Capstone Project Report
Title:- Analysis of approximate subgraph matching algorithms over a graph database
Pranav Dadlani(pad2039@rit.edu)
Advisor:- Professor Carlos Rivero(crr@cs.rit.edu)

1. Introduction

Graph Databases
Graph database is a storage engine that is specialized in storing and retrieving vast networks of data. It stores nodes and relationships and allows high performance traversal. Nodes represents an entity and edges represent the relationship between the different entities. Properties can be added to nodes and relationships(edges) in a graph. Moreover, graph databases allow faster retrieval when compared with relational databases. Graph databases are used to study and analyze interactions which is the reason they are used to mine data from social media.

Approximate Subgraph Matching
Exact subgraph matching problems are NP complete problems. The challenges involved in subgraph matching is the lack of standardized schema, label and structural noise, dynamic updates and massive volume of data. There is also a need for faster query response time. Applications of the subgraph matching algorithms,
- Social Networks
- Biological Networks like PPI Networks
- Computer Vision

Given a target graph T and query graph G, approximate subgraph matching is to find the top-k matches for the query G in T in the presence of structural and label noise.

2. Goal
The aim of the project is understanding the framework built on top of Neo4j, studying the different approximate matching algorithms and comparing as well as analyzing the performance of these algorithms in terms of time, precision and recall on large datasets such as the Biogrid Yeast.
The goal is to determine which is the best algorithm implementation to use given a specific query type depending on the time and efficiency of retrieval of matched results(top-k). The problem is approached by initially generating small query graphs to find the scenarios in which the algorithms do not perform well and then reproducing these kind of query graphs on larger datasets such as Biogrid Yeast.

3. Neo4j Framework
Neo4j is an open-source NoSQL graph database implemented in Java. The existing open source Neo4j framework does not have implementations of the different approximate subgraph matching algorithms. The new framework is designed in a way such as to calculate the top-k solutions given a data and query graph. The matches found could be different for the different algorithms which are implemented in the framework. The performance of the matching
algorithms can be measured by evaluating the correctness of the matches as well as the time taken by the algorithm to evaluate the matches. Label as well as structural noise can be added to increase the number of queries. In the framework, a threshold minSigma value can be set which is either the Levenshtein or Jaccard distance that can be specified in the framework. To add structural noise, the number of hops can be varied in the framework.

There are two types of string label distances namely,

1. **Levenshtein Distance**: It is the minimum number of replacements, insertions and deletions needed to transform a sample string a to string b.
2. **Jaccard Distance**: The Jaccard coefficient measures similarity between finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sets.

Graph inputs are in either gdl(GNU Data Language) or gql(Graph Query Language). Query graphs will be generated by extracting subgraphs of different shapes(paths, trees) and labels from the target graphs.

The different class files and their functionalities are mentioned below,

1. **ExampleApproximateTest class** - The directory name of the data and query graphs need to be specified in gdl format. The Neo4j Parser parses the gdl and creates a database of the data graph and stores it in the db directory. Nema and Tram are the two approximate matching algorithms which are implemented in the framework.

When finding the approximate match using Nema, the quality of the match is determined by a cost metric which is sum of label matching cost and difference of neighborhood matching cost aggregated for all the nodes in the graph. The lowest cost for the subgraph is chosen as the best match for the subgraph. Lambda and sigma are the only varying parameters. Lambda is the label cost parameter which varies from 0-1. A lower value of lambda means that the label cost is given very less priority (the individual node label cost is multiplied by a lower factor) whereas if the lambda value is high the label costs are given a higher priority and the neighborhood matching cost is multiplied by a lower factor. The solutions are calculated for different combinations of values of lambda and sigma. Individual scores for each solution is calculated and the one with the lowest score is supposed to have the most accurate solution. ‘k’ can also be specified which means we can get the top-k approximate matches for a particular query graph.

The solutions generated indicates the node numbers in the data graph which match the corresponding node numbers in the query graph. For example, if nodes 100, 101 and 102 from the data graph match the query graph nodes 0, 1 and 2, the solution generated for the triangle shaped query will be \{0=100, 1=101, 2=102\} with a score of 0. A score of 0 indicates that the match is supposed to be perfect but in some cases Nema gives a score of 0 but the solution set may not be the best and there can be better solutions which may be generated.

2. **rsc/Nema/Example1/data.gdl** - The knowledge or data graph is specified in gdl format.

An example of that format is given as,

```graph
node v0<id="A">;
node v1<id="F">;
edge e0(v0,v1)<id="y">;
```
Here “A” and “F” are node labels and “y” is the edge label and is an edge connected between the two nodes v0 and v1.

The following sections describe the two algorithms in detail,

4. Nema
Nema is a neighborhood based subgraph matching technique for querying large scale database networks. The subgraph matching cost function can have false positives and that is why Nema does not perform well when top k matches need to be retrieved(k>2). Proximity among the entities rather than the actual shape of the graph is used to identify the optimal approximate matches for a given query. Nema is useful when we need to find the topmost match for a query in a target graph. It definitely performs better than keyword search.[1]

The score calculation for Nema is done as follows for each node,

\[
\text{Individual Node Matching cost } f(v,u) = \lambda (\text{Label Cost}) + (1-\lambda)\text{Difference between neighboring cost vectors}.[1]
\]

Lowest k scores give the best k approximate matches.

5. Tram
Tram works on the principle of random walks. It divides the data graph into local neighborhoods and filters parts of the graph that are less likely to produce feasible matches. Random walks is the repeated transition of walker from current node to randomly selected node. Firstly, \(\beta\) signatures of each node (\(\beta\)-structural similarities) are calculated. It tries to match the query graph with these \(\beta\) signatures. The highest score value is chosen as the best Match(Contrary to Nema).[2]

The section below explains the different problems of Nema and Tram.

6. Analysis of Nema and Tram for different types of queries

1.

Data Graph

Query

Butterfly type structure where nodes have similar labels
Nema result,
\{0=0, 1=1, 2=1, 3=1, 4=1\}--0.0
which gives 1 edge and is **incorrect**
Tram result, \{0=0, 1=1, 2=2, 3=3, 4=4, 5=5\}

2.

![Data Graph](image1)

![Query Graph](image2)

when query graph has more number of nodes as data graph
Nema result, \{0=0, 1=-1, 2=-1, 3=-1, 4=1, 5=2, 6=3\} is the result with score as 0 which is clearly **wrong**
Tram result, No solution

3.

![Data Graph](image3)

![Query Graph](image4)

when top-k matches need to be generated for particular queries(k=3)
Nema result, \{0=0, 1=4, 2=2\}. Only 1 solution is generated which is **inconsistent**
Tram result, \{0=0, 1=4, 2=2\}, \{0=0, 1=4, 2=5\}, \{0=0, 1=1, 2=2\}. Two solutions are **exact** matches whereas one of them is an approximate match.
4. when a path is given as a query,
Nema result, {0=0, 1=1, 2=2, 4=4, 5=5, 6=6}. (Exact Match)
Tram result, {0=0, 2=4, 3=5, 4=2, 5=6}

5. In presence of structural noise
Nema result, {0=0, 1=1, 2=2, 3=3, 4=4, 5=5, 6=6, 7=-1}
Tram result, No solution found
When there is label noise, running the above query graph for Nema; minSigma = 0.1 and lambda = 0.6 generates 
\{0=0, 1=1, 2=2, 3=3, 4=4, 5=5, 6=6, 7=7, 8=8, 9=9, 10=10, 11=11\} but with a high score of 7.36
For greater values of minSigma, the query generates false positives and inconsistent results. Running ScoringFunctions.java for the above query graph gives the following result, 
\{0=0, 1=1, 2=2, 3=3, 4=4, 5=5, 6=6, 7=7, 8=8, 9=9, 10=10, 11=11\}; Cost: 3.1999

In case of Tram, for different beta values from 0.1 and 0.7. Also with lambda and minSigma values ranging from 0.1 and 0.9, Tram does not produce any solution. Tram thus does not perform well when the graph consists of label noise.
β values are \{0.1, 0.2\}

For Q1)
When running Nema for path AB, with lambda value as 0.6 and minSigma value <0.4, it gives good approximate results. \((\{0=0, 1=1\}, \{0=13, 1=16\})\). It is only when minSigma value >0.4, Nema produces incorrect results(with -1's).
Tram on the other hand produces good results with any minSigma value. It produces good results for top k-matches for the path given in query 1.
Tram results for top-5 matches are good in this case(considering 2 hops),
\[
\begin{align*}
\{0=0, 1=2\} & \Rightarrow 1.0410104621492482 \\
\{0=0, 1=4\} & \Rightarrow 1.0410104621492482 \\
\{0=0, 1=5\} & \Rightarrow 1.0410104621492482 \\
\{0=0, 1=3\} & \Rightarrow 1.0410104621492482 \\
\{0=0, 1=6\} & \Rightarrow 1.0410104621492482 
\end{align*}
\]

For Q2)
This is a typical problem of Nema. When running the query for values of minSigma<0.4, Nema produces one single edge as the output\((\{0=0, 1=1, 2=1\})\). In short, it produces false positives. For minSigma>0.4, there are -1’s introduced in the output which are incorrect results.
Results for Tram are again good in this case even when top-5 matches need to be retrieved.
\[
\begin{align*}
\{0=0, 1=3, 2=2\}, \{0=0, 1=2, 2=4\}, \{0=0, 1=2, 2=5\}, \{0=0, 1=2, 2=3\}, \{0=0, 1=2, 2=6\}
\end{align*}
\]

For Q3)
When running Nema for the query, the results are again good for lower values of minSigma(<0.4) but with a higher score\((\{0=0, 1=1, 2=2, 3=8\})=8.0\). But even with lower values of minSigma, when asked to retrieve the top-5 matches, top 2 seem to be correct but after that there are only false positives or incorrect matches.
In case of Tram, the results returned for are \(\{0=13, 1=17, 2=16, 3=11\}, \{0=0, 1=3, 2=2, 3=8\}, \{0=13, 1=17, 2=16, 3=10\}, \{0=0, 1=3, 2=2, 3=9\}, \{0=0, 1=1, 2=9, 3=10\}\). As we can see, the last result obtained is incorrect.
But overall, Tram produces much better top-k results for all 3 queries.
8. Tram returns no solution whereas Nema returns an incorrect result with a score 0((0=0, 1=-1))

9. For the above graph, Nema gives an exact match({0=0,1=1,2=2})
Tram gives the top result as {0=0, 2=2}
This shows that Tram does not perform well when query graphs are smaller in size.

10. For beta value from 0.1 - 0.5, Tram gives good results for k=5, lambda=0.6, minSigma=0.5
Nema gives
\{0=0, 1=2, 2=3, 3=4, 4=9, 5=7, 6=8\}, \{0=1, 1=0, 2=3, 3=4, 4=9, 5=7, 6=8\}, \{0=1, 1=2, 2=3, 3=4, 4=6, 5=7, 6=8\}, \{0=1, 1=2, 2=3, 3=4, 4=5, 5=7, 6=8\}, \{0=0, 1=2, 2=3, 3=4, 4=6, 5=7, 6=8\}
Both results are generated with score 0.
The second result obtained does not have an edge FD but Nema gives a perfect match with a score of 0.

Nema provides 2 solutions with a score of 0 which indicates that it found 2 approximate matches. These are false positives generated. The top-2 solutions returned by Nema are,
\{0=0, 1=-1, 2=-1, 3=-1, 4=1, 5=2, 6=3\} and
\{0=1, 1=-1, 2=-1, 3=-1, 4=0, 5=2, 6=3\}
The value -1 indicates that there has been no matching vertex found and in spite of that the score calculated is 0 which indicates the error in score calculation.

The next section explains the generation of butterfly shaped queries with nodes having similar labels. The main motive behind generating such kind of queries was to reproduce the problem of Nema(similar to Example 1) with or without the presence of structural and label noise on large graph datasets like Biogrid Yeast.
7. Generation of Butterfly shaped queries with similar labels
These type of queries consist of 5 labels and 6 edges.

For a graph above, the query generated is as follows,

```graph
node v0<id={"L"}>
node v1<id={"L"}>
node v2<id={"L"}>
node v3<id={"L"}>
node v4<id={"L"}>
edge e0(v0,v1)<<
edge e1(v1,v2)<<
edge e2(v1,v3)<<
edge e3(v1,v4)<<
edge e4(v2,v4)<<
edge e5(v0,v3)<<
```

8. Steps to generate the queries
Initially set the threshold levenshtein distance(minSigma), label type(if vertex has multiple labels) and no of hops. Setting different values for these variables helps to generate different type of queries all with the same butterfly structure.
The pseudo code for generation of butterfly type structure is given below,
The ButterflyGeneration.java module consists of the following two functions,
generateButterfly()

1. Select a random node.
2. Till a butterfly structure is created,
   a. For each neighbor, calculate the levenshtein distance between the random node selected the current neighbor(n).
      i. If the distance is >=minSigma
         distanceWithinRange(n)
      Else
         1. if(hopCount=2 or hopCount=1)
            a. For each neighbor i of n
               i. Calculate LevDistance of i with random node selected
               ii. Add ‘i’ node to a secondHopList
               iii. If the distance is >=minSigma
a. `distanceWithinRange(i)`

2. `if(hopCount=2)`
   A. `For each node in secondHopList`
      a. `For each neighbor j`
         i. `Calculate LevDistance of j with random node selected`
         ii. `If the distance is >=minSigma`

`Function distanceWithinRange(Long vertex)`
Add this vertex to list currentVertices
If sizeof(currentVertices)>=4, ...(Adjacent nodes to the random found)
   i. Call `checkForWings()` and if returns true, the butterfly structured query is generated.
      Call `writeGraph()` which writes the query to a file.

`checkForWings()` function checks for the edges between distinct vertices in the list. For example, the list which consists of nodes a,b,c,d, if it finds an edge between any two vertices(a and c), then it checks for an edge between b and d similar to the wings of the butterfly. If there exists two distinct edges between 4 different vertices, then it returns true.

9. **BioGrid Yeast Database**

The BioGrid is a public database that consists of PPI(protein protein interactions) data from humans and organisms.

BioGrid Yeast Database consists of 3112 nodes, 12519 edges.
Each node is a protein interaction.
A node has 2 labels, id(Unique id) and go(protein constituents)
For example,
Example node,
node v1<id="yjl153c">, go="0043170", "0044238", "0044237", "0044424", "0016853">;

10. **Results for Butterfly shaped query on Yeast**

minSigma =1(exact same labels), label type="go", and exact butterfly type structure,
0 hops,
Query generated is as follows,
graph
node v83<go="0044237">;
node v1333<go="0044237">;
node v652<go="0044237">;
node v1375<go="0044237">;
node v2508<go="0044237">;
edge e0(v83,v1333)<>
edge e1(v652,v1375)<>
edge e2(v2508,v83)<>
edge e3(v2508,v1333)<>;
edge e4(v2508,v652)<>
edge e5(v2508,v1375)<>

Since the label type="go" consists of an array of proteins, a random one is selected from the set and inserted as part of the node in the query.

Running Nema for the above query for the graph database generates the following result,
{83=0, 2508=1745, 652=769, 1333=1745, 1375=769}--0.0
{83=3111, 2508=2053, 652=3074, 1333=2053, 1375=3074}--0.0
{83=1, 2508=133, 652=3072, 1333=133, 1375=3072}--0.0
{83=4, 2508=1597, 652=1538, 1333=1597, 1375=1538}--0.0
{83=3108, 2508=364, 652=640, 1333=364, 1375=640}--0.0

From the above result, all the matches returned by Nema are wrong. Nema returns only one edge like Example 1. The score calculation is incorrect since it gives 0 which is supposedly the best solution.

The top 2 results generated by Tram are as follows,
{83=2963, 652=2724, 2508=2886, 1333=2593, 1375=1587}--4.783683289131438
{83=2963, 652=2724, 2508=2886, 1333=2593, 1375=822}--4.783683289131438

In this case, tram is unable to retrieve exact matches too.

For 2 hops,

2 hop query,

graph
node v769<go={"0044237"}>
node v131<go={"0000267"}>
node v1639<go={"0043234"}>
node v2276<go={"0043234"}>
node v2603<go={"0044237"}>
edge e0(v769,v131)<>
edge e1(v1639,v2276)<>
edge e2(v2603,v769)<>
edge e3(v2603,v131)<>
edge e4(v2603,v1639)<>
edge e5(v2603,v2276)<>

Running Nema for the above query gives the top 5 results as follows,

{769=769, 131=131, 2603=769, 2276=769, 1639=769}--0.0
All the matches returned by Nema are wrong. Nema returns only one edge like Example 1. The score calculation is incorrect since it gives 0 which is supposedly the best solution.

The top 2 results generated by Tram are as follows,

\{769=2593, 131=822, 2603=2886, 2276=2724, 1639=2963\} -- 4.783683289131438
\{769=2593, 131=822, 2603=2963, 2276=2724, 1639=2724\} -- 4.783683289131438

Tram also is unable to retrieve exact matches.

11. Conclusion

- Tram works well when the data graph is large.
- Tram is better than Nema when we want to get the top-k matches. For the top-1 match, Nema functions better.
- Nema produces a lot of false positives and incorrect solutions.
- Tram provides approximate solutions or no solutions in the worst case (more than 2 hops away).
- Performance in Tram comes at the expense of time. It gets slower due to the random walks calculated for each node. Bottleneck is when evaluating the results for larger graphs.
- Tram performs poorly in terms of structural noise. It does not consider solutions which are not direct neighbors.
- There seems to be problem with the overall algorithm of Nema due to certain cases not being handled well. There could be a problem with calculation of scores causing the results to include false positives.

12. Future Work

- Auto-generation of some more queries replicating the problems.
- Testing for different types of queries such as paths, trees, subgraphs on bigger databases like IMDB, Facebook to check for the performance in terms of correctness and time.

13. References

