Sub-graph Isomorphism in GPU

George Joseph- CS15M021

Department of Computer Science and Engineering, Indian Institute of Technology Madras, India.
georgejs@cse.iitm.ac.in

1 Introduction

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.

1.1 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.

1.2 Problem

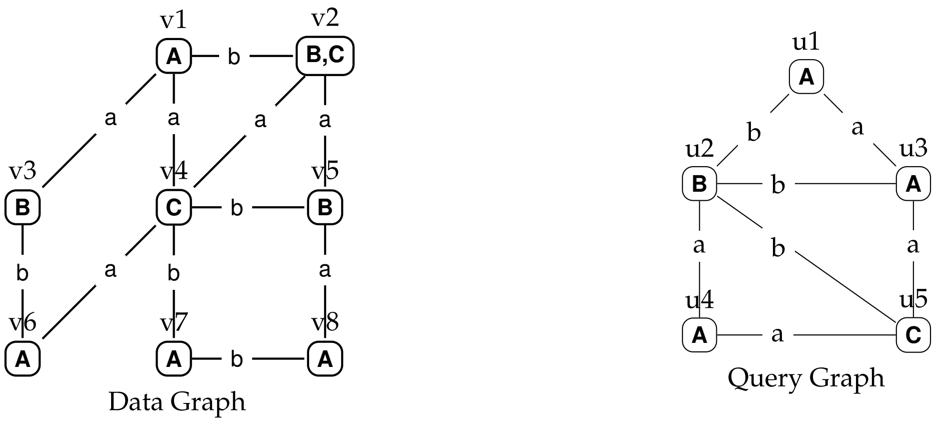
Input: A data Graph D, and Query Graph Q. The graphs D and Q are undirected with nodes and edges having label.See Figure 1.
Graphs given as adjacency list.
Output: Give all the matching mapping of each node in Q to node in D

2 Background Study

2.1 Current Algorithmic Literature

2.1.1 Generic Algorithm

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



Data Graph

Query Graph

Figure 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 $RefinedSet$ prunes out some nodes in the candidate set. The $IsJoinable$ checks whether the map is right. The $UpdateState$ moves to next state (adds new vertex to map).The $RestoreState$ 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 $NextGraph$ 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 $RefinedSet$ will remove $v1$ from $C(u4)$. If Map has these values $\{(u1, v1)\}$,$NextGraph$ returned $u2$, $RefinedSet$ returned $\{v3, v2, v5\}$ and current v is $v5$ then procedure $IsJoinable$ check whether there is an edge between $v1$ and $v5$ like the one between $u1$ and $u2$.

2.1.2 Ullmann Algorithm

This algorithm[2] is simple.The $\mathit{FindCandidates}$ finds same degree nodes. The $\mathit{NextVertex}$ takes the next node in input.The $\mathit{RefinedSet}$ removes nodes already mapped. The $\mathit{IsJoinable}$ iterate over the neighborhood and checks corresponding edge exists. The $\mathit{UpdateState}$ and $\mathit{RestoreState}$ adds and removes the vertex from map respectively.

2.1.3 VF2 Algorithm

VF2 algorithm was proposed in [3].The $\mathit{NextVertex}$ takes the next connected vertex. The $\mathit{RefinedSet}$ 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

2.1.4 QucikSi Algorithm

QuickSi algorithm was proposed in [4].The $\mathit{NextVertex}$ takes vertices in the most infrequent vertex first order.The $\mathit{RefinedSet}$ uses connectivity to mapped vertices to prune.The $\mathit{RefinedSet}$ only iterates over mapped adjacent vertices.

2.1.5 GADDI Algorithm

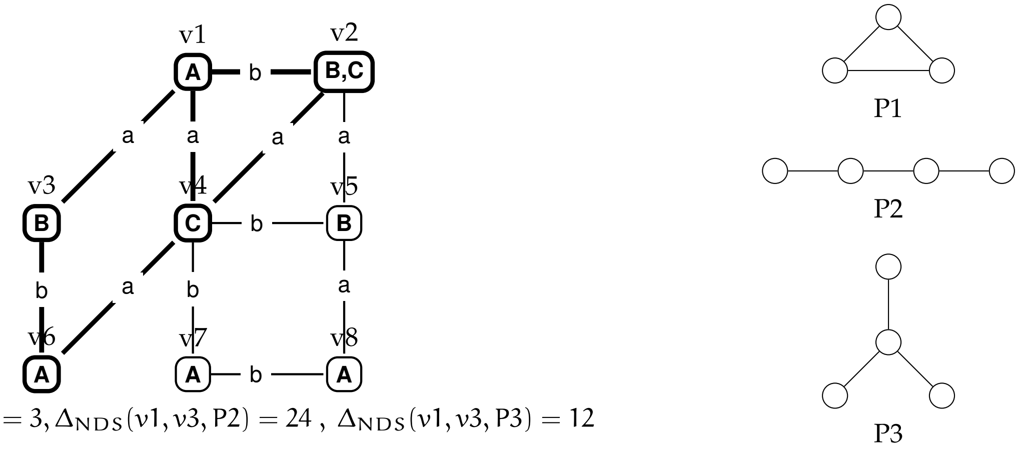
GADDI was proposed in [5]. They use the neighboring discriminating substructure(NDS).$\Delta_{NDS}(u, v, P)$ is the number of occurrences of P in induced subgraph $N_k(v) \cap N_k(u)$. $N_k(u)$ is the graph having all the edges in k distance from u.A matrix L is created such that each row corresponds 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 $\mathit{NextVertex}$ takes the one next in the DFS Tree from the vertex. The $\mathit{RefinedSet}$ 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'.

2.1.6 GraphQL Algorithm

The GraphWl was proposed in [6]. They use neighborhood signatures. The neighbor of the vertex is encoded as the collection of labels of its neighbors. The $\mathit{RefinedSet}$ pruning is based on this signature. This is a one hop signature. For example the $\mathit{sig}(u1) = \{B, A\}$ in Figure 1.



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

Figure 2: GADDI NDS Calculation

2.1.7 SPath Algorithm

The SPath Algorithm was proposed in [7]. 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.

2.1.8 STWig Algorithm

The STWig Algorithm was proposed in [8]. 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.

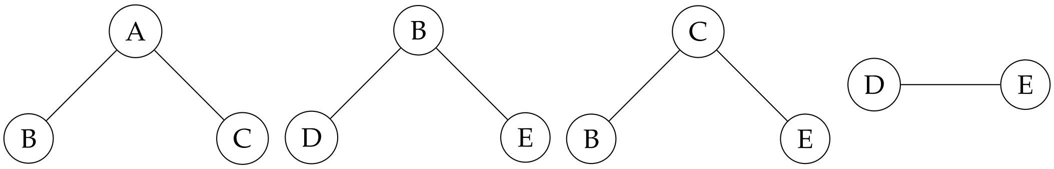


Figure 3: STWig Decomposition

2.2 Findings

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.

3 Implemented Algorithm

3.1 TurboIso

It was proposed in [9].$\texttt{Turbo}_{\texttt{iso}}$ 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.

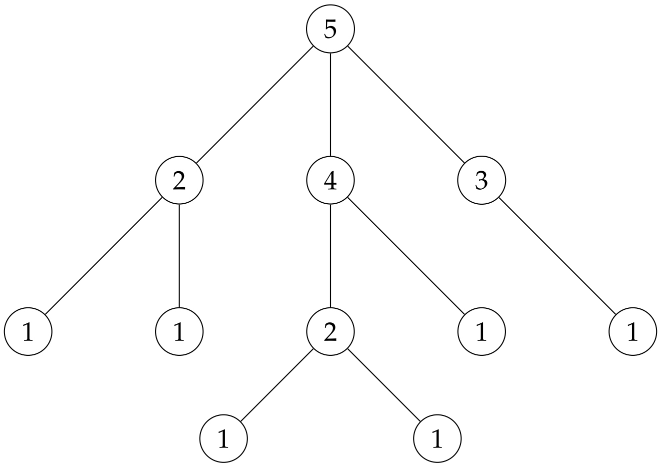


Figure 4: NEC Numbering

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.

3.2 Parallel Implementation

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.

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

```
    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),Map(m)
Output: Mapping of nodes from query graph to data graph.

```
1. if i= |V(q)|
2. CheckMap(m) and report the Map(m) if true and return
3. find the NEC of the vertex
4. foreach node v in CVSGen(u)
    (a) add to map (u,v)
    (b) PermandComb(i+1,M)
    (c) remove(u,v)
```

Procedure: CheckMap()
Parallel check of existence of query graph.
Input: Data Graph D,Map m,Query Graph Q.
Output: true/false.

```
1. running on node v
2. iterate over all neighbors of vertex v.
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 the each permutation possible from the candidates of each vertex in query graph. 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 Findings And Improvements

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 combinations of mapping is currently generated in CPU. It can be moved to GPU.

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

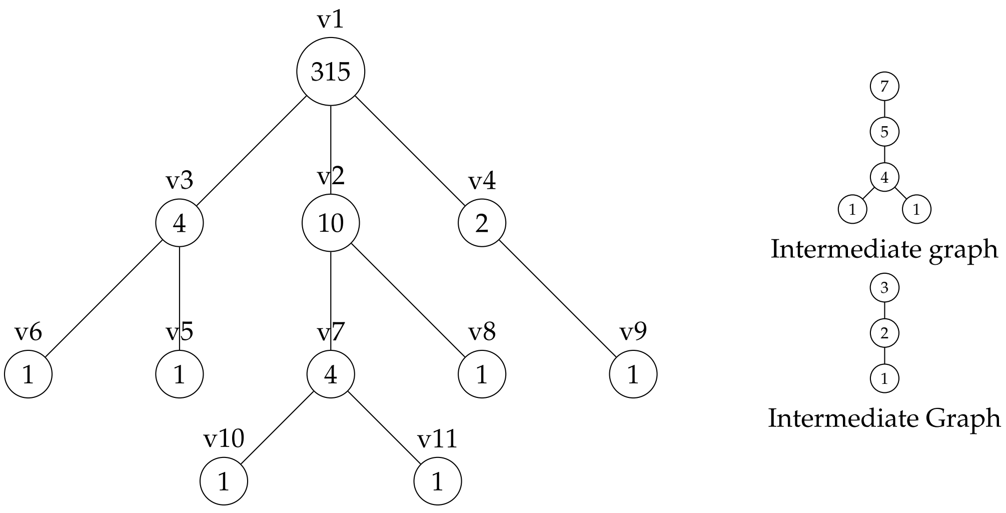


Figure 5: NEC based on primes

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.

4 Future Advancements

1. Using clique to clique edges
   This will help to reduce the amount of search since clique has all the subgraphs.

2. Using multiple trees to create CVS
   We can use multiple trees to create the unique numbering so that it will reduce the candidate set size.

3. Multiple Query Graphs
   Processing for multiple query graphs simultaneously.

4. Dynamic Queries
   The query graph and data graph can change. So we update the previous answers accordingly.

	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 1: Dynamic Changes Difficulty level

When an edge is added to query graph we only need to check on all the previous results to find the required answer. When an edge is added in Data graph we need to check only those graphs that the new edge can be part of. When an edge is removed from data graph we need to remove those answers which have that edge. Removing an edge in query graphs is difficult since we need the partial answers of the computation we done before.

4.1 Summary

1. We studied the Subgraph Isomorphism problem and the state-of-art algorithms. We came to know they are using different pruning techniques to avoid the false candidates as early as possible.

2. We implemented the $\texttt{Turbo}_{\texttt{iso}}$ algorithm.

References

[1] R. K. Jinsoo Lee, Wook-Shin Han, “An in-depth comparison of subgraph isomorphism algorithms in graph databases,” 2012.

[2] J. Ullmann, “An algorithm for subgraph isomorphism,” 1976.

[3] C. S. L P Cordella, P Foggia, “A subgraph isomorphism matching algorithm for matching large graphs,” 2004.

[4] X. L. H SHang, Y Zhang, “An efficient algorithm for testing subgraph isomorphism,” 2008.

[5] Y. J. S Zhang, S Li, “Gaddi:distance index based subgraph matching in biological networks,” 2009.

[6] A. K. S. H He, “Graph-at-a-time:query language and access method for graph database,” 2008.

[7] J. H. P Zhao, “On graph query optimization on large networks,” 2010.

[8] Z. W. Xiaojie Lin, Rui Zhang, “Efficient subgraph matching using gpus,” 2012.

[9] J.-H. L. Wook-Shin Han, Jinsoo Lee, “Turboiso:towards ultrafast and robust subgraph isomorphism search in large graph database,” 2013.