Using the GeX Approach for Approximate Matching on Graph Databases

by

Gireeshma Bokka Reddy

A Project Submitted in Partial Fulfillment of the Requirements for the Degree of Masters of Science in Computer Science

Supervised by
Dr. Carlos R. Rivero
Dept. of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York
Spring 2016
Abstract

Increase in the popularity of social networking websites has increased the need for graph databases as relationships between the data hold an important role here. Instead of forming relationships when the queries are run, a graph database stores the relationships which can be referred to when required.

The keyword-based query model fails in case of complex queries as the results consist of the keywords being related topologically and it cannot process two predicates that specify constraints on the data at the same time. To overcome these drawbacks, Subgraph Matching on the graph databases is used where the queries which are being matched are considered as subgraph patterns to be extracted from the larger graph databases. There are two methods which can be adopted to match the patterns i.e. exact and approximate subgraph matching. In this project we have worked with approximate matching as it helps in obtaining results which are similar to the query as compared to exact matching where the exact pattern is returned.

The GeX top-k query answering algorithm has two phases, cursor initialization and cursor access and solution building [1]. As the GeX querying algorithm is more challenging because of unbounded nodes and edges, label approximation and undirected edges, both the phases of the algorithm which are based on the principles of the Threshold Algorithm have been modified.

The initial goal of the project was to understand the pseudocode of the algorithm and its implementation as mentioned in the paper. The implementation of the same was done using java and Neo4J.

The objective of the project was to study the results obtained on implementing the GeX querying algorithm.
Acknowledgements

I want to wholeheartedly thank the faculty of the Computer Science Department for the support and guidance provided to me throughout my course at RIT where I got to learn and grow. I want to specifically thank the following faculty members that made it possible for me to achieve the completion of my Master's Degree in Computer Science; Dr. Hans-Peter Bischof, Dr. Carlos Rivero and last but not least, Dr. Reynold Bailey. I want to express my sincere gratitude for getting the opportunity to prove myself and to learn from the people whom I came across in the duration of my course.
List of Figures

Figure 1.2.1  Exact and Approximate match with the query graph  1
Figure 5.1.2.1 Creation of nodes  7
Figure 5.1.2.2 Creation of relationship between two nodes  7
Figure 5.1.2.3 Creation of multiple relationships between nodes  8
Figure 5.1.2.4 Creation of all nodes and and relationships between them  9
Figure 5.1.2.5 Graph with multigraph connections  10
Figure 5.1.2.6 Demo dataset  10
Figure 5.1.2.7 Creation of demo dataset on Neo4J  11
Figure 5.1.2.8 Dataset being inserted into Neo4J  12
Figure 5.1.2.9 Result after loading the dataset on Neo4J  12
Figure 5.1.2.10 Result after creating the structure  13
Figure 5.1.2.11 Result after creating the relationships  14
Figure 5.1.2.12 Result after querying the top ten nodes  14
Figure 5.2.1.1 Creating a graph using the java API in Neo4J  16
Figure 5.2.1.2 Result 1 after inserting the dataset into Neo4J  17
Figure 5.2.1.3 Result 2 after inserting the dataset into Neo4J  17
Figure 5.3.1.1 Screenshot of input taken through console from user  19
Figure 5.3.3.1 Depiction of nodes  21
Figure 5.3.3.2 Query graph for exact matching  22
Figure 5.3.3.3 Solution 1 for exact matching  22
Figure 5.3.3.4 Solution 2 for exact matching  23
Figure 5.3.3.5 Solution 3 for exact matching  24
Figure 5.3.3.6 Query graph for approximate matching  25
Figure 5.3.3.7 Solution 1 for approximate matching  25
Figure 5.3.3.8 Solution 2 for approximate matching  26
Figure 5.3.3.9 Solution 3 for approximate matching  27
Contents

Abstract
Acknowledgements
List of Figures
List of Tables

1 Introduction.................................................................................................................................................. 1
  1.1 Motivation............................................................................................................................................... 1
  1.2 Background ........................................................................................................................................... 1
  1.3 Benefits of Approximate Matching on Graph Databases ................................................................. 2

2 System Specifications .................................................................................................................................. 2
  2.1 The GeX top-k query answering algorithm and its implementation using java .................................. 2
  2.2 Storing the dataset using Neo4J .......................................................................................................... 2
  2.3 Limitations of Neo4J ............................................................................................................................ 2

3 The GeX top-k query algorithm .............................................................................................................. 3
  3.1 Definitions of the terms used in the algorithm .................................................................................. 3
    3.1.1 Data graph ....................................................................................................................................... 3
    3.1.2 Query graph .................................................................................................................................... 3
    3.1.3 Path ................................................................................................................................................ 3
    3.1.4 Semantic Relatedness .................................................................................................................... 3
    3.1.5 Approximation cost function ....................................................................................................... 3
    3.1.6 Reduced SR .................................................................................................................................... 3
    3.1.7 Scoring function ............................................................................................................................ 4
    3.1.8 The Data model ............................................................................................................................. 4
    3.1.9 The Query model ........................................................................................................................... 4
    3.1.10 The Query answering model ....................................................................................................... 4
    3.1.11 The Ranking model .................................................................................................................... 4
  3.2 The Algorithm ......................................................................................................................................... 4
    3.2.1 Cursor Initialization ....................................................................................................................... 5
    3.2.2 Cursor Access and Solution Building ............................................................................................ 5

4 Dataset ....................................................................................................................................................... 5
  4.1 Source .................................................................................................................................................. 5
  4.2 Background ........................................................................................................................................... 5
  4.3 Description ............................................................................................................................................ 5

5 Project Development .................................................................................................................................. 6
5.1 First Milestone ........................................................................................................................................... 6
  5.1.1 Working with Neo4J .......................................................................................................................... 6
  5.1.2 Cypher Query Language .................................................................................................................... 6

5.2 Second Milestone ....................................................................................................................................... 15
  5.2.1 Working with the Neo4J API .............................................................................................................. 15
  5.2.2 Using Neo4J server with JDBC ......................................................................................................... 18

5.3 Third Milestone ......................................................................................................................................... 18
  5.3.1 Cursor Initialization ........................................................................................................................... 18
  5.3.2 Cursor access and solution building phase ......................................................................................... 20
  5.3.3 Results ................................................................................................................................................ 21

6 Conclusion .................................................................................................................................................. 27

7 Bibliography ............................................................................................................................................... 27
1 Introduction

1.1 Motivation
Increase in the popularity of social networking websites has increased the need for graph databases as relationships between the data hold an important role here. Instead of forming relationships when the queries are run, a graph database stores the relationships which can be referred to when required. Thus finding new approaches to query from a graph database will result in a significant advancement in dealing with graph databases.

1.2 Background
The methods which have been adopted earlier for querying graph databases such as using SPARQL for RDF datasets aren't easy to use as they expect the person querying to know the structure in which the data is stored and the relationships that have been stored between the data. Another popular method which is the keyword-based query model fails in case of complex queries as the results consist of the keywords being related topologically. This method cannot process two predicates that specify constraints on the data at the same time.

To overcome these drawbacks, Subgraph Matching on the Graph Databases is used where the queries which are being matched are considered as subgraph patterns to be extracted from the larger Graph Databases. There are two methods which can be adopted to match the patterns i.e. exact and approximate subgraph matching. In this project we work with approximate matching as it helps in obtaining results which are similar to the query as compared to exact matching where the exact pattern is returned.

![Figure 1.2.1: Exact and Approximate match with the query graph [2]](image)

In the above diagram, the query graph represents the input entered by the user. The subgraph marked in un-dotted circle represents an approximate match where the nodes are exactly the same.
but the relationship between Node 1 and 3 is missing. The subgraph marked in dotted circle represents an exact match where the nodes and the relationships are exactly the same.

1.3 Benefits of Approximate Matching on Graph Databases
Exact subgraph matching restricts the number of results as the solutions should have both nodes and relationships which are exactly the same as the query graph. Approximate subgraph matching on the other hand provides flexibility and a wide variety of solutions to choose from. The solutions consist of all the paths present between a given set of nodes. Approximate subgraph matching has a major benefit of adding error tolerance to the query results.

2 System Specifications

2.1 The GeX top – k query answering algorithm and its implementation using java
The existing algorithms already provide solutions for keyword and structure based matching in graphs. The Graph- eXplorer algorithm provides an algorithm that solves both these along with approximate subgraph matching[1]. It extends the solutions by providing flexible solutions in the form of approximate matching. The implementation of GeX top – k query answering algorithm is done using java. Java version 7 has been used mainly. The flexibility of the GeX algorithm is mainly implemented with the help of data structures which have been used extensively.

2.2 Storing the dataset using Neo4J
The graph database management system used to store the data is Neo4J. It is open source and is considered as one of the most popular graph database systems available[4]. It has various features such as its compatibility with java which make it apt to be used in this project. It can be connected to the algorithm implemented in java using two ways. The first way is to use the available java API and the second is to use the JDBC connection[4]. The Neo4J interactive database opens on localhost and Cypher Query Language is used to work with it. The data stored using Neo4J is in the form of nodes and relationships which form the building blocks of a graph database and features such as assigning labels and colors are available which make the content of the graph easier to decipher[4].

2.3 Limitations of Neo4J
- Viewing the entire data is very tedious as the nodes keep bouncing off each other and spreading out. This causes a lot of difficulty while browsing through huge datasets.
- Simple queries which do not need relationships take much longer as compared to other database management systems. Hence Neo4j is efficient only in case of queries that involve the usage of relationships.
- The material available online for help in case of issues is very less which leads to a dead end.
3 The GeX top k-query algorithm

3.1 Definitions of the terms used in the algorithm

3.1.1 Data graph
It can be defined as a connected, directed and labeled multi-graph that is used to represent the data which is present in the database[1].
It is represented by $G = (N, E, L_N, L_E)$

$N$ = The set of nodes forming the data graph
$E$ = The set of directed edges which connect the nodes
$L_N$ = The set of node labels
$L_E$ = The set of edge labels

3.1.2 Query graph
A query is a tuple which is entered by the user. A query graph is defined as the graph which is formed using the inputs obtained from the user[1].
It is represented by $q = (N_q, E_q, L_N^q, E^q, V, C)$

$(N_q, E_q, L_N^q, E^q)$ = a connected labeled multi-graph with variables as described above.
$V$ = The vertices of the query graph
$C$ = The conditions which are placed on $V$ i.e. the vertices.

3.1.3 Path
A path is defined as the sequence of consecutive edges between nodes in a graph[1].
It is denoted by $p = e_1, ..., e_m$

3.1.4 Semantic Relatedness
Semantic Relatedness is defined as a collection of node pairs that are related in a meaningful way. Each node pair is given as $e=(n, n') \in SR$ and it is assigned a label $\lambda(e) \in L_C$ and a path $p(e)$ in the data graph connecting the node pairs $n$ with $n'$[1].

3.1.5 Approximation cost function
For a graph $G$ and semantic relatedness $SR$, an approximation cost function $c$ for $SR$ is a monotonically increasing function over the paths of $G$ such that $c(e) = 0[1]$.

3.1.6 Reduced SR
Reduced $SR$ contains all the node pairs which are a subset of $SR$ and denote the same relationship[1].
3.1.7 Scoring function
For a query, \( q = (N_q, E_q, L_{N_q}, L_{E_q}, V, C) \)[1]
\( \mathcal{E} = \) The approximate embedding for a query \( q \)
\( c = \) The approximate cost function
The scoring function is given as follows:

\[
S(\mathcal{E}) = \frac{\alpha}{|N_q|} \sum_{n \in N_q} d_L(\lambda(n), \lambda(f(n))) \quad \text{------------------------ (1)}
+ \frac{\beta}{2|E_q|} \sum_{e \in E_q} \left( d_L(\lambda(e), \lambda(g(e))) + \frac{c(g(e))}{MC} \right) \quad \text{------------------------ (2)}
+ \frac{\gamma}{|C|} \sum_{c \in C} (1 - s(c)) \quad \text{--------------------------- (3)}
\]

Part (1) Measures the semantic relationship between a query node and its data node[1].
Part (2) Measures the semantic and structural approximation between query edge and its corresponding edge in reduced SR[1].
Part (3) Measures query conditions by applying the inverse of the function used for condition evaluation[1].

3.1.8 The Data model
The data model represents data as a connected graph allowing parallel edges or multigraphs and labeled or unlabeled nodes and edges[1].

3.1.9 The Query model
GeX provides an expressive graph-based query language which goes beyond the keyword-based approach. This query language does not require to precisely use the graph vocabulary and structure while querying the graph[1].

3.1.10 The Query answering model
This model deals with node mismatches and adjacency misses in query answers. It considers node label approximations, edge label approximations and adjacency approximations[1].

3.1.11 The Ranking model
To rank the top-k answers, the GeX ranking model is used. Here the scoring function \( S \) is applied on \( \mathcal{E} \) to get the goodness measure of the answer. \( \mathcal{E} \) denotes the approximate embedding for a given query[1].

3.2 The Algorithm

The Graph explorer (GeX) algorithm has a goal to obtain the top k patterns matching the given query. The input taken from the user is used to form the query graph. The GeX algorithm is then
used to obtain approximately matching subgraph of the data graph[1]. The algorithm has two phases which are the following:

- Cursor Initialization
- Cursor Access and Solution Building

3.2.1 Cursor Initialization
The first phase of the algorithm is to collect the set of edges that exactly or approximately match the edges of the query graph[1]. The following functions are performed on each edge of the graph database:
1. Finding the data graph edges matching with the query graph edges.
2. Storing the matched data graph edges as cursors.

3.2.2 Cursor Access and Solution Building
The second phase of the algorithm is to use the cursors or edges obtained in the Cursor Initialization stage. These cursors are used for solution building. The cursors are used to form the subgraph specified by the user[1].

The following functions are performed on each of the stored cursors:
1. The Scoring function is calculated for a combination of the cursors and sorted in ascending order.
2. The top-k values among the sorted cursors are considered as the final results.

4 Dataset

4.1 Source
The data was obtained from the BioGRID website which is an updated interaction repository. BioGRID stands for Biological General Repository for Interaction Datasets. The data is updated on a monthly basis[3].

4.2 Background
The dataset consists of protein and genetic interactions. It is a public database and mainly maintained for curation drives[3]. One of the purposes for which this data is used is to gain insights on the patterns that help in understanding and improving human health.

4.3 Description
- >1,066,335 interactions from Saccharomyces cerevisiae(yeast), Caenorhabditis elegans (roundworm), Drosophila melanogaster (fly) and Homo sapiens (humans) in CSV format are available for download[3].
- This data was obtained from 56,621 publications.
- The version of the dataset that has been used to work with the GeX algorithm is 3.4.133
- The data is available in standardized formats as a combination of all organisms as well as specific to each organism.
5 Project Development

5.1 First Milestone

5.1.1 Working with Neo4J

- Neo4J is an open source software available on the Neo4J website[4]. The Windows installer of Neo4J Community Edition was downloaded from the website and the prompts were followed[4].
- On running Neo4J, the database location is asked to be chosen. On choosing the location, the database needs to be started.
- Neo4J opens up at http://localhost:7474/browser/

5.1.2 Cypher Query Language

Phase 1: Learning from resources available online
1. Installed Neo4J on the system[4].
2. Followed YouTube videos and got familiar with Neo4j.
3. Created the nodes and understood the following.
   a. Command used to create the node is CREATE().
   b. The curved brackets “()” depict a node[4].
   c. MATCH (n) RETURN n [4]
      This command is used to find all the n nodes and show the n values.
   d. CREATE (n:User) RETURN n
      This command creates a node with the label “User”.
      “Return n” shows the node created.
   e. MATCH (n:User) RETURN n.
      This command gives all the nodes matching the label “User”.

Phase 2: Creating a set of nodes and experimenting with the relationships
Example: Creating a family graph:

1. CREATE (n:Person { name : 'Harry' })
2. CREATE (n:Person { name : 'James'})
3. CREATE (n:Person { name : 'Lily'})

- On running the above three statements, the result is:
  Added 1 label, created 1 node, set 1 property, statement executed in 62 ms.
- There is a change in the time taken each time the query is run.
- Once the nodes are created, the graph looked like this using the following command:
  Match (n) return n
Figure 5.1.2.1: Creation of nodes

- The following code was used to form the relationships within a family:

MATCH (a:Person),(b:Person)
WHERE a.name = 'Lily' AND b.name = 'James'
CREATE (a)-[:WIFE_HUSBAND]->(b)

Figure 5.1.2.2: Creation of relationship between two nodes

MATCH (a:Person),(b:Person)
WHERE a.name = 'James' AND b.name = 'Lily'
CREATE (a)-[:HUSBAND_WIFE]->(b)
The following commands were run next:

a. MATCH (a:Person),(b:Person)
   WHERE a.name = 'James' AND b.name = 'Harry'
   CREATE (a)-[:FATHER_SON]->(b)

b. MATCH (a:Person),(b:Person)
   WHERE a.name = 'Harry' AND b.name = 'James'
   CREATE (a)-[:SON_FATHER]->(b)

c. MATCH (a:Person),(b:Person)
   WHERE a.name = 'Lily' AND b.name = 'Harry'
   CREATE (a)-[:MOTHER_SON]->(b)

d. MATCH (a:Person),(b:Person)
   WHERE a.name = 'Harry' AND b.name = 'Lily'
   CREATE (a)-[:SON_MOTHER]->(b)
On running the above commands, the graph looks as below:

![Database Information](image)

**Figure 5.1.2.4: Creation of all nodes and and relationships between them**

- To check for the existence of multigraphs on Neo4j, the following graph was created:

  CREATE (n:Person { name : 'Sirius' })

  a. MATCH (a:Person),(b:Person)
     WHERE a.name = 'James' AND b.name = 'Sirius'
     CREATE (a)-[r:FRIEND]->(b)
  b. MATCH (a:Person),(b:Person)
     WHERE a.name = 'Lily' AND b.name = 'Sirius'
     CREATE (a)-[r:FRIEND]->(b)
  c. MATCH (a:Person),(b:Person)
     WHERE a.name = 'Harry' AND b.name = 'Sirius'
     CREATE (a)-[r:FRIEND]->(b)
  d. MATCH (a:Person),(b:Person)
     WHERE a.name = 'Harry' AND b.name = 'Sirius'
     CREATE (a)-[r:GODFATHER]->(b)

  The resulting graph is as follows:
Phase 3: Working with a smaller csv file
A demo dataset was used to see the working of Neo4J while forming the graphs. The dataset is as follows:

<table>
<thead>
<tr>
<th>ID</th>
<th>Case Num</th>
<th>PrimaryType</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1 A</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>2 B</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>3 C</td>
<td>Y</td>
</tr>
<tr>
<td>5</td>
<td>4 D</td>
<td>Y</td>
</tr>
<tr>
<td>6</td>
<td>5 E</td>
<td>X</td>
</tr>
</tbody>
</table>

Using the following commands I could form the nodes and the relationships shown below:

```sql
// Create IDs
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM 'file:///Users/Lenovo/Downloads/Dataset/Demo.csv' AS row
CREATE (:ID {variableName: row.ID });
```
// Create nodes and indexes
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM 'file:///Users/Lenovo/Downloads/Dataset/Demo.csv' AS row
MERGE (variableType:VariableType { productName: row.PrimaryType}) ON CREATE SET variableType.ID = row.ID;
CREATE INDEX ON :ID(variableName);

//Create relationships
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM 'file:///Users/Lenovo/Downloads/Dataset/Demo.csv' AS row
MATCH (variableType:VariableType { productName: row.PrimaryType})
MATCH (id:ID {companyName: row.ID })
MERGE (id)-[:TYPE]->(variableType);

This resulted in the graphs which looked as follows:

Figure 5.1.2.7: Creation of demo dataset on Neo4J

Phase 4: Working with the actual dataset:
• Used the dataset of Caenorhabditis_elegans[3].
● After converting the tab limited file to a csv it looked like this:

Figure 5.1.2.8: Dataset being inserted into Neo4J

● For loading the file to neo4j, the load csv command was used as follows:
LOAD CSV WITH HEADERS FROM 'file:///Users/Lenovo/Downloads/Dataset/BIOGRID-ORGANISM-Caenorhabditis_elegans-3.4.134.csv' AS line LIMIT 1 RETURN line

● It returned the output as shown below:

Figure 5.1.2.9: Result after loading the dataset on Neo4J
The structure of the node that will be used is created using the following code:

On applying the same logic as above on the dataset that is being used, the following code was written:

```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM 'file:///Users/Lenovo/Downloads/Dataset/BIOGRID-ORGANISM-Caenorhabditis_elegans-3.4.134.csv' AS row
CREATE (:ID {companyName: row.IDInteractorA});
```

Figure 5.1.2.10: Result after creating the structure

```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM 'file:///Users/Lenovo/Downloads/Dataset/BIOGRID-ORGANISM-Caenorhabditis_elegans-3.4.134.csv' AS row
MERGE (nodeType:NodeType { productName: row.IDInteractorB}) ON CREATE SET nodeType.IDInteractorA = row.IDInteractorA;

CREATE INDEX ON :ID(companyName);
USING PERIODIC COMMIT
```
LOAD CSV WITH HEADERS FROM 'file:///Users/Lenovo/Downloads/Dataset/BIOGRID-ORGANISM-Caenorhabditis_elegans-3.4.134.csv' AS row
MATCH (nodeType:Node { productName: row.IDInteractorB })
MATCH (id:ID { companyName: row.IDInteractorA })
MERGE (id)-[:INTERACTSWITH]->(nodeType);

Figure 5.1.2.11: Result after creating the relationships

On querying using:
Match (n) return n limit 10

Figure 5.1.2.12: Result after querying the top ten nodes
5.2 Second Milestone

5.2.1 Working with the Neo4J API[4]
The Neo4J API was mainly used to create the data graph from the given dataset.
The initial algorithm that was used to insert nodes was:
1. Create the database
2. Insert the unique nodes
3. Shutdown the database

But this approach had the following issue:
The nodes and the relationships were being formed by the code in memory. But on checking the Neo4J database through localhost, the changes were not being reflected.

This issue was overcome with the help of the following two steps:

- Batchinserter
  When batch insertion was not used, the graph was being created in the memory but not on Neo4j. For the changes to be reflected, batch insertion was used[4]. It improves the performance by bypassing transactions.

- Path change
  The path to which the configuration files of the extracted binary code was pointing had to be changed.

  The location of the file is[4]:
  conf/neo4j-server.properties

  The following line was updated[4]:
  org.neo4j.server.database.location= /Users/Lenovo/Downloads/Project/Db

After overcoming this issue, on inserting a set of unique nodes (Harry Potter and James Potter), the following graph was obtained:
The following algorithm was then used to load the dataset in the form of unique nodes:

1. Extract edges from the dataset which are represented by the set of Interactor Ids in the same row.
2. The following points need to be noted before going to the next step:
   - In Neo4J, when a node or a relationship is created, a unique id is assigned to it.
   - In the algorithm, a hashmap is used to keep a track of the nodes that are being created. The node id which is assigned is stored as the key and the Interactor Id is stored as the value.
   - In the next step of the algorithm, the hashmap is checked for the presence of each of the nodes.
   - To form a relationship, node ids are used. In the scenario where one or both of the nodes are present in the hashmap, the key value which is the node id is extracted.
     - If the node is already present, the node id is the key which is extracted from the hashmap.
     - If the node is not present, it is created and the new node id is used.
3. The node ids obtained in the above step for each of the nodes is used to form the relationship between them which is in the form of an edge.

The result of the implementation of the above algorithm using the Neo4J java API gives the following graph:
The properties of the dataset that was used above are as follows:

- It consists of the roundworm (Caenorhabditis elegans) dataset.
- It has 8658 edges.
• The graph is created by using the interactor IDs of A and B.

The result obtained on running the above code had the following characteristics:

• 2916 unique nodes were being created.
• It took about 1 minute and 16 seconds to load.
• 8658 relationships were created which is equal to the number of rows in the dataset used.

5.2.2 Using Neo4J server with JDBC
The JDBC connection was mainly used to query the graph. The following code snippet shows how the JDBC connection was formed and used:

```java
GraphDatabaseService db = new GraphDatabaseFactory().newEmbeddedDatabase(new File("Path to the database location on the system"));
try (Transaction ignored = db.beginTx(); Result result = db.execute("Query being used"))
{
    while (result.hasNext()) {
        // Perform an action with the result
    }
}
```

5.3 Third Milestone

The main method of the java implementation performed the following actions in the same order:

1. Load the dataset in order to form the data graph.
2. Ask the input from the user through console to form the query graph.
3. Find the matching edges of the query graph from the data graph.
4. Use Levenshtein distance for the node labels entered by the user if required.
5. Use the cursors obtained in the previous step and store the path to the approximate subgraph.
6. Print the stored path as output

5.3.1 Cursor Initialization
• Steps 2, 3 and 4 in the above mentioned in the main method algorithm come under Cursor Initialization.
  ❖ Step 2: Ask the input from the user through console to form the query graph

The input taken from the user is in the form of edges.
Example:
A — (Interacts With) – B — (Interacts With) — C =
A — (Interacts With) – B + B — (Interacts With) — C

The algorithm used for this is:
1. Ask the user for relationships in the form of A --- InteractsWith --- B
2. Store each relationship in the hashmap with the input serial number as the key and the input nodes as the value.

Example: The input edges given by user are:

3. The value of K is 2.

Figure 5.3.1.1: Screenshot of input taken through console from user

- Step 3: Find the matching edges of the query graph from the data graph.

The algorithm used for this is:

1. Find all relationships between the nodes which match the edges of the query graph
   - The Cypher code used to obtain the result of this is given by:
     ```cypher
     MATCH p = (:Interactor{Name:'string1'})-[KNOWS*length]- (:Interactor{Name:'string2'}) RETURN p, length(p) LIMIT 10;
     ```
   - `string1` and `string2` represent the Interactor Ids or nodes between which the path is being queried.
• *Length* is varied from 0 to 5 in a for loop to obtain nodes which are within a 0 to 5 hop distance from each other.

2. A hashmap is used to store the list of cursors which are obtained as the resultset.

• The key of the hashmap is the serial number of the edge which had been assigned while storing the user inputs.

• The value of the hashmap is an arraylist. The arraylist consists of the resultset obtained after running the above query on the user input set of nodes.

• If the resultset does not have “k” values, Levenshtein distance is used to obtain the remaining values.

❖ **Step 4: Use Levenshtein distance for the node labels entered by the user if required.**

Levenshtein distance is defined as the minimum number of changes that need to be made so that the two strings being compared are equal.

The following algorithm is followed to obtain the levenshtein distance between the user input and the value present in the data graph:

1. Create a two dimensional array with of dimensions m+1 and n+1 where m and n are the lengths of both the strings.
2. The 0th row of the two dimensional array is filled with the column number and the 0th column is initialized with the row number.
3. While moving across each column in each of the rows, for a given position (i,j) the minimum values among the values at ((i,j-1)+1), ((i-1,j)+1) and ((i-1,j-1) + (0 if the values at (i-1,j-1) are equal and 1 if unequal)) is placed at (i,j).
4. The last value i.e. (m+1,n+1) gives the levenshtein distance between both the strings being compared.

In order to restrain the algorithm, a Levenshtein distance of less than 4 is considered for implementation.

5.3.2 Cursor access and solution building phase
This phase consists of the 5th and 6th step of the algorithm.

❖ **Step 5: Use the cursors obtained in the previous step and store the path to the approximate subgraph.**

The algorithm followed to achieve this is as follows:

1. Using the node ids of the nodes forming the edges stored in the Cursor initialization phase, a continuous path is formed. This path will approximately match the query graph.

2. For each of such paths obtained, the following criteria are used to place them in ascending order:
- The sum of the levenshtein distance of both the nodes.
- The number of hops between the nodes.
- The node ids which show the traversal required to get to the subgraph.

3. Based on the above criteria, the top – k values are chosen as the output.

- **Step 6: Print the stored path as output**

The Cypher query used to obtain this is:

MATCH p =(s)-[KNOWS*length]-(_)-(_)-[KNOWS*length]-(t) where ID(s)="start" and ID(t)="end" RETURN p, length(p) LIMIT 10;

Start and end represent the two nodes and length is represented using the number of hops. These values are obtained from the previous step.

All the obtained paths are combined together to form the subgraph.

### 5.3.3 Results

The following are two scenarios in which the solutions of the algorithm have been plotted:

The following is the representation of the nodes:

![Figure 5.3.3.1: Depiction of nodes](image)

**Scenario 1(Exact Matching):**

**Input:**

entrez gene/locuslink:179791 InteractsWith entrez gene/locuslink:178104
entrez gene/locuslink:178104 InteractsWith entrez gene/locuslink:175133
entrez gene/locuslink:175133 InteractsWith entrez gene/locuslink:174150
entrez gene/locuslink:174150 InteractsWith entrez gene/locuslink:179791
Figure 5.3.3.2: Query graph for exact matching

Output:
Solution 1:

Figure 5.3.3.3: Solution 1 for exact matching
Solution 2:

Figure 5.3.3.4: Solution 2 for exact matching

Description:
1. The scoring function is given as the sum of the number of hops, the levenshtein distance and the traversal to find it.
2. In solution 1, the path between 178104 and 179791 has 1 hop.
3. In solution 2, the path between 178104 and 179791 has 2 hops.
4. Hence solution 1 has a lesser scoring function.
Solution 3:

![Figure 5.3.3.5: Solution 3 for exact matching](image)

**Description:**

1. In solution 1 and 2, the number of nodes in between 175133 and 174150 is 0.
2. In solution 3, the number of nodes in between 175133 and 174150 is 2.
3. Hence solution 3 has more scoring function.

**Scenario 2 (Approximate Matching):**

**Input:**
- `entrez gene/locuslink:75604 InteractsWith entrez gene/locuslink:174506`
- `entrez gene/locuslink:174506 InteractsWith entrez gene/locuslink:17770`
- `entrez gene/locuslink:174506 InteractsWith entrez gene/locuslink:174504`
- `entrez gene/locuslink:175604 InteractsWith entrez gene/locuslink:177720`
Output:

Solution 1:
Description:

1. Approximate matching scenario took 45 minutes and 20 seconds to run.
2. The nodes in the solution are the ones obtained after running the algorithm and using Levenshtein distance.

Solution 2:

![Solution Diagram](image)

Figure 5.3.3.8: Solution 2 for approximate matching

Description:

1. The number of nodes between 177720 and 174504 is 2 in solution 1.
2. The number of nodes between 177720 and 174504 is 3 in solution 2. Hence solution 1 has a lesser scoring function.
Solution 3:

Description:
1. The number of nodes between 175604 and 174506 is 2 in solution 2.
2. The number of nodes between 175604 and 174506 is 3 in solution 3. Hence solution 2 has a lesser scoring function.

6 Conclusion
The GeX Top-K Query algorithm is very accurate. After implementing the algorithm, it was observed that the scoring function gave efficient order of the matching subgraphs. But as the size of the dataset increases, the time taken to compute the result increases considerably. The algorithm uses the entire dataset for each query which is computationally intensive. Future work can include the approximation of the edge labels in scenarios where the edge labels vary.

7 Bibliography


4) The Neo4J Manual v2.3.3, 2016-03-22 20:17:07