Sub-graph Isomorphism in GPU

George Joseph- CS15M021[1]

[1]*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 it is having lots of applications. Since the current era is trying to extract lots of features from images, videos,audios,etc using many pattern recognition techniques. Everything is having a implication to the sub-graph matching or graph mining. Many people have tried hard to find a good way to solve the problem. The problem being an NP Hard no one yet succeeded. The numerous cores of GPUs may help us to solve this problem faster. Each node search is independent so they can be done 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 graph D and Q are undirected with nodes and edges have no label.
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 for subgraph isomorphism will help us to study the aspects of state-of-art algorithms in deep.

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 $FindCandidates$ finds the vertices in Datagraph which can be mapped to query vertex. The $NextVertex$ finds the next vertex in querygraph which should try to be map.
The $RefinedSet$ it 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.

2.1.2 Ullmann Algorithm

This is the first algorithm came out for subgraph matching.The $FindCandidates$ finds same degree nodes. The $NextVertex$ takes the next node in input.The $RefinedSet$ removes nodes already mapped. The $IsJoinable$ iterate over the neighborhood and checks corresponding edge exists. The $UpdateState$ and $RestoreState$ adds and removes the vertex from map respectively.

2.1.3 VF2 Algorithm

The $NextVertex$ takes the next connected vertex. The $RefinedSet$ 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

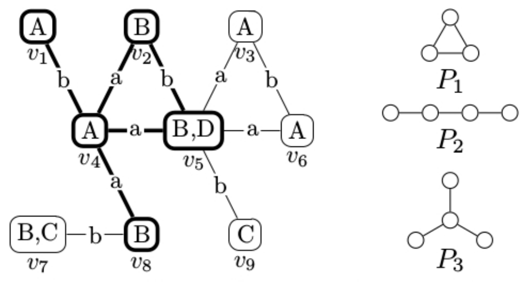
2.1.4 QucikSi Algorithm

The $\mathsf{NextVertex}$ takes vertices in the most infrequent vertex first order.The $\mathsf{RefinedSet}$ uses connectivity to mapped vertices to prune.The $\mathsf{RefinedSet}$ only iterates over mapped adjacent vertices.

2.1.5 GADDI Algorithm

They use the neighboring discriminating substructure(NDS).$\delta_{NDS}(u, v, P)$ is the number of occurrence of P in induced subgraph $N_k(v) \cap N_k(u)$. $N_k(u)$ is the graph having all the edges in k hopes from u.A matrix L is created such that each row corresponds to an induced graph g and each column represent each pattern.
The $\mathsf{NextVertex}$ takes the one next in the DFS Tree from the vertex. The $\mathsf{RefinedSet}$ 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 $u^{'}$.



$$\Delta_{\mathrm{NDS}}(v_1, v_4, P_1) = 3,\ \Delta_{\mathrm{NDS}}(v_1, v_4, P_2) = 8,$$
$$\Delta_{\mathrm{NDS}}(v_1, v_4, P_3) = 24$$

Figure 1: GADDI NDS calculation

2.1.6 GraphQL Algorithm

They use neighborhood signatures. The neighbor of the vertex is encoded as the collection of labels of its neighbors. The $\mathsf{RefinedSet}$ pruning is based on this signature. This is a one hop signature.

2.1.7 SPath Algorithm

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 $\mathsf{RefinedSet}$ pruning is based on these signatures.

2.1.8 STWig Algorithm

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.
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.

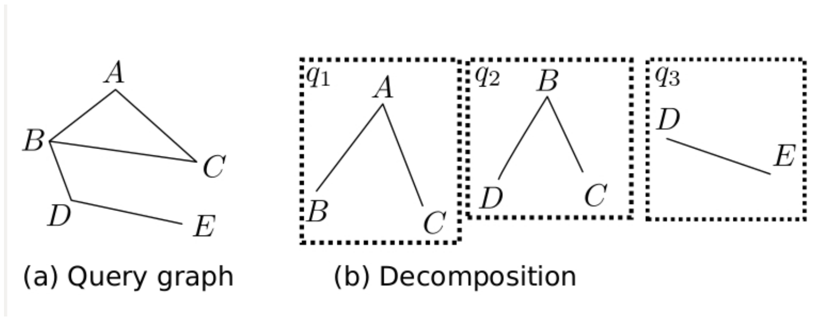


Figure 2: STWig Decomposition

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

$\texttt{Turbo}_{\texttt{iso}}$ 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.

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.

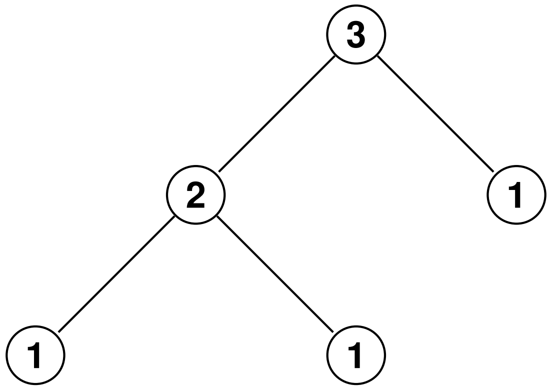


Figure 3: NEC Unique Numbering

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: Permanence()
Parallel check all possible permutation and combination.
Input: Data Graph D,Query Graph Q,CVS,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 if true
3. find the NEC of the vertex
4. foreach node v in CVS(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.
```

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 Results

Will be added

3.5 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 have 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 do search the subgraphs in this node.
It actually captures all subgraphs at the root. See the figure below.

The root node is getting 6 since the subgraph with $node1$ child has id 2 and it is also present in the root. The id 4 is given because id 2 is present there two times. The id 3 is used to represent graphs with a child 4. That is why root got 6(3*2).

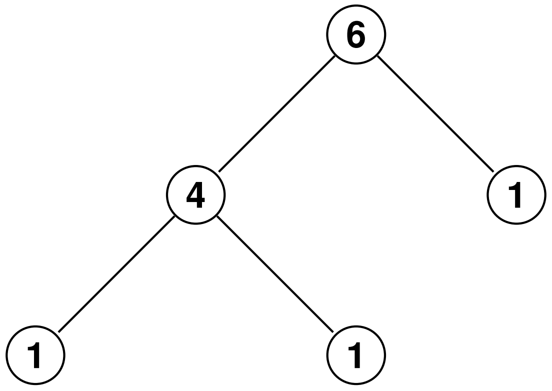


Figure 4: Numbering based on primes

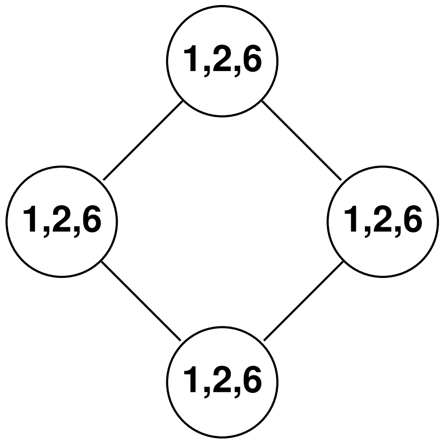


Figure 5: Candidates given in data graph

But this doesn't came to be a good way.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 keeping only primes. See the figure 5.

In the first iteration every node gets id 1. In the next iteration every body gets 2 since each of them has a child 1. Then in third iteration everybody will get a 3 which gets multiplied by 2 already present in the node giving 6. The id 3 is assigned since it has a 2 node as child.

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 makinf 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 Conclusions

1. There are many places we can improve the performance like in dynamic queries
2. The state-to-art algorithms are relying on candidate set to make the algorithm faster.
3. The parallelism property of the problem is suitable for GPUs.