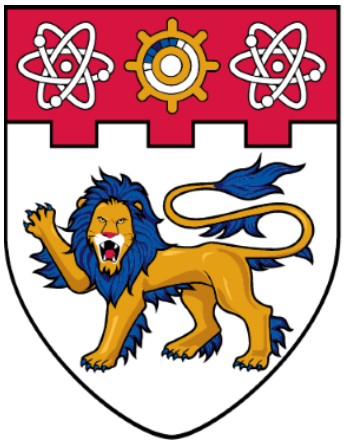


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Project 2 Report: Graph Algorithms

Team Members:

Name	Matriculation Number
Acharya Atul	U1923502C
Anil Ankitha	U1923151C
Arora Srishti	U1922129L
Kothari Khush Milan	U1922279J
Pal Aratrika	U1922069F

Part A

Algorithm Design: Memory Search

This problem can be solved using the standard breadth-first search algorithm. However, the task is to find the nearest hospital for all the nodes. Therefore, the Breadth first search algorithm can be modified such that it remembers the past expansions and uses them to calculate the nearest hospital for other nodes. Storing of past explorations helps to calculate the nearest hospital to nodes quicker and it prevents repeated expansions. If the nearest hospital from Node '1' is 'h' and the shortest path is '1'-'2'-'3'-'h', then 'h' is the nearest hospital to all the nodes lying in this path. When expanding an arbitrary Node 'x' and after a certain depth it reaches a Node 'y' whose nearest hospital has been found, the algorithm can cut the maximum depth of exploration to cost of expansion to Node 'y' + path length of Node 'y' because any exploration further than this depth would result in a sub-optimal solution and this prevents redundant exploration of Node 'y'.

Pseudo code

find_path(node)

```
FOR all nodes DO:
 IF node is hospital
  CONTINUE with next iteration
 IF node's path has been found
  CONTINUE with next iteration
 ASSIGN min_cost = infinity
 ASSIGN connection = null
 ENQUEUE(node)
 WHILE queue is not empty and
 node's path has not been found
 DO:
   current_node = DEQUEUE()
   FOR all neighbors of
   current_node DO:
    IF neighbour is hospital
    retrieve path using neighbour

ELSE IF neighbour' path has been
found and cost of expansion to
neighbour + path cost of neighbour
to hospital < min_cost:
  min_cost = cost of expansion to
  neighbour + path cost of
  neighbour to hospital
  connection = neighbour

ELSE IF cost of expansion >
min_cost
  retrieve path using connection

ELSE:
  IF neighbour has been visited:
    CONTINUE next iteration
  Mark neighbour as 'visited'
  ENQUEUE neighbour
```

retrieve_path(node)

```
IF node's path has not been found:
   node's path = [node]
WHILE node is not null DO:
   node.parent's path =
   [node.parent] + node's path
   node = node.parent
```

Algorithm Analysis

Let |V| be the length of adjacency-list, that is, there |V| key-value pairs. The adjacency list in this algorithm is the dictionary of all the nodes of graph G. Thus, we have a total of |V| nodes in graph G. Let H be the total number of hospital nodes. Hence, we have V-H nodes that are not a hospital.

The for loop that loops through the adjacency-list calls the function retrieve_path and passes a single node (single key) as the parameter. As the loop executes V times, therefore, we have V function-calls in total. Thus, the time complexity of this loop is equal to $O(|V|)$, where |V| is the number of nodes in the graph. We have linear time complexity.

Moving on to the body of the function called in the above loop, we look at the best case as well as the worst case scenarios.

In the best case, each node passed onto the function is already a hospital. As a result, we will only have l comparison and the size of the shortest_path dictionary would equal to the number of hospitals in the graph. Hence, the time complexity of this particular function would be $O(|V|)$ due to |V|

comparisons. But since we have V = h, therefore, we can say the time complexity in the best case is *O*(h). Thus, its linear in best case.

In the worst case, the node is neither in the shortest_path nor in the hospitals dictionary. Due to this we have V-h nodes which are not a hospital and their path to the hospital is to be found. In this case the node passed onto the function is first appended to a queue. explored_node is a dictionary that keeps track of the distance of the parent node from a hospital. The while loop: while len(queue)>0 has an initial value of 1. Hence, it will execute at least once always. If the initial conditions of this loop are True then the retrieve_path function is called immediately. The time complexity of that function has been analysed separately ahead as well. On entering the loop, the first node of the queue is popped out. As a result, for the queue length to be greater than 0, the last else condition has to be executed. In the else condition, we access the value of the key: adjacency_list[curr_node], where curr_node is the node popped out from the queue. This value is a list of all the nodes that have relation with curr_node. Hence, if we add the length of all such lists of all the |V| nodes, we get the total number of relations present in the graph. This is the total number of edges present in the graph. Let's assume that in the graph G, which has |V| nodes, there are |E| edges. Hence, the outer while loop with the inner for loop will execute for a total of m times in the worst-case scenario. Hence, the time-complexity of the find_path function in worst case scenario is ***O*(|E|)**.

The retrieve_path function is called in the find_path function. It has two parameters, the given node and a dictionary of all the explored nodes which is stored in the variable expanded_nodes.

This function performs a comparison every time it is called, to check if the given node is present in the shortest path or not. If false, it adds the node. Best-case scenario for the loop present in the function would be that no node has been explored, that is, the expanded nodes dictionary is empty, and the parent node is always the hospital. It will have linear time complexity in this case of O(h).

Worst-case scenario would be that the expanded_nodes[node][1] has a list in which all nodes have been explored. Therefore, the number of times the loop will execute will be equal to the length of that list. In the graph g, we have |V| - h nodes that are not part of the hospital list. Hence, the loop will execute |V| - h times in the worst case. As a result, the time-complexity of this function is linear and is equal to *O*(|V-H|).

After analysing all the functions separately, we can compute the time complexity of this entire algorithm. In the best-case scenario, the time complexity will be ***O*(h)** [Since h=|V|], which is linear. However, in the worst-case scenario the time complexity of this algorithm will be the combination of the time complexities of find_path and retrieve_path functions. Find_path has a time complexity of *O*(|E|) and it calls the function retrieve_path for each node and that has a time complexity of *O*(|V-H|). Thus, the worst-case time complexity of this algorithm is ***O*((|V|-h)|E|)**.

Part B, C and D:

Algorithm Design Multi-Source BFS

The problem can be solved using a multi-source breadth first search with a small modification. All hospitals are enqueued and will first visit all the source vertices. After that it will visit the vertices which are at a distance 1 from all source vertices, then at a distance 2 from all source vertices and so on and so forth. The variables are different for part (b) and part (c), (d) due to the nature of the tasks. However, the underlying idea is the same. For Part b, the queue contains the node ids that have been visited and the 'path' variable keeps a track of the distance to the nearest hospital. For parts (c) and (d) The queue contains entries of tuples of the form

(node id, hospital id) so that the algorithm can keep a track of the hospital that the node is being expanded from. Once the 'k' nearest hospitals of a node have been found, any successive (node id, hospital id) combination will not be enqueued.

Pseudo Code

```
ENQUEUE all hospitals
Mark all hospitals as visited
WHILE queue is not empty DO:
  node = DEQUEUE()
  FOR all neighbours of node DO:
    IF length of neighbour's
    hospital list = k:
       CONTINUE next iteration
    ELSE IF (neighbour, hospital
    id) is not visited:
        Mark (neighbour, hospital
        id) as 'visited'

        ENQUEUE ((neighbour,
        hospital id))

        APPEND hospital id to
        neighbour's hospital list
```

Algorithmic Analysis

Let the total number of vertex nodes, which includes both hospital and non-hospital nodes be 'V' and the total number of hospital nodes be 'h'. Let 'k' be the number of nearest hospitals required. The algorithm(adjacency_list, hospitals, k) is the function which finds the k nearest hospitals. The initialisation of variables nearest_hospitals, visited_nodes and queue takes O(1) or constant time. As a pre-process, a loop runs through the list of hospitals to push all hospital nodes into the queue and mark them as visited. The loop runs h times hence this takes O(h) time. Then the multisource BFS algorithm starts. For the first k hospitals that reach a node, all the vertices get queued and dequeued once and all the edges can be traversed twice, both forward and backward. The number of nodes visited, which is stored in 'visited_nodes' follows the following function:

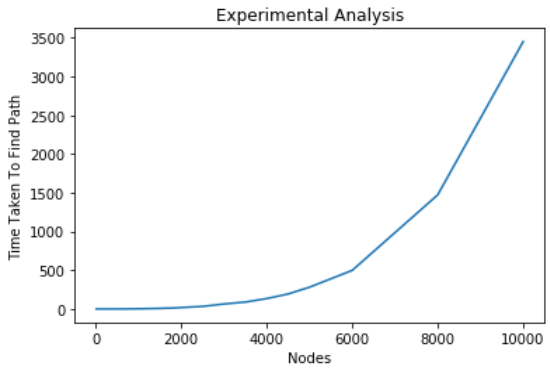
'visited_nodes' = $\begin{cases} k * |V| + h \ when \ k < h \\ h * |V| \ when \ k \geq h \end{cases}$

The number of 'visited_nodes' is equivalent to the number of items pushed into the queue. Therefore, each node is pushed into the queue at most k times and each edge is visited at most 2*k times. Thus, it has ***O*(k*(|V|+2|E|))** time complexity. Substituting the value of k = 1, we get the average time complexity of part (b) to be O (|V| + 2|E|). Therefore, this algorithm solves part (b) as the time complexity is not dependent on h. Substituting the value of k = 2, we get the time complexity of part C to be *O*(2*(|V|+2|E|)).

In the worst case, it is a complete graph where E = V(V-1). Hence the worst-case time complexity will be *O*(kV^2). In the best case, the sparse graph may have E = V-1. Hence the best-case time complexity will be *O*(kV).

Experimental Analysis

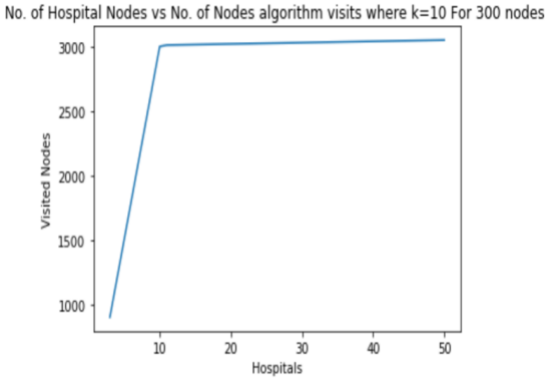
Part A:



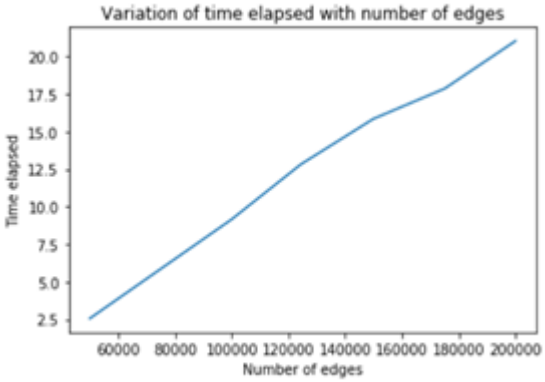
The above graph is a result of the experimental analysis performed by our team on the algorithm for part A and B. The intuition for this analysis is that we compare the time elapsed in finding the shortest path for all possible nodes with the total number of nodes in the graph. The X-axis consists of number of nodes and the y-axis represents the time taken to find the path in seconds. From the graph, one can see that the time elapsed is quite similar for the first 2000 nodes. But after that threshold is crossed, we see a curve that keeps on getting steeper as the

number of nodes increase. This is because as the number of nodes become larger, the number of edges increase by a lot more. This causes the graph to be much more complex and as we have proved in our theoretical analysis that time-complexity in worst-case is of the order $O(|V||E|)$, the time taken will also increase since both |V| and |E| increase, where |E| increases by a much greater magnitude.

Part B, C and D:



Above is the graph showing experimental analysis of the parts c and d. The X-axis on the graph indicates the number of hospital nodes and y-axis represents the number of visited nodes. In this particular case we have chosen 10 hospitals for 300 nodes. From the graph we can observe that there is a steep linear relationship till a threshold value of k where the visited nodes equal number of hospital nodes*number of total nodes. After the threshold, the slope of the line is very less. Hence, the performance of the algorithm is better with h>k, which is the general case, and it grows very slowly after h crosses k. From here we can conclude that the time complexity depends on k*V.



In this above graph, the X-axis shows the Number of edges and the Y axis shows the Time elapsed to run the algorithm. It is a linear relationship between the time elapsed and the number of edges hence from here we can conclude that the time complexity will depend on k*E.

Combining both the inferences stated above, the time complexity comes to O(k*(V+2E)) which is consistent with our algorithm analysis.

Conclusion

After testing the algorithms on both random graphs and real road networks, they perform the tasks mentioned in part (a)-(d) correctly and effectively. The algorithm for part (a) has a time complexity of $O((|V| - h)|E|)$ which is further proved by the graph obtained in its experimental analysis. The algorithm for part (c) and (d) has a time complexity of $O(k*(|V|+2|E|))$ which is again proved by the graph obtained in its experimental analysis. Hence, we can conclude that both the algorithms designed by our team are successful and effective.

Contribution Statement

Atul Acharya	Algorithm Design and Algorithm code
Anil Ankitha	Presentation and Report Writing, Implementation of Real road Network Data
Arora Srishti	Implementation of real road networks, pseudocode writing, presentation
Kothari Khush Milan	Theoretical and Experimental Analysis of Memory Search Algorithm
Pal Aratrika	Theoretical and Experimental Analysis of Multi-Source BFS

References

https://www.geeksforgeeks.org/multi-source-shortest-path-in-unweighted-graph/