**Searching in a Knowledge Graph**

**Assumption:**

1. Find the closest node defined by collective closeness:  
   *Collective closeness = shortest distance from every query word*
2. *Distance from every node is considered for avoiding the greedy based algorithm from being biased to any single node.*
3. *Many things could be done easily with neo4j cypher query, but I chose to do it myself within python for implementing the algorithms.*

***Food for thought:***

1. *Which one is to be considered close: Nodes having connection to all query words with large distance or nodes with less distance but not connecting to all nodes?*
2. *Handling multiple relationship/weight for the same pair of nodes? (same node, same relationship direction)*
3. *Total distance of a node from query words i.e. collective distance, adds the weight of common relationship two times. Can anything be changed in this for making the algorithm more efficient?*
4. *The distance calculated by the algorithm for query words and nth neighbor hop may not be the shorted distance.*

**Solution 1:**

**Minimum spanning weighted tree with Prim’s algorithm**

A brutal force approach for finding the closed node for a given list of queries.

Steps:

1. Build a minimum weighted spanning tree (only for query words present in the knowledge graph)

2. From the created tree, find the sum of the distance of every word with the query words i.e. collective distance. For unconnected node, add a constant large value.

For e.g.:

a,b,c,d are query words.

e,f,g,h,I,j are the other words in the knowledge graph.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | e | f | G | h | i | J |
| (collective distance)  a+b+c+d |  |  |  |  |  |  |

# TODO: Changing cost function to average (a+b+c+d)

3. Select 5 nodes having the list value of cost function (a+b+c+d)

**Pros:**

1. Could provide the optimized solution.

**Cons:**

1.It need to process each and every node in the graph.

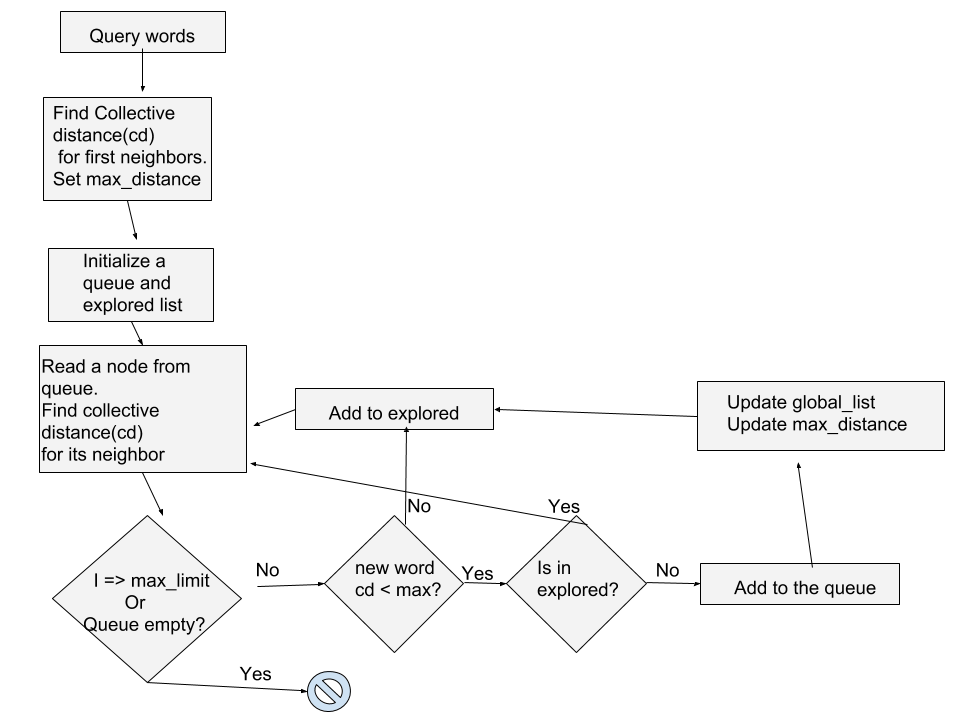
2. Takes more space and time. Time Complexity O(nd). Where n= no. Of nodes and d= no. of query words in the list.

**Solution-2:**

**Recursive Neighbor algorithm based on Breadth First Search**

Solution 1 computes the shortest path for every node present in the knowledge graph. It is absurd to compute the shortest path for every node as the closest node lies in the neighborhood of the query node. It may be a good way to find an optimal solution in smaller knowledge graph. But for larger graph, it is computationally expensive and inefficient.

This algorithm starts with the query words and explore their neighbors based on the collective distance. A node having less collective distance is assumed to be the closed node with the query words.



Steps:

1. Find the collective distance of each neighbor of all query words and sort them in an ascending order.
2. Create a list of n nodes (n: top n close nodes from the query words). Select a max distance: nth node. (i.e. collective distance of the nth node).
3. Expand all neighbors of the nodes expect nth node. If the collective distance of that node is less than the max distance, remove the nth node and add the new node to the list.   
   For e.g.: This is top 5 first hop close word’s distance with query words.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 4 | 6 | 10 |

Exploring the nodes having weight less than the max, we get a possibility of finding closer node. But exploring nodes having greater distance, can only provide neighbors with more weight only unless we implement a optimal shorted path algorithm.

1. Repeat the process till no new neighbors with collective distance less than max distance is found

**P.S:**

The value used to represent the unconnected nodes plays an important role in determining the closer node.

**If the value is too high:** missing connections with only less no. of node is also assumed to be more far.

**If the value is too low:**  Nodes having no connection could be assumed to be closer than really connected nodes.

**Solution-3:**

**Improvement to solution 2:**

Same as solution 2, but finding shortest path to each neighbor in the queue using shortest path algorithm for finding optimal closeness (i.e. Dijkstra’s algorithm).

***Other Solutions to explore***

**Solution-4:**

**Closeness Centrality based algorithm:**

Inverse of sum of distance with query words could be a good matric for the closeness.

**Solution-5:**

**Page Rank algorithm (accounting for relationship degree)**

Ranking could be an important metrics which accounts for the relationship degree of the nodes.