**CE2001/CZ2001 ALGORITHMS**

**Project 2 - Graph Algorithms**

**Done by:** <Tutorial Group SE1 Group 6>

Woon Yoke Min, Tan Yap Siang, Tan Leng Hwee Gordon, Tey Yi Heng, Tan Jun Hong

**Date:** 2nd November 2020

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Design of Algorithm**

Task A and B: Breadth First Search (BFS)

We explored many algorithms. Algorithms like Dijkstra take into account weights of the graph, which is unnecessary in our problem statement. Since this is an unweighted and undirected graph, BFS seems most appropriate, as the algorithm expands each node at the present depth before moving on to the next. Since all the weights are equal, the depth becomes the distance, and we can use it to find the shortest path. We ensure that every node is visited through a “marking” mechanism, where upon exploration of each node, we mark that node as visited and add the unexplored neighbours into a queue. This repeats until every node is visited. (those that are connected to the source node).

For this part, we will be using the following data structures; a queue (LinkedList), a hash table (HashMap) of load factor 1 called previousNodes to store edges where key is the current node and value is the previous node. We used a hash table instead of an array as the graphs are not in running number i.e. in a graph of 1 million nodes, node number 50 may not exist. Adding a node into the hash table acts as the marking mechanism in BFS where nodes that are keys of the hash table are considered as visited and will not be visited again.

The BFS will first be initialised by adding all hospitals into a queue. We will also add a key-value pair of hospital, hospital for into previousNodes. This is to indicate the end of backtracking, where the key and value pairs contain the same node, indicating a hospital is reached.

After which, we will remove the first node in the queue and check whether its nearest neighbours have been visited before using the hash table previousNodes. If it has not been visited before, we will add the neighbour as the key and the node that is being checked as the value into the hash table. We will also add the neighbour into the queue. If it is visited, we repeat the process with the next neighbour, if any. This continues until the queue is empty.

Returning Shortest Path

We will perform backtracking by repeatedly obtaining the previous node from the previousNodes hash table until we eventually get to a hospital. This is through obtaining key value pairs from previousNodes repeatedly where the value of the key-value pair will be assigned as the key to be searched. The nodes visited while backtracking will form the path and distance will be computed in terms of the edges. We return the shortest path to a hospital for ALL nodes.

Task C and D: Modified Breadth First Search

For this part, we will be a different set of data structures:

1. A queue that stores the current node and the hospital it is from. Because 2 items need to be stored in queue, every item in the queue is an array, as follows : [currentNode #, the hospital # its path is from] (LinkedList of arrays of size 2)
2. Partial Minimum Distance Spanning Tree (PMDST). This is a hash table with hospitals as key and another hash table as value. The nested hash table has a node as its key and previous node as value. This represents the individual edges that constitute the path to the hospital. (HashMap of another HashMap)
3. Visited order. This is a hash table with node being the key and linked list of hospitals visited in order as value. This represents each node and the sequence of hospitals that visited that node. It is automatically ordered as the 1st hospital added to the sequence is also the nearest hospital, the 2nd hospital is the 2nd nearest and so on.

The modified BFS will be initialised with hospitals and hospitals being added as an array pair into a queue first. The first element of the array pair will indicate which node to visit, while the second element shows which hospital the path is from. We also create another hash table to store each hospital’s path. For each hospital, we will also add a key-value pair of (hospital:hospital). This is to indicate the end of backtracking, similar to the BFS above. We have another hash table that stores the node as key and the hospitals that visited it in order, which is just the hospital when initialising.

After which, whenever we visit a node with a hospital its path is from, we check if the adjacent nodes have been visited before. If not, we will initialise a new linked list for that node and add it to the visited order hash table. Following that, we will check if k hospitals have already visited this adjacent node and if the current hospital already exists in the adjacent nodes’ linked list of visited hospitals. If not, we will add this adjacent node (key) and the current hospital into the linked list (value) to the visited order hash table. These adjacent nodes will also be added to the queue, along with the hospital. The edge will also be added to the PMDST of the respective hospital, with the adjacent nodes as “current nodes” and the current node as “previous node”.

This repeats until the queue is empty. The marking mechanism for this algorithm is slightly different as we have to check if k hospitals visited that node, so we have to use a list instead and check if the list size is less than k.

Returning Shortest Path

We retrieve all nodes that are connected to at least one hospital in the graph. From which, we will get the list of hospitals for every node from visitedOrder. We will perform backtracking similar to how we did to the BFS algorithm by using the paths we stored in PMDST for hospitals. We will obtain distances to the k-nearest hospitals for all nodes.

**Time and Space Complexity Analysis**

**Pseudocode for Task A and B:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Algorithm: Breadth First Search

Notes: think of the key:value pairs in HashMap previousNodes as edges between key and value

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1: function executeBFS(Graph graph, LinkedList queue)

2: new HashMap previousNodes // stores nodes (key) and its previous nodes (value)

3: // make previousNode = currentNode for hospitals and add hospitals to queue

4: for (hospital in allHospitals) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ N.A.

5: add hospital(**key**):hospital(**value**) to previousNodes // mark hospitals as dead-end

6: queue.add(hospital)

7: // Start BFS using queue

8: while Queue !empty \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ O(|V|)

9: currentNode = first element of queue

10: remove first element of queue

11: for (**each neighbour of currentNode**) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Max |E| loops (Only if that node connects to every other nodes)

12: if (neighbour **NOT** visited)

13: // Mark visited

14: add neighbour(**key**) and its currNode(**value**) to previousNodes

15: add node to end of queue

16: return HashMap of previousNodes

1: function writeShortestPath(Arraylist nodes, HashMap previousNodes)

2: for (every node in graph) { \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ O(|V|)

3: if (**node does not have edge that is part of a route to a hospital**)

4: continue

5: currentNode = node

6: previousNode = previousNode.getValue(node)

7: distance = 0

8: append currentNode to shortestPath

9: // Backtrack from node to hospital

10: while (**not hospital**)

11: increment distance

12: append previousNode to shortestPath

13: currentNode = previousNode

14: previousNode = previousNodes.getValue(previousNode)

15: return shortestPath

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Time Complexity Analysis:

Since we use adjacency lists to check the neighbours of currentNode, the worst case occurs in a connected graph, where there exists a path from any vertex to any other vertex. Every node will be visited. Since every node will be added inside the queue eventually, the calculation of time complexity for initial addition of the hospitals to the queue can be left out. Hence, worst case time complexity of **executeBFS**: O(|V| + |E|). In fact, the time complexity of a connected graph is always O(|V| + |E|).

Backtracking time complexity is dependent on the length of the shortest route to hospital from that node. The absolute worst case is when the hospital is at the far end, and the furthest node away from the hospital has to traverse every other node to reach that hospital. In that case, the worst case time complexity is O((|V| + |E|)^2).

Space Complexity Analysis:

To store nodes and its neighbours, we used the concept of adjacency list and implemented a **HashMap<Integer, ArrayList<Integer>>** to store the graph, where ArrayList<Integers> contains the neighbours of each node. Total space complexity **to store graph** is O(|V| + |E|).

We used a **HashMap<Integer, Integer> previousNodes** to store the edges of paths that lead to a hospital. Its worst case space complexity is O(|E|), where every edge in the graph is part of a route to a hospital. **Queue** is implemented with a **linked list**, with a maximum of |V| elements in the worst case. Hence, worst case total space complexity is **O(|V| + |E|)**.

**Pseudocode for Task C and D:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Algorithm: Modified Breadth First Search (k refers to the top k nearest hospitals from a node)

NOTE: queue is a linkedList of arrays of [currentNode #, the hospital its path is from]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1: function executeModifiedBFS(Graph graph, LinkedList<int[]> queue, int k)

2: new HashMap <Hospital #, HashMap<Node #, prevNode>> partialMDST

3: new HashMap <Node #, LinkedList of hospitals visited in order> visitedOrder

4: // make previousNode = currentNode for hospitals and add hospital to queue

5: for (hospital in allHospitals) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ N.A.

6: queue.add([hospital, hospital])

7: // Create HashMap of hospitals to have prevNode as currentNode, add to partialMDST

8: new HashMap previousNodes

9: add hospital(**key**):hospital(**value**) to previousNodes // mark hospitals as dead-end

10: add hospital(**key**):previousNodes(**value**) to partialMDST

11: // Create LinkedList of hospitals in visited order, add to visitedOrder

12: new LinkedList visitedHosp

13: add hospital to visitedHosp

14: add hospital:visitedHosp to visitedOrder

15: // Start BFS using queue

16: while Queue !empty \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ O(k|V|)

17: currNodeAndHosp[2] = first int[] of queue // [0] is currNode; [1] is currHosp

18: remove first int[] of queue

16: for (**each neighbour of currNodeAndHosp[0]**) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Max k|E| loops

17: if (not visited)

18: add neighbour(**key**):new LinkedList(**value**) to visitedOrder

19: if (less than k nearest hospitals in LL and currHosp has not visited node (aka edge not in partialMDST))

20: add (neighbour:currHosp) to queue

21: add currHosp to LL of hospitals visited by visitedOrder.get(neighbour)

22: add edge(neighbour, currNode) to partialMDST.get(currHosp)

23: return partialMDST, visitedOrder

1: function writeKShortestPaths(Arraylist nodes, HashMap partialMDST, HashMap visitedOrder)

3: for (node in nodes) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ O(|V|)

4: if (!**visitedOrder.containsKey(node)**)

5: continue

6: currentNode = node

7: previousNode = previousNode.getValue(node)

8: // Find path for each hospital

9: for(**int 1 to # of hospitals that visited node**) { \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ O(k)

10: currHosp = first element of LL in visitedOrder.get(node)

11: remove first element of LL

12: currNode = node

13: prevNode = partialMDST.get(currHosp).get(node)

14: distance = 0

15: create new path // empty

16: append currNode to path

17: while (**not hospital**) {

18: Backtrack from node to hospital, similar to writeShortestPath

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Time Complexity Analysis:

Similar to part A and B, initialising and adding hospitals to the queue can be excluded from the calculation of time complexity since in the worst case, k\*|V| nodes will be added to the queue. The worst case occurs in a connected graph, with each nodes visited k times and thus having the length of |E|, which results in maximum of k\*|E| times of running the for loop. Hence, worst case time complexity of **executeModifiedBFS**: O(k\*|V| + k\*|E|). This time complexity holds for all connected graphs.

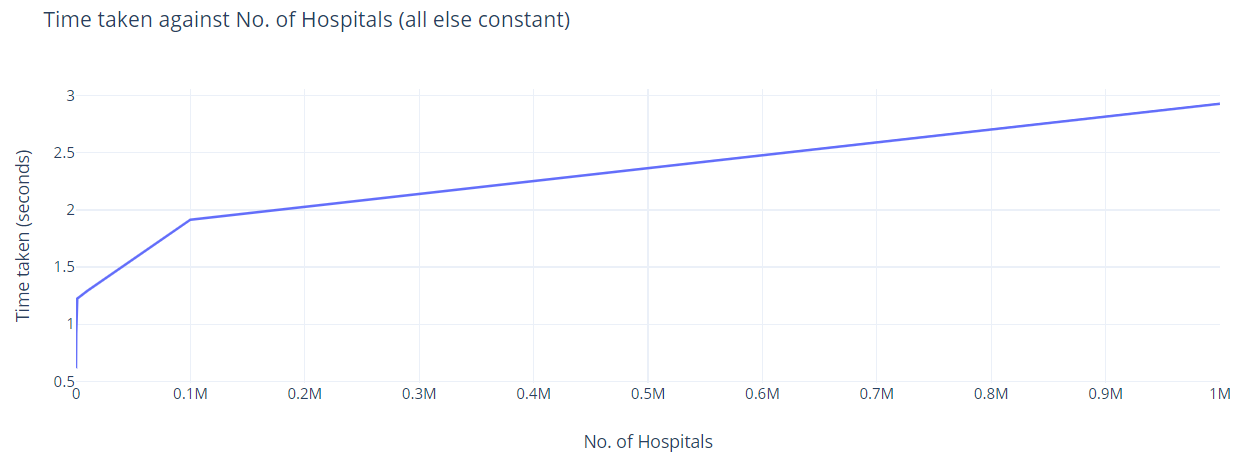
k determines the number of paths to print. The worst case is all the hospitals are at the far end, and the furthest node has to traverse all other nodes to reach a hospital. Hence, worst case time complexity of **writeKShortestPaths**: O(k\*(|V| + |E|)^2).

Space Complexity Analysis:

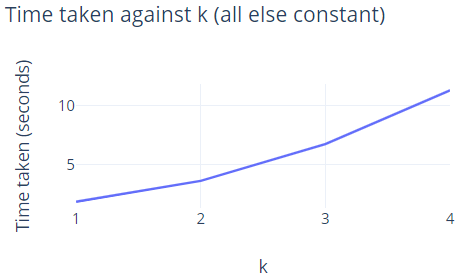
Similar to part A and B, we used **adjacency lists** to store the graph. Hence, total space complexity is O(|V| + |E|). We implemented a **HashMap <Hospital #, HashMap<Node #, prevNode>>** partialMinDistSpanTrees. HashMap<Node #, prevNode> represents all edges that are part of a path to that particular hospital. This requires O(k\*|E|) in the worst case, where every edge is part of the path to a kth nearest hospital. We also used a **HashMap <Node #, LinkedList<Integer>>** visitedOrder to store a linked list of the order in which each node visited the hospital(s). This requires O(k\*|V|) in the worst case. Queue is implemented with a **LinkedList<int[]>**. In worst case, where k\*|V| nodes are added into the queue, space complexity of O(k\*|V|). Hence, worst case total space complexity is **O(k\*(|V| + |E|))**.

**Empirical Study (Refer to Appendix for the actual raw data)**

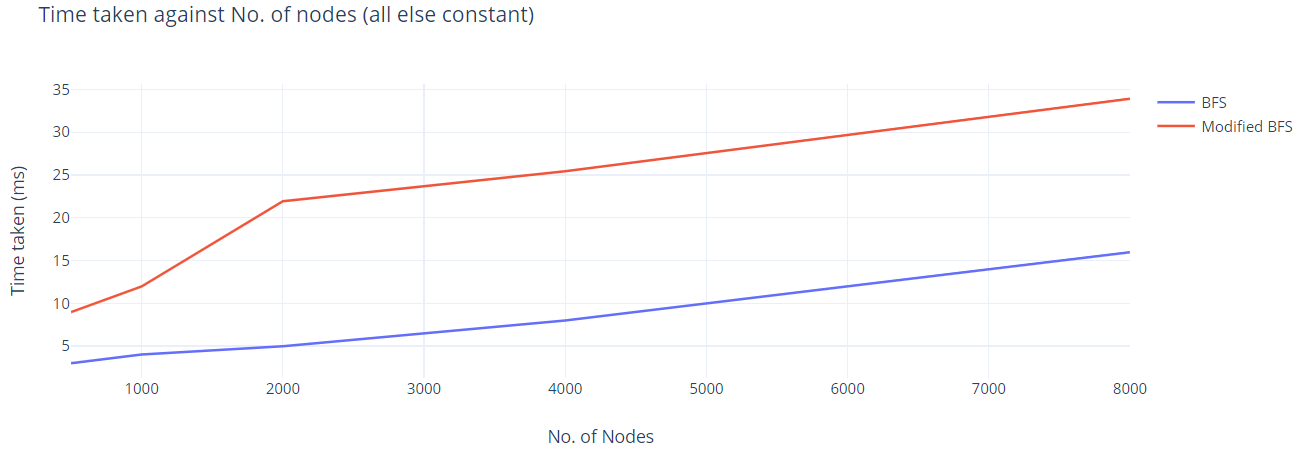
Scenario #1: running BFS with k = 1, roadNet-CA (Nodes: 1965206 Edges: 5533214) as graph, changing no. of hospitals



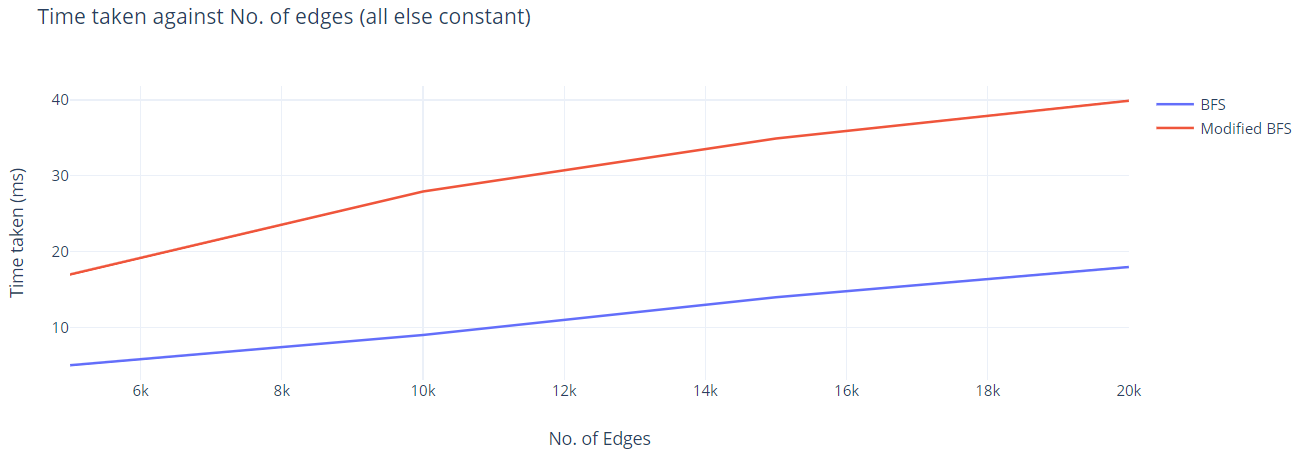
Scenario #2: running modifiedBFS with h = 4, roadNet-CA (Nodes: 1965206 Edges: 5533214) as graph, changing value of k



Scenario #3: h = 5, k = 2, keeping no. of edges constant, changing no. of nodes



Scenario #4: h = 5, k = 2, keeping no. of nodes constant, changing no. of edges



We assumed that time taken for initialisation is negligible. In scenario #1, we can observe that there is a trend of increasing time taken. This may be due to the change in number of visited nodes that we checked. However, this increase is not significant as the proportion of increase in time in comparison to that of the number of hospitals is very small. For example, the increase of 10 times more hospitals (from 0.1M to 1M) only resulted in a 1.5 times increase in time taken, which is not really a big time concern. In scenario #2, we can observe that the number of k causes the time complexity of modified BFS to increase by multiples of k. In both scenarios #3 and #4, we can observe that when the number of nodes and edges increases by x amount, time taken to run BFS increases by x amount as well, while that of modified BFS increases by 2x amount as we used k = 2 to run these two experiments.

Hence, we can conclude that the empirical findings of the effects of h and k on the performance of the relevant algorithms are consistent with our theoretical analysis.

**References:**

* Online Graph Maker · Plotly Chart Studio. (n.d.). Retrieved from <https://chart-studio.plotly.com/create/>
* Multi Source Shortest Path in Unweighted Graph. (2020, Apr 24). Retrieved from <https://www.geeksforgeeks.org/multi-source-shortest-path-in-unweighted-graph/>

**Statement of Contribution:**

All team members contributed equally.

* Implementation of all algorithms (done in Java): Jun Hong
* Empirical Experiment And Analysis of Time and Space Complexity (Report): Yoke Min
* Design of algorithms (Report): Yap Siang, Gordon
* Presentation: Yi Heng