Implementing GEX- A Framework for approximating complex queries on graph-modeled data

Vishwas Tantry (vxt1640@rit.edu)
Advisor: Dr. Carlos Rivero
Contents

1. Abstract ............... 2
2. Introduction ............. 2
3. Pseudo Code ............. 4
4. Milestones ............... 5
5. Detailed Algorithm Analysis .... 6
6. Implementation Overview .... 19
7. Results and Analysis ....... 20
7.1 Testing on Yeast Dataset .... 36
8. Challenges .............. 37
9. Future Work ............ 38
10. References .......... 38
1. Abstract

In this information age, terabytes of data are processed per second. The challenge of the modern age is to find effective ways to store and represent this data. Storage and representation is just half of the work done, the other half is finding effective ways to extract knowledge out of this stored data corpus.

The advent of social networking websites like Facebook and Twitter, biological and chemical databases and information from the web have resulted in new types of data that has to be represented in an unique way such as Graph[1]. It is very difficult to store such data in an relational database because of the complex connections between various objects in the data. Consider an example of Facebook, a person has many friends who in turn have multiple friends. They like various pages and photos of their friends. It would be so difficult to track so many connections in a relational database resulting in redundancy of information. Graph database models the data in a way that allows to retain the complex relationships that exist between objects in the data and maintains the inherent structure of the data, thus allowing for an effective querying strategy. Graph databases can use efficient graph algorithms to retrieve patterns and solutions to the queries.

2. Introduction

Graph databases cannot take advantage of the structured query language (SQL) because it warrants knowledge of exact location of the objects in the graph and the complex relationships that exists between them makes it extremely difficult. Just storing the data in graph data structure is half work done, the other half is extracting the data efficiently. Extraction of the data as the database size increases becomes extremely difficult. In this project I am implementing Gex, a framework for approximating matching of complex queries on graph data and further evaluate it with other graph algorithms. Gex makes it possible to search large databases having minimal graph structure very efficiently. It is more than just a keyword search and hence it is more efficient than other keyword search graph algorithms.

Therefore two well known techniques have been proposed for querying graph databases, keyword based query model and graph based approach. In keyword based approach only the
keywords are searched in the graph with no knowledge of the inherent structure needed. This technique is useful in some scenarios where multiple schemas co-exist but is very difficult to formulate queries with correct semantics which will result in exact answers. Graph based techniques approximates edges by using the actual topological paths between entities in the graph rather than understanding the semantic nature of the edges.\textsuperscript{1}

In this project, I will be implementing a graph based technique called Gex \textsuperscript{1} in Java and comparing it with a technique called Nema.\textsuperscript{2} Gex is superior to most of the previous techniques because of the following reasons,\textsuperscript{1}

1. It is very generic in nature and is useful for any graph based datasets.

2. It introduces an expressive query graph based query language that can be useful to formulate queries which takes into consideration the relationship between the data, the nature of the relationship between two entities, any piece of missing information for both data and relationship between the entities and constraints on the data.

3. It also considers the semantic nature of the relationship between two entities i.e the how strong or weak two entities are related to each other.

4. It uses a score function to evaluate all the retrieved results and select the best amongst them.

5. The scoring function is further used in Top k retrieval algorithm which limits the results of the query to only k best results thus making it very efficient comparing to previous algorithms.

I will be implementing the algorithm from the paper in Java.\textsuperscript{1} Further I would be evaluating the results of the queries of the aforesaid implemented algorithm using the Neo4j database.

I will conclude my project by comparing the results of the Gex with NeMa and the types of queries that GeX supports but not NeMa and vice-versa.

\textsuperscript{1}
3. Pseudo Code

**Top- k Algorithm**

```java
for(int i=1 to i=|total number of edges in the query(E)|)
    ....... (label start node, edge label, end label) <- I_X-L
    ......... List L_i <- I_X-SLRS or I_X-SRLV((label start node, edge label, end label)
    ......... C_i <- newCursor(List L_i).
int i=-1, Ans = {}
while(i->nextCursor(i) >0) {
    ...........(node1, node 2, cost(virtual edge between node 1 and node 2)) <- C_i getNextElement()
    ............computeAnswer(i,node1,node2,cost(node1,node2),Currentsolutioncost=0,nodeList=null,
    edgeList=null,Ans=null)
    .............LowerBound<- \sum_{j=0}^{E} C_j.peekCost()
    .............If (\|Ans\| >= k and LowerBound >= Ans[k].score ) {
        ................. Abort answer computation
    .................}
output Ans

**Compute Answer Algorithm**

ComputeAnswers(i,node1,node2,cost(node1,node2),Currentsolutioncost,nodeList,edgeList,Ans)
NodeList[index_node]=node1
NodeList[index_node2]=node2
edgeList.update(node1,node2,i)....
Currentsolutioncost =+Currentsolutioncost + cost
j<- getNextEdge_QUERY(i)
If (j < 0 ) {
    ............ if( Ans[k].score >CurrentSolutioncost OR \|Ans\| <k )  {
        .........................Ans.add(nodeList,edgeList,currentsolutioncost)
    .................return
    .................}
if (\|Ans\|> k and currentsolutionCost + C_j.peekCost() > Ans[k].score ) {  
    ................. Abort answer calculation
    .................return
    }  else {
    .................while((node1',node2',cost(node1',node2')<-C_j.seek(EdgeList.getNextEdgeNode
    start,EdgeList.getNextEdgeNode end) != NULL) {  
        .................computeAnswers(j,node1',node2',cost(node1',node2'),currentSolutionCost,nodeList,
        edgeList,Ans)
    .................return
    .................}
```
GetNextCursor Algorithm

```java
int getNextCursor(int current) {
    int next = -1
    if (current == (queryedges.length - 1)) {
        current = -1
    }
    for(int i=(current+1);i<queryedges.length;i++) {
        if(c[i].size()>0){
            next=current+1
            return next
        }
    }
    return next
}
```

4. Milestones

- Milestone #1: Understanding of the core algorithm and the framework from the paper.
- Milestone #2: Implementing the algorithm in Java using the framework in the paper.
- Milestone #3: Implementing the database using Neo4j and testing the algorithm using real world dataset.
- Milestone #4: Evaluating Gex with other algorithms and interpretation of the results.
5. Detailed Algorithm Analysis

Understanding the GeX algorithm

Similarity function($sim$):
The GeX algorithm models tries to give approximate query results by performing two kinds of approximation, label mismatches and structural mismatches. GeX handles label mismatches by using a similarity function. For the ease of understanding I will represent similarity function as $sim(a,b)$ where $a$ and $b$ are labels of the two nodes or labels respectively. The sim function will always give output in the range of 0 to 1.0 means that the two words are an exact match while 1 means the two words are total mismatch. The paper does not mention any algorithm for calculating this similar function since the calculation cannot be generic. It depends on the domain of the dataset. I will be using Jaccard distance or Levenshtein distance based on the dataset I will ultimately use for testing purpose. The similarity function defined above is used in many places in algorithm later.

Path $p(e)$:
Path $p$ for an edge is just a sequence of the nodes obtained by traversing from the source node to the end node of the edge. For example, consider a graph as shown in the figure.1.

![Graph Diagram]

The path from A to C is $p=\{e_1, e_2\}$

Semantic Relatedness:
This is one of the important concepts of the GeX approach for graphs. Semantic relatedness is a set which contains nodes from the graph which are related to each other meaningfully. Semantic relatedness follows two important properties,
i. All the edges of the graph are a subset of the semantic relatedness function. This property is referred to as graph containment property.
ii. If there is an edge between two nodes n1 and n2 and the path between the two nodes n1 and n2 contains another node n3 in between nodes n1 and n2. Suppose also n1 and n3 are semantically related and are a part of semantic relatedness set. Then by the path decomposition property nodes n3 and n2 are also a part of semantic relatedness set with its edge also included in the semantic relatedness set.
This means that the nodes are semantically related to each other which is calculated using the Sim function defined above. Semantic relatedness also contains the edges that connect these two nodes. The edges are labeled accordingly from the set of concept labels. The edge is “virtual” if there is no actual path between them in the graph. In that case the virtual edge is approximated by the path p defined above in the actual graph. If it happens that the virtual edge is present in the graph then the path p between the two edges is same as the virtual edge in Semantic relatedness set. Therefore for such an edge the cost of approximating the virtual edge to the actual edge is 0. This gives us an introduction to the cost function.

Cost function c():
The cost function c is used for quantifying the approximation of a path. The cost function c is used to indicate the cost of approximating an virtual edge from the semantic relatedness set to actual path( p) in the graph. For example, the cost c for a virtual edge is 0 if the edge is present in the actual graph. The cost function c can be further explained by the below figure 2.

![Diagram](image)

Suppose there is an virtual edge between A and D. In actual graph this virtual edge between A and D is approximated by the actual path p between A and D which is p={e1,e2,e3}. Since there are two nodes in between the A and D the cost is 2. Therefore the cost of the edge between A and D is 2. If there was a direct edge present between A and D, then the cost would have been 0 since there was no approximation needed.

The GeX approach also has an enhanced version of the semantic relatedness set which tries to remove redundant information that may have crept in the semantic relatedness set. There may be set of two nodes that are present in the semantic relatedness set multiple times with same edge label but different approximating path. The different approximating path will have different cost. In order to improve efficiency of the algorithm only the less expensive version is maintained in the reduced semantic relatedness set.
Consider the above figure 3., this is the less expensive version of the node pair A and D since the cost is just 1. Assume the edge label between the A and D is also same as the one in the figure 2. This version of the node pairs will be given preference compared to the previous version in figure 2 which has cost 2. Since the reduced version of the semantically related set has less expensive versions of the node edge pairs the sub graphs generated will be more efficient and accurate.

Understanding SR set:
Semantic relatedness is a set which contains nodes from the graph which are related to each other meaningfully. Semantic relatedness follows two important properties,

i. All the edges of the graph are a subset of the semantic relatedness function. This property is referred to as graph containment property.

ii. If there is an edge between two nodes n1 and n2 and the path between the two nodes n1 and n2 contains another node n3 in between nodes n1 and n2. Suppose also n1 and n3 are semantically related and are a part of semantic relatedness set. Then by the path decomposition property nodes n3 and n2 are also a part of semantic relatedness set with its edge also included in the semantic relatedness set.

The edges are labeled accordingly from the set of concept labels. The edge is "virtual" if there is no actual path between them in the graph. In that case the virtual edge is approximated by the path p defined above in the actual graph. If it happens that the virtual edge is present in the graph then the path p between the two edges is same as the virtual edge in Semantic relatedness set. Therefore for such an edge the cost of approximating the virtual edge to the actual edge is 0. This gives us an introduction to the cost function.

The SR is constructed first by inserting all the edges from the data graph into the SR set. Then the following rules are used to label the edges in the SR.

SR uses the type information of the edge to label the edges in the SR. The type information is used in the following way:
If type of the edge in the data graph is subclass then the edge label in the SR is labeled as isA.
If type of the edge in the data graph is type then the edge label in the SR is labeled as type.
If type of the edge in the data graph is property then the edge label in the SR is labeled same as in the data graph.
If type of the edge in the data graph is ispartof then the edge label in the SR is labeled as ispartof.
If type of the edge in the data is domrel is subclass then the edge label in the SR is labeled same as in the data graph.

Then iteratively virtual edges are added. Virtual edges are added using many properties but they are very specific to the example given in the paper. It is not generic and very complex to understand. So I have decide to use my own rules for adding virtual edge.

For all edges e = (x, y), e' = (y, z) ∈ SR:
If type edge e and edge e' is isA OR
if the type of edge e is isA and type of edge e' is type OR
if the type of edges e and e' is ispartof then

type of edge e''(x,z) is type of edge e and label of edge e'' is same as label of edge e.

The virtual edges thus added in this way and some more rules that I am working on would be used to create SR. The edges in SR will be used to create Index I_{X-SRLS} that would contain all the edges from the SR. But to increase the efficiency of the search the edges would be grouped based on the three labels of node pairs and corresponding edge. For example, $W,director,$X would contain all the edge having label director that connect any two nodes. In this way all the edges are grouped having similar patterns. Any new query edge having similar pattern would be searched through this index I_{X-SRLS}. Hence the output list would contain only the edges that approximately matches the query edges. In this way only the relevant results are given as solution edges rather than burdening the user with irrelevant results.

Scenarios where GeX would fail
If for example we have a query which has only node information but no corresponding edge information then in that case the GeX will try to get the approximately matching edge using the information which is available. Suppose the query edge is

```
Clint
  ?
  
MDB
```

In this case the algorithm would search the I_{X-SRLS} with the following information,
Clint, $X, MDB

The data graph edges with this pattern will be given out as results. It will also include the edge Clint, actor, MDB which will have the highest rank since it is a direct edge with cost 0. If there is a scenario where all three labels are a variable then in that case it can be matched with all edges present in the SR and the computations will increase manifold. In Gex it is a challenge to get exact labels of the query edge and node. Therefore techniques like Edit distance, Apache Lucene or Jaccard distance should be used. It is very vital to get the correct labels to reduce search space and also search the relevant list from the I_XSRLS^.

**Scoring function S()**

The scoring function is used to rate the accuracy of the solution to the query. It consists of three parts,

i. The first part calculates how similar the query nodes are with respect to the data nodes. The sim function defined above calculates the approximation required to match query nodes with data nodes. It is calculated using the following formula,

$$S1 = \left( a_c / \text{Total number of query nodes} \right) \sum_{n \in N_q} \text{sim}(n, f(n))$$

where n is the query node label and f(n) is the approximated data node label corresponding to the query node. $a_c$ is some constant.

ii. The second part calculates the approximation between the query edge labels and corresponding graph edge labels.

$$S2 = \left( b_e / 2 \times \text{Total number of query edges} \right) \sum_{q \in E_q} \text{sim}(e, g(e)) + c(e)/MC$$

where e is the query edge label and g(e) is the approximated query label corresponding to the query edge. $b_e$ is some constant and it also takes into consideration the cost of the path which is defined above. Mc is a normalizing constant.

iii. The third component calculates the cost of the condition in any node or edge variables.

$$S3 = \left( c_c / \text{Total number of conditions} \right) \sum_{c \in C} (1 - s(c))$$

The addition of all three parts is the total score function that is used to calculate score a function.

$$S = s1 + s2 + s3.$$
The higher the score, higher is the approximation and therefore it will be less accurate solution. Lower the score, less is the approximation, higher the accuracy of the solution.

**Top-k Algorithm Pseudo code**

- First we need to generate the query.
- The parser will extract import concepts from the query and will have the list of nodes and edges with their approximate labels.
- Then iterate through all the edges in the graph, for all edges from the query, iterate through it i.e

  \[
  \text{for(int } i=1 \text{ to } i=|\text{total number of edges in the query}|) \]

  - Based on the edge label and the node labels that are connected by current edge search for the corresponding node and edge labels from the reduced SR list.

\[
\text{(label start node, edge label, end label)} \leftarrow \text{I}_X \text{-L}
\]

- The paper introduces an inverted B+ tree Index (I\_X-\_L) containing all the node and edge labels in the reduced SR set for the sake of efficiency. It makes the search and retrieval of the labels very efficient from the reduced SR set which otherwise would be very inefficient if you consider a large database.

  Path pointer List L\_i \leftarrow \text{I}_X \text{-SLRS or I}_X \text{-SRLV}(\text{(label start node, edge label, end label)})

- Once you get the labels for the node and the edge, search the Inverted Index I\_X-\_SLRS which is another data structure built on top of reduced SR to get quick access to path pointer lists.

- If one of the nodes or edge has a value constraint then the search proceeds in I\_X-SRLV after getting the List number from I\_X-\_SLRS. This is another inverted index created on top of reduced SR to get accurate list elements more efficiently and quickly using the List number and value.

\[
\text{C}_i \leftarrow \text{newCursor(Path pointer List L\_i)}.........................................................(1)
\]

- Once the appropriate list is retrieved the next step is to allot a cursor to it. You can think cursor as an iterator over a stack or queue. The list is assigned to the cursor and the cursor iterates through this list of paths by using various cursor functions like next(), seek(), peek().

The for loop ends here, till this moment every edge will have an appropriate cursor attached to it which iterates over the relevant path pointer list based on the nodes and edge labels of the query edge.

**int i=0, Ans = null**

Initializing the variable i=0 and answer set to be null, I will be used to iterate through the cursors defined above for every edge. Ans would contain all the node lists and edge list of the each of the Top k solutions.
while(i->nextCursor(i) > 0) {

- Now the next step is to get solutions from the cursors by iterating through every cursor defined above. The while loop runs until all the cursors have been visited.
- The nextCursor() takes in the current cursor number and returns a new cursor. Since the cursor starts from 1, when we pass 0 for the first time we can get any cursor from C₁ to Cₑ. The strategy for selecting next cursor could be anything which left for the user. We could use round robin strategy or a random strategy.

(node1, node 2, cost(virtual edge between node 1 and node 2)) <- Cᵢ.getNextElement()

- The cursor selected from the previous step will extract it’s elements from the List which was initialized in the initialization step in (1)
- The extracted elements are the node pairs and the cost of the virtual edge between them.
- For easy representation cost(virtual edge between node 1 and node 2) can be represented as cost(node1,node2)

computeAnswer(i,node1,node2,cost(node1,node2),Currentsolutioncost=0,nodeList=null,edge;list=null,Ans=null)

- The cursor elements along with the Currentsolutioncost ,nodeList,edgeList and Ans are passed to the computeAnswer() function which calculates total cost of the solution where first edge of the solution is the edge between node1 and node2. Inside the function the other edges of the solution along with corresponding nodes are calculated.
- The function is recursive and is called until the complete solution is computed which means the function is called each time for the query edge.
- The first time the function is called the values for the nodeList,edgeList and the final Ans is null. Also the Currentsolutioncost is zero.
- LowerBound<- summation of next edge element in each of the cursor
If (there are K answer until now and LowerBound >= Ans[k].score ) then
    Abort answer computation
output Ans
}

The elements in the cursor are sorted based on their cost. The Ans is also sorted based on the cost of the solutions. Now if there are already K answers then the threshold would be the cost of Kth solution in Ans.
For any solution to be accepted as the top K solution it should have cost at less than or equal to the Kth score. And next best solution for the edges possible would be the LowerBound. Therefore if LowerBound is greater, the algorithm aborts and Ans is returned as the final solution which contains top K solutions. The outer while loop runs until the the function nextCursor() returns -1.
The function returns -1 only when all the elements in each cursor are iterated over by the cursor.

**Answer Computation Algorithm**

`ComputeAnswers(i,node1,node2,cost(node1,node2),Currentsolutioncost,nodeList,edgeList,Ans)`

- this function calculates the remaining edges of the current solution. The current solution has two nodes until now. Therefore they are inserted into the nodeList.
  
  `NodeList[index_{node1}]=node1`

- `NodeList[index_{node2}]=node2`

- here `index_{node1}` and `index_{node2}` are the respective id’s of the nodes.

**edgeList.update(node1,node2,i)...**

Then the edgeList which is used to store the edges of the solution is initialized with the first edge of the solution. This edge is the edge connecting the two initial solution nodes, node1 and node2. `i` is used to store the edge in the correct index.

The update function also updates edgeList with adjacent edges.

For example, edgeList after updating edge between n1 and n2, the edgeList will also add entries for edges between n2 and n3 and n1 and n4.

![Diagram](https://via.placeholder.com/150)

**Currentsolutioncost +=Currentsolutioncost + cost**

This initializes the Currentsolutioncost with `cost(node1,node2)` i.e cost of the first node pairs added to the solution, and when the function `computeAnswers()` is recursively called the Currentsolutioncost gets updated with the cost of the latest edge added to the solution.
j<- getNextEdge_QUERY(l)
If (j < 0 )
   if( Ans[k].score >CurrentSolutionCost OR Total number of computed answers till now is < k )
      Add the current solution to the Ans.add(nodeList,edgeList,currentsolutioncost)
return

This step is used to get the next query edge from the query graph. If the all the query edges have been visited then computation of the current solution is stopped. Since the answers computed are in the increasing order of cost, the Kth answer would be the threshold for a new answer to be included into the Ans. If the number of computed answers is less than the k, then the current solution is included in the Ans. The Ans includes the node list of the solution along with cost and edge list.

If total number of answer calculated is greater than k and current solution is no good then abort the answer calculation. else find out the next edge element from the current cursor. If it’s null then it will return else it will recursively call the computeAnswer function to get next set of edges of the solution. Here the Cursor.seek takes in the node pairs and searches the List associated with node the calling cursor with node pairs as values. This is done using $I_{X,SRN}$ which is used for searching for an cursor element faster in an list using the node pairs as a value. It is similar to $I_{X,SLRV}$ which instead uses a label value.

If (Total number of computed answers till now is > k and currentsolutionCost + cost of next edge that will be included is > Ans[k].score )
   Abort answer calculation
   return
else
   while((node1', node2' , cost(virtual edge between node1’ and node2’)) <- C.j.seek(EdgeList.getNextEdgeNode_start,EdgeList.getNextEdgeNode_end) != NULL) {
      computeAnswers(j,node1’,node2’,cost(virtual edge between node1’ and node2’),currentSolutionCost,nodeList,edgeList,Ans)
   }
return

Next cursor algorithm

The next cursor algorithm simply returns a value between 1 to |E| where |E| is the number of query edges. Based on the strategy used, it returns appropriate value between 1 to |E|. One condition that needs to be satisfied before returning the value is that the cursor size of the
corresponding edge should have be more than 0. That means at least one element should be still available in the list that is still not accessed by the algorithm.

Example:
Consider the Movie database modelled as a graph. The figure below is a snapshot of the database which describes about the movie “Million Dollar Baby”. It has information about its Actors and Director. The property boxes for every entity describes the entity. For example the entity Hilary Swank is described by its properties such as year of her birth, name and place of birth.

Now consider the following query,
Find the Name of the actor who acted in movie directed by Clint Eastwood?
The query can be represented as a graph as shown in the below figure.
Now the first step is to store all the node and edge labels of the data graph in data structure. The paper recommends storing the node and edge label information in a database. For accessing the labels of node and edges very quickly they are stored in inverted index.

The main feature of Gex is semantic relatedness set (SR) which is described above in detail. The SR is constructed using multiple data structures. \( L_{x,SLRS} \) is an inverted index which stores cluster information and assigns the list of paths that belong to that cluster.

The edge which belong to SR are first clustered based on the structural summary only. They are clustered based on the value triple, node values and edge corresponding them. Here in our case the clusters would be,

\[
@a,L_{\text{DIRECTOR}},@b \\
@a,L_{\text{ACTOR}},@b \\
@a,L_{\text{born}},@b \\
@a,L_{\text{year}},@b
\]

where \( @a, @b \) are variables that represent any nodes. This can be interpreted as any two nodes that are connected by a director edge. Similarly the second cluster will have all the nodes that are connected by an actor edge. This makes the storage of the SR very efficient as repetitive patterns are clustered together in a single inverted index.

\( @a,L_{\text{DIRECTOR}},@b \) \( L_1 \) ----< the edge between million dollar baby and Clint Eastwood is the first edge that will feature in this list. The list is sorted based on the cost \( c \). The above mentioned edge has 0 cost since no approximation is needed. The virtual edge is same as the actual edge. We represent this edge as \( \text{em} \).

Similarly we can have many combinations with greater costs in increasing order.

For the query we have 3 edges, \( e_1, e_2 \) and \( e_3 \).

We want \( K=1 \),

So the for ( int i= 1 to 3)

for the first edge \( e_1 \) we have labels, (ClintEastwood, \( L_{\text{DIRECTOR}}, y \)).
Using this information we have to query the inverted index $I_{X_{SLRS}}$, we get a list with elements of the edges. This matches the first pattern therefore we get $L_1$. This listed is then initialized to a cursor. We call it $C_1$. Similarly for edge $e_2$ and $e_3$ we get two more cursors $C_2$ and $C_3$ after two more iterations of the for loop. $C_2$ gets initialized with $L_2$ having edge $en$ that represents edge between Hillary Swank and Clint Eastwood. $C_3$ gets initialized with list $L_3$ with multiple edges. This is because the pattern $@[a,L_{NAME},GLNAME]@b$ is satisfied by edge between Hillary Swank and her name property Hillary swank (el) and edge between Clint Eastwood and her name property Clint Eastwood(ep).

This ends the cursor initialization phase. The next is the Cursor access and solution building phase.

The nextCursor gives the output as 1 since it was 0 passed as the argument to the function and it returns 1 based on the round robin mode.

c1.next will iterate through the list $L_1$ and would return the first element that is edge $em$.

Computeanswer would then take in arguments $i=1$, $node1=clinteastwood$, $node2=milliondollarbaby$, $cost=0$, $currentsolutioncost=0$, $nodeList$, $edgeList$, $Ans$)

The nodeList gets populated with clinteastwood and milliondollar baby and edgeList with $em$. Also edgeList will store next edge information which is Hillary Swank.

Currentsolutioncost will get updated to 0 since the cost of edge $em$ was 0.

Then the nextQueryEdge would return 2.

Since there $2 > 0$ and since $|Ans|<1$, the control enters the while loop where a $C_2.seek(milliondollarbaby,hillaryswank)$ would give edge $en$.

The values are sent recursively to computeAnswers but this time the arguments would be $i=2$, $node1'=milliondollarbaby$, $node2'=hillaryswank$, $cost=0$, $currentsolutioncost=0$, $nodeList$, $edgeList$, $Ans$.

Again inside the computeAnswer function the nodeList gets updated with hillaryswank and edgeList with $en$. Also edgeList will store next edge information which is name property Hillary Swank.

The procedure again updates until nodeList updates with name property HillarySwank and edgeList with $el$.

The nextQueryEdge would return 0 since there are no more edges.

The Ans would update with the nodeList, edgeList and cost which is zero.

The function returns top -K algorithm. Now the first solution is ready. $LBound$ is used to indicate whether next set of edge in cursor are good enough for the answers. If it is greater than the K answer it stops the answer computation process.

The output is Ans with nodeList ,edgeList and cost.
**Scoring function of Gex**

The scoring function for Gex is defined as,

\[ S1 = \left( \frac{a_c}{\text{Total number of query nodes}} \right) \sum_{n \in N_q} \text{sim}(n, f(n)) \] ....where \( n \) is the query node label and \( f(n) \) is the approximated data node label corresponding to the query node. \( a_c \) is some constant.

ii. The second part calculates the approximation between the query edge and corresponding graph edge.

\[ S2 = \left( \frac{b_c}{2} \times \text{Total number of query edges} \right) \sum_{e \in E_q} \text{sim}(e, g(e)) \] ....where \( e \) is the query edge and \( g(e) \) is the approximated query edge which represents the solution edge.

iii. The third component calculates the cost of the condition in any node or edge variables.

\[ S3 = \left( \frac{c_c}{\text{Total number of conditions}} \right) \sum_{c \in C} (1 - s(c)) \]

The addition of all three parts is the total score function that is used to calculate score a function.

\[ S = S1 + S2 + S3. \]

The higher the score, higher is the approximation and therefore it will be less accurate solution. Lower the score, less is the approximation, higher the accuracy of the solution.
6. Implementation Overview

I have implemented Gex algorithm in the framework provided by Dr. Carlos. The main challenge was to identify and understand the necessary section from the whole framework. After understanding and identifying the relevant sections of the framework I started making necessary changes to the framework to implement GeX. I have made following changes/modifications to your framework to integrate the GeX framework.

Following are the list of additions to the framework:

- I have added a new package `edu.rit.matching.algorithms.gex`.
- The package consists of all the source code of GeX algorithm. It also consists of helper files that are needed to run the GeX algorithm.
- For testing purposes I have added a new `.java` file called `ApproximateMatchingGexTest.java` inside the `edu.uidaho.matching.test.approximate package`.
- The test file is used to provide the K value and location of data and query file.
- I have also modified the `ApproximateMatchingThread.java` file to include GeX algorithm.
7. Testing and Evaluation

I first tested my algorithm and compared with Nema for various small datasets.

Example 1
Nema Output: (K=1) and min sigma=0.2
{0=0, 1=1, 2=2, 3=3, 4=4, 5=5, 6=6, 7=-1, 8=-1, 9=-1, 10=-1, 11=-1} -- 6.259459459459459

Gex Solution: (K=1 and min sigma =0.2)
{0=5, 1=1, 2=2, 3=3, 4=4, 5=16, 6=17, 7=18, 8=20, 9=21, 10=19, 11=22} -- 1.296969696969696

Nema fails to give any answer if min sigma is greater than 0.1, Gex correctly answers for any value less than 0.3.
Example 8
Data graph:

```
Nema Output:
{0=0, 1=1, 2=2}--0.0

Gex Output:
{0=0, 1=4, 2=5}--1.4499999999999997
{0=0, 1=1, 2=2}--1.4499999999999997
```

Given the data graph and the query graph above there are two solutions which are possible but Nema gives only one solution because of the random seed point it selects in the initial stages. Gex gives both the answers and their scores are identical. The Gex in this scenario extracts correctly all the possible answers and gives correct scores to it.

Example 7
Data Graph:

```
Query Graph:
```

```
Nema solution:
\{0=0, 1=1, 2=2, 3=3, 4=4, 5=5, 6=6\} \approx 6.6060606060606055

Gex solution:
\{0=0, 1=1, 2=2, 3=3, 4=4, 5=5, 6=6\} \approx 1.5928571428571427

Here both Nema and Gex gives correct solution.

**Example 6:**

Query Graph:

Data Graph:

Nema Solution:
Nema and Gex both do not provide correct solutions. Nema gives -1 for two nodes which is incorrect since the nodes in the data graph start from 0. Gex does not provide any solution.

Gex fails in this scenario where the number of query nodes are greater than the data nodes. Gex fails because in the cursor initialization phase new candidate edges are created by Cross-product of the matching nodes from the search space of the nodes. In this scenario there will be no edges created because there is not a single node from the data graph which is matching query node B. The search space for node B is empty and hence no candidate edges are generated for the query.

Example 3:

Data

Nema solution:
{0=0, 1=1, 2=1, 3=1, 4=1}--0.0

Gex solution:
{0=1, 1=1, 2=1, 3=1, 4=1, 5=2, 6=3}--0.0
{0=3, 1=3, 2=8, 3=4, 4=4}--1.5
{0=5, 1=1, 2=3, 3=0, 4=2}--1.5
{0=6, 1=2, 2=3, 3=1, 4=8}--1.5
{0=1, 1=2, 2=8, 3=0, 4=3}--1.5499999999999998
{0=2, 1=2, 2=1, 3=8, 4=6}--1.5499999999999998
{0=2, 1=2, 2=1, 3=8, 4=0}--1.5499999999999998
{0=2, 1=3, 2=6, 3=8, 4=1}--1.5499999999999998
{0=2, 1=2, 2=3, 3=6, 4=7}--1.5499999999999998
{0=2, 1=2, 2=3, 3=1, 4=4}--1.5499999999999998
{0=2, 1=2, 2=3, 3=6, 4=1}--1.5499999999999998
{0=2, 1=2, 2=5, 3=1, 4=5}--1.5499999999999998
{0=3, 1=3, 2=8, 3=7, 4=8}--1.5499999999999998
{0=3, 1=3, 2=7, 3=4, 4=4}--1.5499999999999998
{0=6, 1=6, 2=2, 3=1, 4=1}--1.5499999999999998
{0=8, 1=3, 2=6, 3=7, 4=1}--1.5499999999999998
{0=0, 1=2, 2=3, 3=8, 4=8}--1.5499999999999998
{0=1, 1=1, 2=0, 3=3, 4=2}--1.5499999999999998
Nema Solution:
{0=0, 1=1, 2=1, 3=1, 4=1}--0.0
{0=8, 1=3, 2=2, 3=2, 4=2}--0.0
{0=7, 1=3, 2=2, 3=2, 4=2}--0.0
{0=6, 1=1, 2=0, 3=0, 4=0}--0.0
{0=5, 1=1, 2=0, 3=0, 4=0}--0.0
{0=4, 1=3, 2=2, 3=2, 4=2}--0.0
{0=3, 1=2, 2=1, 3=1, 4=1}--0.0
{0=2, 1=1, 2=0, 3=0, 4=0}--0.0
{0=1, 1=0, 2=0, 3=0, 4=0}--0.0

Gex solution:
{0=0, 1=1, 2=0, 3=0, 4=1}--1.45
{0=0, 1=0, 2=1, 3=1, 4=0}--1.45
{0=1, 1=0, 2=1, 3=1, 4=0}--1.45
{0=1, 1=1, 2=0, 3=0, 4=1}--1.45
{0=1, 1=2, 2=1, 3=1, 4=2}--1.45
{0=1, 1=1, 2=2, 3=2, 4=1}--1.45
{0=1, 1=5, 2=1, 3=1, 4=5}--1.45
{0=1, 1=1, 2=5, 3=5, 4=1}--1.45
{0=1, 1=6, 2=1, 3=1, 4=6}--1.45
{0=1, 1=1, 2=6, 3=6, 4=1}--1.45

Nema again fails when you don’t specify any label to any node. Nema has this problem even when all the nodes have same labels. So it was obvious that it will fail in this case as well. Gex as in the case of same labels for nodes performs with some false positives.

Query 2:
Nema solution:
\{0=0, 1=1, 2=1, 3=1, 4=1\}--7.5
\{0=8, 1=3, 2=2, 3=2, 4=2\}--7.5
\{0=7, 1=3, 2=2, 3=2, 4=2\}--7.5
\{0=6, 1=1, 2=0, 3=0, 4=0\}--7.5
\{0=5, 1=1, 2=0, 3=0, 4=0\}--7.5
\{0=4, 1=3, 2=2, 3=2, 4=2\}--7.5
\{0=3, 1=2, 2=1, 3=1, 4=1\}--7.5
\{0=2, 1=1, 2=0, 3=0, 4=0\}--7.5
\{0=1, 1=0, 2=0, 3=0, 4=0\}--7.5
Here Nema fails as it gives the same data node for two different query nodes. Here since the query nodes have same labels Nema fails here because it gives same edges for two different query edges.

Gex solution:
\{0=0, 1=1, 2=2, 3=3, 4=4\}--0.8500000000000001

Gex does give correct answers here. The update function is not properly defined in the Gex paper and hence the exact implementation of the Gex update function cannot be confirmed from the paper. But I tried to implement update function but it seems it gives few false positives along with correct solutions.
My implementation first finds a solution edge for the first query edge. Using this solution edge it tries to get all the solution edge which is related to the solution edges which are already present in the solution. This creates problems similar to what we have in the Nema. Therefore I have included a check which ensures that the solution edge is not selected twice and hence avoids problems similar to that Nema. The overhead with this solution is that time complexity increases because we need to check every solution edge already present in the solution every time we need to add a new solution edge.

**Example 4:**

Data Graph:  
Query Graph:  

Nema solution:
{0=0, 1=-1}--1.1333333333333333

Gex solution:  
{0=0, 1=5}--1.85

Nema gives incorrect results .Gex on the other hand gives correct solution. Since Gex looks ahead for its solution any number of hops Gex is able to find to an approximate solution. On the contrary Nema does not look ahead more than two hops hence it gives incorrect solution.

**Example 2:**
Nema solution:
\{0=0, 1=2, 2=2, 3=5\}--8.38716577540107

Gex solution:
\{0=0, 1=1, 2=2, 3=5\}--1.0833333333333333
\{0=0, 1=2, 2=3, 3=5\}--1.0833333333333333
\{0=0, 1=1, 2=3, 3=5\}--1.0833333333333333
\{0=0, 1=1, 2=1, 3=5\}--1.1666666666666666
\{0=0, 1=3, 2=5, 3=5\}--1.1666666666666666
\{0=0, 1=3, 2=3, 3=5\}--1.1666666666666666
\{0=0, 1=1, 2=0, 3=5\}--1.25
\{0=0, 1=2, 2=5, 3=5\}--1.25
\{0=0, 1=2, 2=1, 3=5\}--1.25
\{0=0, 1=2, 2=4, 3=5\}--1.25

The Nema solution as well as Gex solution correctly identifies the V0 and V3 from the graph. V0 represents Kate Winslet node and Lang represents Stephen Lang. The Nema only gives 1 solution which is incorrect since it matches the nodes V1 and V2 to the same node Titanic in the data graph. Gex correctly answers the query and gives 10 solutions for K=10. Only the top 3 solutions are correct and therefore they have low score. All the subsequent solutions have greater score and therefore are not correct solutions. Here correct solutions doesn’t imply exact matching but approximate matching. Gex matches V1 and V2 to Performance or Titanic interchangeably.
Query 1

Nema solution: (K=10) returned only 1 solution
{0=0, 1=1, 2=2, 3=-1} -- 71.85294117647058

Gex solution: (K=10)
{0=0, 1=1, 2=2, 3=8} -- 1.25
{0=0, 1=2, 2=7, 3=8} -- 1.25
{0=0, 1=7, 2=10, 3=8} -- 1.25
{0=0, 1=10, 2=9, 3=8} -- 1.25
{0=0, 1=10, 2=11, 3=8} -- 1.25
{0=0, 1=1, 2=7, 3=8} -- 1.25
{0=0, 1=2, 2=10, 3=8} -- 1.25
{0=0, 1=7, 2=9, 3=8} -- 1.25
{0=0, 1=7, 2=11, 3=8} -- 1.25
{0=0, 1=1, 2=10, 3=8} -- 1.25

The query requires any two nodes which are approximately related to Winslet and Eastwood. Here there are two issues with Nema:

i. it returns only one solution irrespective of the value of K
ii. it is not able to match “eastwood” to any node in the graph and hence it returns -1 for node 3.

For Gex, k=10 returns multiple solutions ... in this case there are multiple best solutions with same score hence the first one is not really the only best solution.
The best solutions matches matches $0^{th}$ node to $V0, 3^{rd}$ node to $V8$ which is Clint Eastwood. The remaining two nodes are matched to “Performance”, “Titanic” and “Million Dollar Baby” interchangeably.

**Query 2:**

The query requires any node that are related by the Winslet and Cameron nodes by actor and director relationship respectively.

Nema solution: (K=10) returned only 1 solution
{0=0, 1=2, 2=6, 3=5}--20.87433155080214

Gex solution: (K=10)
{0=0, 1=1, 2=6, 3=5}--1.26666666666666666
{0=0, 1=4, 2=6, 3=5}--1.26666666666666666
{0=0, 1=2, 2=6, 3=5}--1.26666666666666666
{0=0, 1=3, 2=6, 3=5}--1.43333333333333331
{0=0, 1=7, 2=6, 3=5}--1.43333333333333331
{0=0, 1=6, 2=6, 3=5}--1.6
{0=0, 1=5, 2=6, 3=5}--1.6
{0=0, 1=10, 2=6, 3=5}--1.6
{0=0, 1=9, 2=6, 3=5}--1.7666666666666666
{0=0, 1=11, 2=6, 3=5}--1.7666666666666666

Gex solution: (K=1)
The top three solutions for Gex have “Performance” , “Direction” and “Titanic” as node 1. “Titanic” seems to be the best solution for the given query.

Example 5:

Query 1:

Nema solution (K=10)

\[
\begin{align*}
0=0, 1=1 & \rightarrow 0.6 \\
0=13, 1=16 & \rightarrow 0.6 \\
0=18, 1=16 & \rightarrow 0.9666666666666666 \\
0=17, 1=16 & \rightarrow 0.9666666666666666 \\
0=16, 1=17 & \rightarrow 0.9666666666666666 \\
0=15, 1=16 & \rightarrow 0.9666666666666666 \\
0=14, 1=16 & \rightarrow 0.9666666666666666 \\
0=4, 1=1 & \rightarrow 0.9666666666666666
\end{align*}
\]
For query 1 both Nema and Gex gives correct results.

**Query 2:**

![Diagram of a tree with nodes A, B, and V0, V1, V2]

Nema solution: (K=10)

{0=0, 1=1, 2=1}--1.7999999999999998
{0=13, 1=16, 2=16}--1.79999999999999998
{0=18, 1=16, 2=16}--2.1666666666666665
{0=17, 1=16, 2=16}--2.1666666666666665
{0=16, 1=17, 2=17}--2.1666666666666665
{0=15, 1=16, 2=16}--2.1666666666666665
{0=14, 1=16, 2=16}--2.1666666666666665
{0=4, 1=1, 2=1}--2.1666666666666665
Here Nema fails as it gives the same data node for two different query nodes. Here since the query nodes have same labels Nema fails here because it gives same edges for two different query edges.

Gex solution (K=10)

{0=1, 1=2, 2=2}--0.7666666666666666
{0=1, 1=3, 2=2}--0.7666666666666666
{0=1, 1=2, 2=3}--0.7666666666666666
{0=1, 1=4, 2=2}--0.7666666666666666
{0=1, 1=2, 2=4}--0.7666666666666666
{0=1, 1=5, 2=2}--0.7666666666666666
{0=1, 1=2, 2=5}--0.7666666666666666
{0=1, 1=6, 2=2}--0.7666666666666666
{0=1, 1=2, 2=6}--0.7666666666666666
{0=1, 1=7, 2=2}--0.7666666666666666

Gex does give correct answers here but it also involves false positives such as the solution number 1. The update function is not properly defined in the Gex paper and hence the exact implementation of the Gex update function cannot be confirmed from the paper. But I tried to implement update function but it seems it gives few false positives along with correct solutions. My implementation first finds a solution edge for the first query edge. Using this solution edge it tries to get all the solution edge which is related to the solution edges which are already present in the solution. This creates problems similar to what we have in the Nema. Therefore I have included a check which ensures that the solution edge is not selected twice and hence avoids problems similar to that Nema. The overhead with this solution is that time complexity increases because we need to check every solution edge already present in the solution every time we need to add a new solution edge.

Example 9

<table>
<thead>
<tr>
<th>Data</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0=12, 1=16, 2=16}--2.1666666666666665</td>
<td>{0=7, 1=1, 2=1}--2.1666666666666665</td>
</tr>
</tbody>
</table>
Query 1:
Nema Solution (K=10)

\{0=0, 1=2, 2=3, 3=4, 4=9, 5=7, 6=8\} -- 0.0
\{0=1, 1=2, 2=3, 3=6, 4=9, 5=7, 6=8\} -- 0.0

But Nema gives only 2 solution. Even though the first solution is correct here and it is the best solution with Nema score of 0. The second solution is a correct solution but it is not the best solution and hence its score should be not 0. Nema calculates incorrect score here for the second solution.

Gex Solution (K=10)

\{0=0, 1=2, 2=3, 3=4, 4=9, 5=7, 6=8\} -- 1.4499999999999997
\{0=0, 1=2, 2=5, 3=4, 4=9, 5=7, 6=8\} -- 1.4857142857142853
\{0=1, 1=2, 2=3, 3=4, 4=9, 5=7, 6=8\} -- 1.4857142857142853
\{0=1, 1=2, 2=3, 3=6, 4=9, 5=7, 6=8\} -- 1.4857142857142853
\{0=1, 1=2, 2=3, 3=6, 4=9, 5=12, 6=8\} -- 1.5214285714285711
\{0=0, 1=2, 2=3, 3=6, 4=9, 5=12, 6=8\} -- 1.5214285714285711
\{0=0, 1=2, 2=3, 3=6, 4=9, 5=12, 6=8\} -- 1.5214285714285714
\{0=0, 1=2, 2=3, 3=6, 4=9, 5=12, 6=8\} -- 1.557142857142857
\{0=0, 1=2, 2=3, 3=4, 4=9, 5=12, 6=8\} -- 1.6285714285714283

Gex calculates correct solution and its scoring function is accurate in this scenario. The second solution given by the Nema is 4th best solution for Gex and the score is higher than the best solution.
7.1 Testing on Yeast Dataset

I used Yeast dataset for testing Gex and Nema since the Yeast dataset represents a more realistic dataset. Yeast dataset represents protein interactions and describes the constituent elements of each protein. Yeast dataset has approximately around 3000 nodes. The dataset is stored in the Neo4j and is integrated in the framework. The results are summarized in the table below:

To compare them both the algorithms were run using the same set of queries. The queries were generated by extracting an actual edge from the graph and then introducing various noise to it. This resulted in the various versions of the same query having some structural and label variations. The expected output was getting the original structure back as one of the results.

<table>
<thead>
<tr>
<th>No.</th>
<th>Gex</th>
<th>Nema</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>When the Query nodes &gt; Solution nodes Gex will not give you original structure but it will try to match the extra node to some node in the graph*</td>
<td>Nema gets the original structure back</td>
</tr>
<tr>
<td>2</td>
<td>The solution generation process of Gex is not limited to 2 hops.Hence it generates correct results.</td>
<td>Nema restricts its approximation to 2 hops.</td>
</tr>
<tr>
<td>3</td>
<td>It always generates k number of solutions</td>
<td>Most of the time Nema fails to generate more than K number of solutions</td>
</tr>
<tr>
<td>4</td>
<td>Query nodes with same labels: Gex generates correct solution but with some false positives #</td>
<td>Nema fails in this scenario</td>
</tr>
<tr>
<td>5</td>
<td>Query edges with same labels: Gex generates correct solution but with some false positives #</td>
<td>Nema fails in this scenario</td>
</tr>
<tr>
<td>6</td>
<td>Due to higher time complexity of the algorithm the solution generation process slows down **</td>
<td>Faster compared to Gex</td>
</tr>
</tbody>
</table>

* Gex algorithms always try to generate a list of matching nodes corresponding to the each query node. This is followed by generation of edges by using cross product between the list of matching nodes. Every edge has an associated cursor attached to it which maintains the list of candidate edges generated from the above step. Therefore the when an extra node is added in
the query it will always generate a new cursor for the edge containing this new node. So it is not possible for Gex to approximate a structure containing lesser number of edges than the total number of query edges.

# Gex does give correct answers here. The update function is not properly defined in the Gex paper and hence the exact implementation of the Gex update function cannot be confirmed from the paper. But I tried to implement update function but it seems it gives few false positives along with correct solutions. My implementation first finds an solution edge for the first query edge. Using this solution edge it tries to get all the solution edge which are related to the solution edges which are already present in the solution. This creates problems similar to what we have in the Nema. Therefore I have included a check which ensures that the solution edge is not selected twice and hence avoids problems similar to that Nema. The overhead with this solution is that time complexity increases because we need to check every solution edge already present in the solution every time we need to add a new solution edge.

** GeX has a large computation overhead if we have a graph such as Yeast, which does not conform to the RDF properties. GeX algorithm suggests some rules for creation of indexes to increase algorithm efficiency but these rules are not generic and very specific to RDF graphs.

For some cases as in the Yeast example the creation of indexes itself takes about $O(E^3)$ where $E$ is the number of edges.

GeX algorithm also depends on the data preprocessing techniques such as creation of indexes on labels and edges to reduce significant computation overhead. The preprocessing techniques require require one to have good knowledge about the dataset.

8. Challenges

The biggest challenge was to understand the algorithm implemented in paper. The main task was to understand the pseudo code and relate it to the contents of the paper. The major part of initial milestones was to just understand the algorithm.

The next challenge was to implement the algorithm in Java based on the pseudo in the paper. The implementation was complex since Gex was proposed for RDF types and it was difficult to implement the same algorithm for non RDF types of graphs. The understanding and integration of the framework with Gex was also very important. Some of the difficulties I faced while implementing the algorithm was the lack of clear explanation for some functions. For example update() function was explained briefly with no implementation details. I had to assume the working of the function and implement it. Overall the project was a challenging task but was very interesting to learn.
9. Future Work

Although I implemented Gex and tested with few sample datasets, I feel there is some scope for future work which may involve testing the performance of Gex using RDF graphs. I think working on RDF graphs would be good parameter to measure Gex results since the algorithm was implemented at the first place for RDF types of graphs. It would be very interesting to see the performance of Gex for RDF types graphs and compare it with Nema. Also testing with many other datasets to compare the results with Nema will be useful for evaluating the algorithms.

10. References

