge	**Easy**	**Difficult**	**Easy**
Data Remove Edge	**Easy**	**Difficult**	**Easy**
Data Unchanged	**Easy**	**Difficult**	**Static**

Table 4.1 Dynamic Changes Difficulty Level

4.2 Intermediate Answers

The intermediate answers will be saved so that dynamic answers can be processed faster. A,B,C,D are the intermediate answer saved variables. A store the

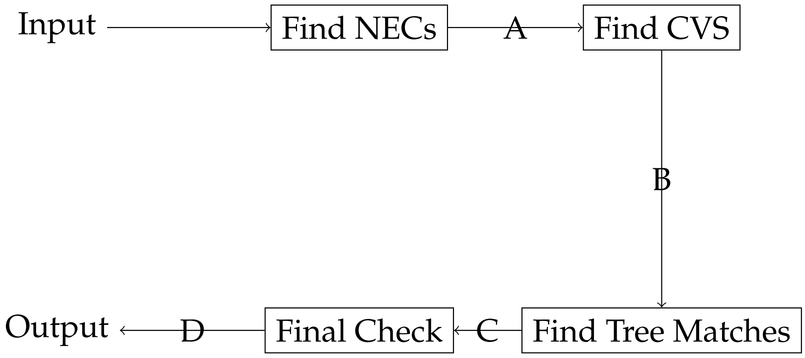


Figure 4.1 Intermediate Answers

NECs of each vertec in query graph. B store CVS of each query vertex in data graph. C store the possible tree matches. D store the final exact graph maps.

4.3 Adding Edge to Query Graph

It one of the easiest case because we need to check all the previous cases. The final answer will be a subset of previous answer.ie; Some of the matching will become invalid since the added edge may not be present. Only D changes. Multiple queries can be done in parallel by checking the existence of the added edges in all the previous answers. In parallel on all previous answers check the presence of added edges.

4.4 Deleting Edge from Data Graph

It is also easy because the final answer is subset of previous answer.Multiple Queries can be processed similiar to the previous case.Only D changes. Even if we needed to

4.5 Deleting Edge from Query Graph

This is a difficult case since we need to find the mappings that are going to be added. This case is divided into two parts.

4.5.1 Deleting a non-tree edge

The tree inside the query graph remains unchanged. So the tree matches are correct. We need to go through all the tree matching(C) and check for possible additions of maps to final answer. So saving the intermediate answer C helps in finding the solution faster.

4.5.2 Deleting a tree edge

Since the tree is changed here the CVS of vertices may change.So rather than calculating all the CVS there is a more efficient way. If a edge u-v in the tree is deleted and u is parent of v in tree. All the nodes in the path from u to root(parent,grand-parent,.. of u) should recalculate the CVS. When multiple queries are given the deletion may be from different parts of the tree. But each node should be processed once.

Algorithm 4 Dynamic tree edge deletion of thread t

```
Input: Data Graph D,Query Graph Q,Delete u-v(u is parent of v).

Output: CVS updates.

1. w=v
2. for each parent of w(till root)
    (a) mark w for t
3. for each parent of w(till root)
    (a) if mark at w is t, acquire lock for w
4. for each parent of w(till root)
    (a) if mark at w is t and able to acquire locks for all childs of w
    (b) Recompute NEC of w
    (c) if not a previously computed NEC then
    (d)         C(w)=FindCandidates(w,D) update
    (e) Release all locks
```

The above algorithm marks all the parent nodes from a particular deleted edge. This marking helps to make sure only one thread process one node. The processing will be done such a way that no parent nodes are processed if any child of the parent is unprocessed. So the processing order will be leaf to root. This will change the values inside A and so as B.

4.6 Adding Edge in Data Graph

This will also add entries into B. But A will not be changed since no edges in query graph is changed.

Algorithm 5 Dynamic data edge addition of thread t

```
Input: Data Graph D,Query Graph Q,Delete u-v(u is parent of v).
Output: CVS updates.
  1. w=v (for u also)
  2. for each child of w(till |Q| length)
     (a) if acquire locks for all childs of w
     (b) foreach NEC's x
     (c) C(x)=$FindCandidates$(x,D) update
     (d) Release all locks
```

This algorithm moves to |Q| length from both u and v in a BFS fashion. At each Data node it is checked whether it can be added to CVS of any of the NECs. When there is an overlap of regions of different thread locks are used to synchronize them. Since the Data graph is huge and query graph is small and so the possibility of two edge deletion happening near(edge length $<$ |Q|) is small, the number of locks waits will be minimal.

4.6.1 Parallel Execution

All the four quires can be processed in parallel since they are operating on different data. 4.3 and 4.4 are processing on D while 4.5 and 4.6 are processing on A and B. So it is safe to run them parallel. Parallel adding to A and B doesn't make the answer inconsistent. It will add a vertex into the CVS which will be removed when exact matching is performed in the end.

4.6.2 Inferences

So a FindTree match algorithm should be done at each stage of the output to get the new possiblities. But this stage is the costliest of all the four steps. Even one vertex is added to any two CVS, this will cause to recompute most of the permutations and combinations again. So there is no computational advantage by dynamic processing. Running a subgraph isomorphism solution after applying all edge updates becomes equally fast as the dynamic version.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

The state-of-art algorithms for Subgraph Isomorphism use different pruning techniques to avoid the false candidates as early as possible. The part of the algorithm which considers all possibility is the most time consuming part. Since we can’t avoid checking some possibility the one efficient way of making this part faster is decreasing the number of candidates. The complexity of the pruning technique helps to remove more false candidates thus making the algorithm faster. The possibility checking part is made faster by using GPU by checking different possibilities in different threads. Thus a million possibilities are checked in parallel. The dynamic version of the problem is

REFERENCES

H He, A. K. S. (2008). Graph-at-a-time:query language and access method for graph database.

H SHang, X. L., Y Zhang (2008). An efficient algorithm for testing subgraph isomorphism.

Jinsoo Lee, R. K., Wook-Shin Han (2012). An in-depth comparison of subgraph isomorphism algorithms in graph databases.

L P Cordella, C. S., P Foggia (2004). A subgraph isomorphism matching algorithm for matching large graphs.

P Zhao, J. H. (2010). On graph query optimization on large networks.

S Zhang, Y. J., S Li (2009). Gaddi:distance index based subgraph matching in biological networks.

Ullmann, J. (1976). An algorithm for subgraph isomorphism.

Wook-Shin Han, J.-H. L., Jinsoo Lee (2013). Turboiso:towards ultrafast and robust subgraph isomorphism search in large graph database.

Xiaojie Lin, Z. W., Rui Zhang (2012). Efficient subgraph matching using gpus.