Implementation of “Subgraph Matching with Set Similarity in a large Graph Database”

By
Sharvari Barve

Advisor
Carlos Rivero
Acknowledgements

I would like to thank Prof Carlos Rivero for his guidance throughout the project and Prof Leon Reznik for his guidance in the colloquium.
The paper implemented in the MS capstone project is “Subgraph Matching in with Set Similarity in Large Graph Database” by authors “Liang Hong, Lei Zou, Xiang Lian and Philip S.Yu”[1]. The report is my understanding of all the methods used in the paper[1] and my implementation of them[1].

Motivation for the problem:-
The authors of the paper[1] have focussed on set similarity matching as opposed to precise vertex tag matching[1]. The queryNode comprises of a number of attributes with different significance which need to be used when finding a matching dataNode in the main graph[1]. Most of the algorithms developed in this paper are considering the fact that computation of such similarity between vertex attribute sets is tougher than simple matching[1].

Scope of the Problem:-
The problem described in the paper[1], is results that are obtained from a big graph, when a query is given[1]. The results are not only of the same form as the query graph, but also should consider how alike the queryNode is to the corresponding mapped dataNode using the significance of each attribute in their attribute list as the authors have described in the paper[1]. The important distinction of in this problem in the paper[1] is the variable weight of each attribute of the queryNodes which disallows exact node matching[1].

Preprocessing:-
The preprocessing in the paper[1], contains the following steps, which allow to reduce the number of potential nodes for further consideration from the data graph by forming tree structure and then eliminating dataNodes from it.

The authors of the paper[1] describe, inclusion relation is related to the subset- superset property, which helps us in narrowing down the potential matches[1]. In general, if A and B are two sets of nodes in the data graph and B is the superset of A and C is a generic queryNode set.

Similarity(A,C)<cutoff => Similarity(B,C) <cutoff[1].

This helps us in trimming down further nodes in the data graph.

Inverted TreeStructure Construction
Initially a treeStructure needs to be built using certain attributes subsets which occur more than the cutoff number of times[1]. In this structure, nodes are used in a level wise fashion. The condition described in the paper[1], to be satisfied is that the upper level node i.e list of elements contained in the node is the subset of the corresponding connected node on the lower level[1], and this condition is to be satisfied for all the levels.
Learning neo4j
I learnt the basics of Neo4j, and created a sample database of Students, which consists of a student Node comprising of a name, age and courses taken. The students who have common courses have a connection between them. I learnt the language Cypher used for query and manipulation in Neo4j[3].

Java Project
I have created a java project which follows the following structure
Capstone Project
-OfflineStructure.java
-dataNode.java
-QueryNode.java
-TreeNode.java
-verticalPruning.java
-horizontalPruning.java
-antimonotonePruning.java
-structuralPruning.java
-MappingClass.java
-MappingtoGraph.java
This could be considered a summary.

DataNode
DataNode is a node that is part of the main graph. DataNode contains the elements like the courses taken by the student and the students who are connected.

DataGraph
DataGraph is the entire graph, which consists of all the dataNodes and all the connections present in the graph.

QueryGraph
QueryGraph is similar to DataGraph, the main difference is that query graph has much lesser nodes than data graph[1].

QueryNode
QueryNode is similar to dataNode, however is part of query one.

Offline tree structure
One of the most important parts of the paper[1], the offline structure is an important preprocessing step, the operations we do with it, will significantly reduce the time in later steps[1].
This structure is constructed like a tree structure, with each node being a treeNode object, having an arraylist of children (the children point in upward direction), the child is one of the subsets of the parent node[1]. This structure initially is the same for all queryNodes calculated from the most occurring elements and their subsets as TreeNodes in the tree structure as from the idea in paper[1].

To construct the first frequent pattern, I use a hashmap which is maintained as a counter, and out of that the required number of elements are chosen, from which this tree is formed from treeNodes with maximum courses first and which match with actual dataNodes[1].

TreeStructure is constructed with a LinkedHashMap<Integer, ArrayList<TreeNode>> tree;
1->(treeNode1,treeNode2)
2->treeNode3
The key represents the level,
The treeNode by itself contains Set<String> elements as well as ArrayList<Treenode> children.

The newer treeNodes are then populated with content according to the subsets of the lower treeNodes, in case a treeNode is already present, the same instance of the treeNode is used for the superset’s pointer.
Clarification about inverted Lists:-

The inverted list has a treeNode pointing to a set of datanodes. Thus, whenever pruning occurs in the treeStructure, it is made to happen in the inverted list too as it is mentioned in paper[1].

Inverted Lists are constructed in such a way that each treeNode points to a list containing dataNodes if the specific attribute set present in the treeNode is the same as the attribute set present in the dataNode[1].

In the images given below, rather than printing out the address of the treeNode I have printed out the courses that it contains.

The structure of the invertedList is

HashMap<TreeNode, Set<DataNode>> potentialMapping=new HashMap<TreeNode,Set<DataNode>>();
Thus each treeNode points to a set of dataNodes which have all the content present in the treeNode.
That means v1,v2,v3 have Physics as their single constituent, while v4,v5 have Maths, English as their sets.

**Similarity between Nodes**

The paper[1] has mention of Jaccard value for checking how similar a queryNode is to a dataNode. The intersection and union of the two set attributes of the nodes are needed to find the similarity[1] for which Guava library is used.

**Jaccard coefficient:**

Jaccard Coefficient= Σweighted set intersection/Σweighted set union [1]
This similarity metric is used both for horizontal pruning as well vertical pruning[1].
**Horizontal pruning**[1]

The authors of the paper[1] describe, the basics of horizontal pruning is finding after adding how many number of elements in the query graph, would the similarity cutoff still be valid which gives us the maximum number of attributes a treenode contains[1]. Thus, using inclusion relation and maximum number of elements allowed for the subgraph, the levels below the nodes with lists having the maximum elements can be removed[1].

There is a need to find the maximum elements that can be contained in the set[1] of a treeNode in the lattice so that it can still be above the similarity threshold with the queryNode.

Example:
Consider that we have a frequent Pattern comprised of

K={"literature","programming"," geography","history","maths"}  
And we need a similarity of 0.8
Consider a querySet with attributes q= {"programming","literature"}
Let the weights be
Literature-0.2
Programming-0.1
Geography-0.3
History-0.2
maths-0.2

When given this query q, after adding how many elements to q will the similarity constraint still be followed will lead us to the maximum allowable length[1].
Consider adding history to q
q’={programming,literature,history}

Let’s find similarity between q and q’
Using Jaccard coefficient
Similarity=(0.1+0.2)/(0.1+0.2+0.2)=0.6

But we need a similarity of 0.8, hence we can prune all the treeNodes with element sets of length greater than equal to 3 , if however we needed a similarity less than 0.6, we could have added more courses.
This above example is inspired from an example in paper[1].
How is it implemented?
As we can see the frequent structure tree is pruned horizontally, thus the maximum length is determined above each the similarity can never be satisfied[1]. The entire functionality of horizontal Pruning is present in horizontalPruning.java, which consists of methods like findSimilarity and pruneHorizontally which takes arguments related to the maximum size of the set which would still follow cutoff.
Thus when the tree is pruned, the dataNodes in the inverted list of the specific pruned treeNode is removed out of consideration as a candidate match[1].

Vertical Pruning[1]
The authors of the paper[1], state this as the elements in the sets are not in random order, but are arranged according to the higher priority of their weights. The authors use this property to find minimum number of elements that need to be in common, and since the elements are in order it is called prefix matching[1]. Vertical pruning basically occurs by removing the nodes in order one by one, till the cutoff is not satisfied[1]. It means removing all the nodes in the lattice, that do not contain this matching prefix with the query which in turn is needed to satisfy the cutoff[1].

Example:
Let K be the frequent Pattern {"literature","history","geography","programming","maths","music"}.
In a similar way to horizontal pruning, a query set q {programming, literature} is considered for vertical pruning.
Consider the following weights
Literature-0.2
History-0.1
Programming-0.4
Maths-0.2
music-0.1

We need to find out how much of the query set needs to be present in the set of the treeNode at the minimum for the similarity threshold to be followed.
Consider that the similarity threshold is 0.5
Let \( q = \{ \text{programming}, \text{literature} \} \)
The courses are ordered according to their weights[1].
Now we remove the element programming
\( q_1' = \{ \text{literature} \} \)
Let's calculate similarity between both of these
\[ \text{similarity} = \frac{0.2}{0.4 + 0.2} = \frac{2}{6} = 0.33 \]
Thus we need to have programming as an element of the element set in the considered
treeNode, the remaining TreeNode which do not contain the “programming” can be removed[1].
This example is inspired from an example in paper[1].

**How is it implemented?**
Vertical pruning is related to pruning according to the minimum prefix required, as seen here the
treeNode which do not contain the prefix are removed from the structure. Vertical Pruning
contains methods for not only finding similarity but pruning vertically which would prune the
offline structure according to the set given and prune the nodes not containing all the elements
in the set. Thus the dataNode in the corresponding inverted lists of the pruned TreeNode can
be removed from consideration[1].

![Tree Structure](image.png)
Pruning on nodes not containing {maths} and length>2

In general after pruning the inverted lists(candidates for a single node) would look like,
Consider a queryNode q, if its structure looks like this

![Diagram of potential data nodes](image)

What does this mean? It means that q has only to be matched with v4 and v5 now v1, v2, v3 are out of consideration. So basically for every queryNode we store this tree structure along with its set of potential matches.
Its stored like Map<queryNode, treestructure>
Tree Structure contains the actual tree structure as well as the potential matches.

**Anti Monotone Pruning[1]**

As initially described in the paper about inclusion relations and thresholds[1], the authors have applied this relation to reduce the nodes of the lattice structure. The connected lower level node is the superset of the upper level one, thus if the upper level node of the lattice has similarity less than the cutoff, the connected nodes on the lower level can be discarded[1].

\[
\text{anti Monotone similarity} = \frac{\sum \text{weights of query set}}{\sum \text{weight of union}} \quad [1]
\]

<table>
<thead>
<tr>
<th>Inverted List</th>
</tr>
</thead>
<tbody>
<tr>
<td>[programming] a5</td>
</tr>
<tr>
<td>[literature, programming] a3</td>
</tr>
<tr>
<td>[maths, geography, programming] a5</td>
</tr>
<tr>
<td>[literature, maths, programming] a1</td>
</tr>
</tbody>
</table>

The above is a screenshot between courses in treeNode in first part and courses in treeNode to the dataNode it which has the same ones.

Thus as each treeNode at each level is a subset of the connected node of the level below, it obviously won't have similarity greater than the threshold if its subset has similarity lesser than
the threshold, since in fact we will add more elements in the union, which will reduce the similarity[1].

Consider an example
Programming:0.25
Literature:0.25
History:0.25
maths:0.25
querySet {“programming”,“literature”}
When consider elements of treeNode eg
Consider treeNode with element set{“history”,“maths”}
Antimonotone similarity=(history+maths)/(sum of courses )
If the similarity threshold is 0.6
We can prune all its supersets eg {“history”,“maths”,“literature”}
The example above is inspired from an example in paper[1].

**How is it implemented?**

Anti monotone pruning is related to the property that if a certain treeNode has similarity less than the threshold[1], all its nodes below will be effectively its supersets will have similarity less than that, thus they can be pruned. I have created a recursive function which effectively prunes the treeNode as well as all its supersets.

**Universal set of Elements:-**

If we consider a data graph, I built a treeSet considering all elements of DataNodes, to form an universal set[1].

Eg
Consider s1 with courses {Math, Science}
S2 with courses {English, Math}
Thus the universal set would be {English, Math, Science} ordered alphabetically.

**Data Signature:-**

Data Signature as described in the paper[1] is calculated for every DataNode, the bit vector is calculated using Java BitSet for the dataNode itself[1].

Then another bit vector is calculated by considering all the bitSets of its neighbours[1].

Combined DataNode signature={self bit vector, Union of neighbours};[1]
Query Signature:-
Query signature as described in the paper[1] is for a single queryNode stores the bitset of the queryNode as well as bitSets of its neighbours, the difference is that over here due to smaller number of queryNodes, all the neighbours are stored in the signature[1].

Finding Candidates to Prune:-
The algorithm for this process is described in paper[1] as, similarity is found between the dataNode bitSet and queryNode bitSet[1].
If this similarity is less than required value, this dataNode is removed from consideration[1].
If the similarity constraint is satisfied, we need to calculate the similarity with all the connected nodes of queryNode with the union of neighbours of dataNode[1].
If this similarity is less than required value, we need to remove this dataNode from consideration[1]. Thus we consider both similarity as well as structure, while finding the candidates to consider for a single queryNode[1]

Matching steps
The authors of the paper[1] have proposed an algorithm which uses dominating set i.e all nodes are either present in or are neighbours of nodes in the dominating set[1].
Thus instead of finding matches for the entire structure given in the query, only matches for dominating set are obtained[1]. Thus the authors emphasise that this would in turn reduce costs of not only finding potential matches for each node, but also matching the entire structure between query and the main graph[1].
The distance preservation[1] conditions to be satisfied include that at least one neighbour must be common in the main graph, between two vertices of dominating graph of a corresponding length between 1 and 2 or they must be adjacent [1].
The first step that the authors use is to find only the corresponding graph in the main graph for the graph from the dominating set[1]. This algorithm uses states to represent graphs which are incomplete matches to the query graph[1]. This incomplete answer is explored till either a complete query match is found, or the earlier conditions break causing to discard the potential match[1].
The next step is to find the result for entire query graph. Thus only matching using similarity is used for the limited number of vertices in dominating set, while the entire result is obtained by the distance conditions imposed between the connections in dominating and other vertices[1].

Finding Dominating Set:-
We consider the query graph to find its dominating set. I have not used the algorithm in the paper, however have implemented a simpler greedy algorithm from another source[4]. Since this cannot find the exact minimum dominating set, I have implemented a simpler version to focus on other parts of the paper[1]. Here the dominating set is found by considering the queryNodes with highest number of neighbours[4], adding the node as well as its neighbours to a set and then considering the queryNode with next highest number of Nodes but not present in the HashSet[4].
**Find Dominating Graph:**

After finding this set, edges need to be constructed between queryNodes of this set. The edges are considered in the following way. The authors of the paper[1] have formulated the following: if the queryNodes are connected in the queryGraph, they are connected in this graph assigned a value of 1 or if they have a neighbour in common, they are assigned the edge is assigned a value of 2. The figure below referred from [1] shows how queryGraph is transferred to dominating query Graph.

![Diagram](image1)

Figure: from queryGraph to graph[1]

The queryNodes which are dark, are the ones in the dominating graph [1].

![Diagram](image2)

Figure:from queryGraph to graph[1]

A and B have a neighbour in common, hence the weight between A and B in the dominating graph is 2 [1]

![Diagram](image3)

Figure:From queryGraph to graph[1]

All the above images are referred from the paper[1].
A and B have two nodes distance between them, hence the weight in the dominating graph is 3

**Dominating Matching algorithm:-**

This algorithm described by the authors in paper[1], works by using the dominating graph and matching it to the main graph. I have used an HashMap mapping from a queryNode to a dataNode. Thus the dominating matching algorithm given in paper[1], is a recursive one, in which it recursively tries all pairs, checks if it satisfies the edge weights and if finally the mapping is able to map all queryNodes from dominating graph, the mapping is returned.

In this case, We select a node a from queryNodes and another node b from dataNodes, If a has an edge of 2 with another node c, b would have one node in between the mapping of c[1]. If such a mapping is not present, we add the given pair and then go on mappings unless rule is violated[1].

**How is it implemented?**

This is basically after finding the set, uses a recursive function according to the algorithm in [1], implemented in MappingClass.java. It has different methods called

- `isS`
- `findMapping`
- `isDistancePreservationSatisfied`
- `is3neighbours`
- `is2neighbours`
- `is1neighbours`
- `isSimilaritySatisfied`
- `isFullMappingCovered`

This class estimates if a part of dominating query graph is worth considering further or not. Then we get a series of matches all which satisfy the conditions of being a candidate.

**Finding Match for the entire queryGraph**

Now we have the mapping between the dominating nodes of the query graph with the data graph[1]. The next algorithm described by the authors in paper[1], used to calculate the the exact match for the query graph, then match of just the dominating graph. The abstract algorithms are given in the paper[1] for both finding mapping of the dominating query graph as well as finding mapping of the entire graph. Initially the mapping which was calculated in the previous algorithm is used to initialize the structure[1]. It checks for the limitations imposed between the dataNodes as given by data preservation rule to find the other dataNodes[1]. As described in the algorithm, while iterating through each non-dominating queryNode, it iterates through all of the one distance neighbour and two distance neighbours[1]. In this case it goes through all the candidate dataNodes for the neighbours[1], and find their neighbours respectively and then considers a pair found from this if it satisfies the similarity consideration[1]

**Implementation:-**
The implementation is in MapToGraph.java. The finalMap is instantiated with each of the potential dominating ones[1], to see if that's the right path to go, the moment there is a mismatch or structurally it does not fit with the main query graph, I stop considering it according to the algorithm in paper[1]. Many of the methods are similar to finding a mapping previously, however, the previous mapping depends on structural as well as node similarity, while over here we only use structure to find matches, and verify at end if node similarity is also satisfied[1].

Results

Various test cases were created to see how the time of getting a match changes with the size of the query graph, as well as under different test cases in which there is a change in how connected most of the queryNodes are to each other, type of student’s courses. Initially, the graph below is time in seconds vs size of the data Graph for a constant size queryGraph of size 5.

The graph below is increasing the size of query graph for a constant data graph of size 21.
A factor noticed is when we have a fixed data graph, the time initially decreases with size of query Graph as we can see in graph, but later increases, so we have a point where the factor of lesser number of matches in the factor is overcome by the price of matching a larger graph, a point noted by the authors[1] also.

The large dataset was used to see how the implementation works, when subjected to a huge dataset. The following results were obtained.

For the entire dataset

<table>
<thead>
<tr>
<th>Size of query Graph</th>
<th>Time taken(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>9119</td>
</tr>
<tr>
<td>6</td>
<td>17349</td>
</tr>
</tbody>
</table>

There are a number of factors affecting the results, some of which agree with the author’s views[1]. One factor is the size of the dominating set which would obviously influence, as there are two different ways to match different kinds of queryNodes.

Another one is number of distinct elements in the entire queryGraph which is used to construct the tree structure, obviously larger the tree structure, it takes longer time to work on it, which is similar to point raised by authors about the occurrences of constituents in a dataNodes.

**Conclusions from Results**

There is a noticeable change in runtime with number of distinct elements changing [1], this point is mentioned by the authors as well.

The dominating set of the query graph has an effect on the time to get the answer, this might be due to more time consuming matching algorithm for dominating set by using similarity and using only neighbor relations for the rest of the graph.

**Difficulties faced**

I had a tough time finding libraries to suit the lattice formation in the paper[1]. The best source that I found in Stanford Java library related to Lattice[2]. However I realized that implementing a tree structure on my own would probably be the best idea, since I can tweak it to suit the project. I used a few readymade libraries, however like Guava libraries mainly for Set functions like finding intersection, union and subsets.

I had a tough time testing with a large database, since attributes needed to be provided for thousands of dataNodes.
**Future work**
The implementation can be improved, to give reasonable runtime for queryGraphs>8 nodes, this would include some sort of caching, so that recursive function does not work on the same partially matched queryGraphs again and again something like this explanation[7].
References