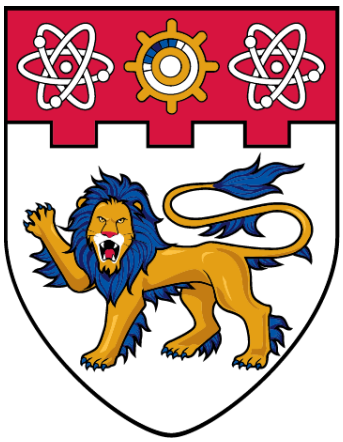

NANYANG TECHNOLOGICAL UNIVERSITY

SINGAPORE

CZ2001 Algorithm Lab 2 Report

Group 4

Chang Heen Sunn U1920383H

Luo Wenyu U1922009G

Ooi Wei Chern U1920504J

Saw William Joseph U1921246F

Tan Ching Fhen U1920787D

1 Introduction

From social networking to global positioning system (GPS), graph data structure and algorithm are ever-present in our modernized lives. While many are unaware of the underlying mechanisms of graph algorithms, their application and use are common and each algorithm are designed to solve a specific problem such as navigation, determining the shortest path to a destination, and maze solving. In this project, we will utilise and modify the existing algorithm and apply the algorithm to an undirected unweighted graph which represents a city's road network. We aim to find the nearest hospital from each of the node and the distance between them, also to deal with different circumstances, we improve the algorithm to find the top K nearest hospitals and their distances.

2 Breadth-First Search

Our algorithm utilises the idea of Breadth-First Search (BFS). The first method, *.solve()*, finds the path to the nearest hospital from every node. It is unique because it applies BFS on all hospitals simultaneously, thus it can find the shortest path and distance with just one traversal of the road network. The second method, *.topK()*, finds the distances to top k nearest hospitals from any node. Both algorithms are altered such that it will stop traversal down a path if there is no need to.

```
defaultdict(list,
            {'0': ['1', '6309', '6353'],
             '1': ['0'],
             '6353': ['0', '6354', '6364', '6386'],
             '6309': ['0', '6310', '6344'],
```

Figure 1

We first create an adjacency list of the graph in the form of a dictionary as seen in Figure 1 above. For instance, 1, 6309 and 6353 are adjacent to 0.

```
def solve(self):

    Q=Queue()
    for H in self.hospitals:                    # shortest path of a hospital is simply to itself
        self.solutionPath[H][H]=[H]
        self.solutionDistance[H][H]=len(self.solutionPath[H][H])-1
        Q.put(H)
                                                # apply BFS to each hospital simultaneously!
    while not Q.empty():
        currentNode=Q.get()
        H=list(self.solutionPath[currentNode].keys())[-1]     # H is the nearest hospital of the curre
        for neighbourNode in self.adj_list[currentNode]:      # for the neighbour nodes, update their
            if self.solutionPath[neighbourNode]=={}:        # e.g. ['567']+['678','345','0'], so ['567
                self.solutionPath[neighbourNode][H]=[neighbourNode]+self.solutionPath[currentNode][H]
                self.solutionDistance[neighbourNode][H]=len(self.solutionPath[neighbourNode][H])-1
                Q.put(neighbourNode)
```

Figure 2

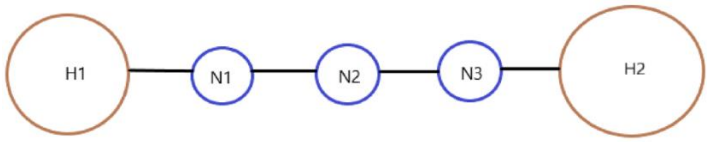

Figure 3

First algorithm

Given the simple graph above, the first algorithm, solve, updates the shortest path for hospitals H1 and H2 as ['H1'] and ['H2'] respectively. H1 and H2 are put in Queue and BFS begins. Next, the algorithm updates the shortest path for N1 as simply itself ['N1'] plus the path of the previous node, thus ['N1']+['H1'] =['N1', 'H1']. Next, the algorithm updates the shortest path for N3 as simply ['N3']+['H2'] =['N3', 'H2']. Next, the algorithm updates the shortest path for N2 as simply ['N2']+['N1', 'H1']= ['N2','N1', 'H1']. Lastly, N2 is visited again from N3, but it already has a shortest path solution, hence solutionPath is not updated and the BFS traversal down this path stops. Note that, the solutionDistance is simply length of the path minus one.

By applying BFS on all hospitals simultaneously in road network, the algorithm can generate a nested dictionary of shortest paths and distance from a node to its nearest hospital in a single traversal. Figure v shows the nested dictionary of shortest paths stored in solutionPath. For instance, the nearest hospital from node '292133' is '292058', with the shortest path underlined in brown. SolutionDistance stores the nested dictionary of shortest distances. For instance, '292697' has a distance of 3 to hospital '292724'.

```
'292133': {'292058': ['292133', '292059', '292058']}
'292139': {'292058': ['292139', '292059', '292058']}
'292141': {'292058': ['292141', '292059', '292058']}
```

Figure 4

```
'292697': {'292724': 3},
'292730': {'292724': 3},
'296812': {'292724': 3},
```

Figure 5

Proof of correctness

Intuitively, the nearest hospital from N3 is H2. However, suppose that the algorithm found that the shortest path is from N3 to H1. However, this contradicts the 'breadth-wise' traversal of BFS. **Since BFS began simultaneously on all hospitals, the breadthwise traversal that began from H2 will always reach N2 first.** Therefore, the algorithm would assign N2's shortest path to H2 before a path could be mapped to H1.

Second algorithm

Figure 6

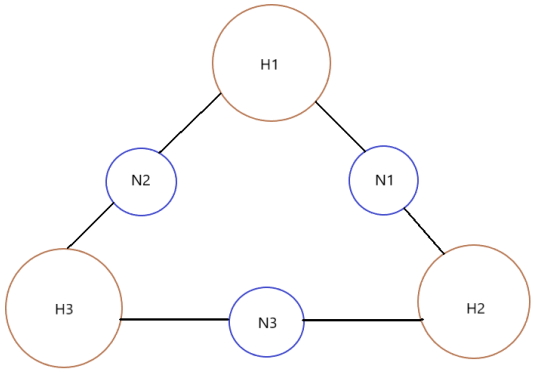

Figure 7

The second algorithm finds the distances of top k nearest hospitals for every node by applying BFS on each hospital consecutively in a for loop. Suppose K=2. BFS is first applied on H1. First, distance of H1 to itself is updated as 0 in topKDistances. Next, N1 and N2 are visited next and their distances to H1 **are the distance from the previous node plus one** i.e. 0+1= 1. Next, H2 and H3 are visited and their distances to H1 is 1+1 = 2 and lastly, distance from N3 to H1 is 2+1=3. A similar process is run for H2 as well. By the end, each node will have 2 distances attached to it.

When BFS is executed on H3, the shortest distance from N2 and N3 to H3 is 0+1 = 1. However, from the previous run of BFS, N2 has a distance of 3 to H2 and 1 to H1 tied to it. Since 1 < 3, the distance from N2 to H2 is deleted and replaced with distance 1 to H3. The same applies for N3. The distance from H1 and H2 to H3 is 1+1=2. Both H1 and H2 already has a distance of 0 to itself and distance 2. Since the **existing distances are less than or equal to the new distance** to H3, no update to topKDistances is needed and BFS does not continue traversing down.

The following figure shows topKDistances, a nested dictionary. For example, the top 2 distances for node '59999' will be 19 and 58 to hospitals '60590' and '232827' respectively.

```
In [48]: # top K distances are found in .topKDistances instance variable
         Finder.topKDistances
```

```
'59168': {'60590': 19, '232827': 61},
'60143': {'60590': 19, '232827': 61},
'60159': {'60590': 19, '232827': 60},
'76243': {'60590': 19, '232827': 61},
'76244': {'60590': 19, '232827': 61},
'59912': {'60590': 19, '232827': 58},
'59999': {'60590': 19, '232827': 58},
'60002': {'60590': 19, '232827': 58},
'59131': {'60590': 19, '232827': 54},
'59910': {'60590': 19, '232827': 56},
'59141': {'60590': 19, '232827': 59},
```

Figure 8

3 Time complexity

First algorithm *(.solve())*

Let C0 be the total time cost of assigning the solutionPath and solutionDistance for each node, putting and getting the nodes from Q. Since the algorithm traverses every node once, given V nodes, there will be a time cost of C0V. Let C1 be the time cost of comparison to check if a node already has a solution. Since the algorithm traverses every edge once, given E edges, there will be a time cost of C1E. In conclusion, the time complexity in all cases is:

$$C_1|E| + C_0|V| = O(|V|+|E|).$$

Second algorithm *(.topK())*

As shown in Figure 6, let C2 be the total time cost of assigning values to topKDistances, labelling True for explored nodes, putting and getting nodes from Q. Let C4 be the time cost of comparing the Boolean value of a node with False to check whether it has been explored. In the first k rounds of BFS, every node and edge are traversed, so there is a time cost of C2Vk + C4kE.

Let C3 be the time cost of deleting existing solutions from topKDistances. For the remaining h-k rounds of BFS there is an additional time C3 spent on each node. Since the algorithm stops traversal down a path when there is no need, only a subset of nodes v' and edges e' are traversed. Therefore, there is a time cost of (C2+C3) (h-k) v'+ C4(h-k) e' in the remaining h-k rounds of BFS, where h is the total number of hospitals. Therefore, the time complexity is:

$$C2|V|k + (C2+C3)(h-k)|v'|+ C4k|E| + C4(h-k)|e'| \leqslant (C2+C3)(h)|V| +C4h|E| = O(h(|V|+|E|))$$

where v' and e' are subsets of V and E respectively.

4 Empirical Analysis

In this session, we are interested to study the effect of the parameter number of hospitals (h) on our .solve() algorithm, the effect of the parameter top k-th nearest hospital on our .topK() algorithm, as well as the effect of both parameter h and k on the .topK() algorithm.

First algorithm *(.solve())*

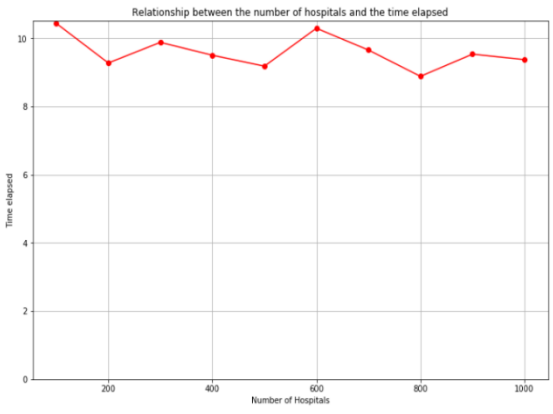

Figure 9

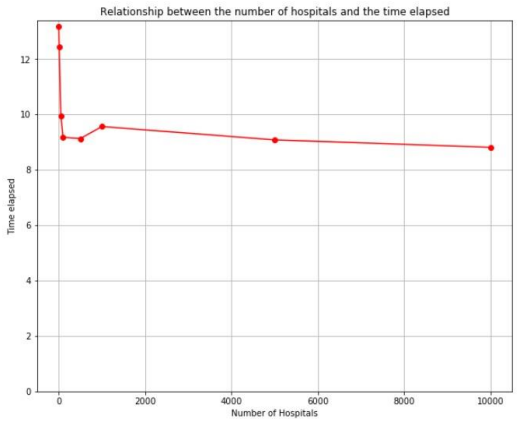

Figure 10

The number of hospitals in Figure 9 ranges from h=100 to h=1000. The graph turns out to be quite choppy, and we cannot really extract any useful information out of it.

In Figure 10, we changed the range of number of hospitals to now varies from h=1 to h=10000. The result plotted on a graph appears to be similar to a negative exponential graph. Based on this graph, we can deduce that the larger the number of hospitals, the lesser the time taken for the algorithm *.solve()* to run.

Second algorithm *(.topK())*

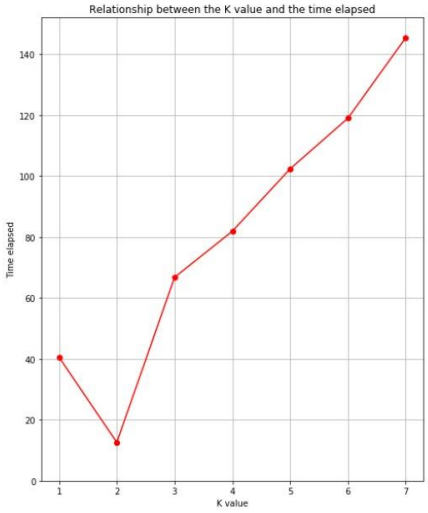

Figure 11

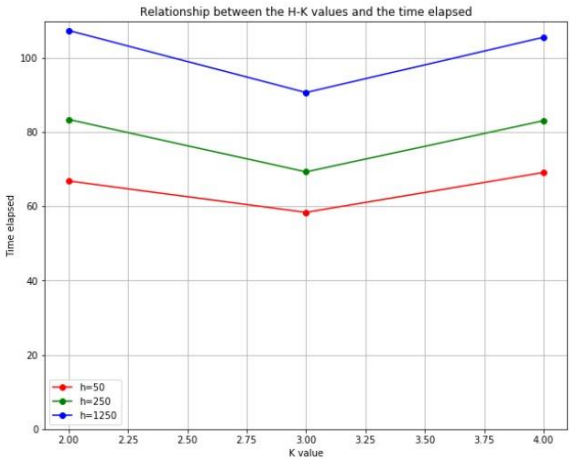

Figure 12

To explore the relationship between K value and the time elapsed, in Figure 11 we tested our algorithm for values of k from k=1 to k=7. The results show that, except for the anomaly at k=2, the result is a positive linear graph. The runtime of the algorithm *.topK()* increases linearly with the increase in the value of k.

To continue our study on the effect of h and k on the *.topK()* algorithm, as illustrated in Figure 12, a simple 3x3 matrix experiment was done with k value and number of hospitals (h). The number of hospitals, h is taken as 50, 250 and 1250 for the analysis whereas k took the value of 2, 3 and 4.

As we increase the number of hospitals (h), the whole graph shifts upwards indicating the longer runtime of the algorithm. We can't conclusively draw a solid conclusion for the effect of k on the algorithm, but based on our previous analysis, we can make smart deduction that the runtime of the algorithm also increases linearly with the value of k if we are to extrapolate the k value to a higher value.

5 Conclusion

The original BFS algorithm that was first proposed in 1945 can be quite slow if run multiple times through the graph without optimisation. However, it can be adapted to multiple situation such that only a single run is enough. In our first algorithm, we modified and improved the algorithm to the point where a single run through the graph is sufficient to determine the shortest distance and path. We achieved this by making the BFS done simultaneously at each hospital. Should we have not done that, the algorithm would be relatively slower and can be very time consuming when involving a graph of extremely large size. By our analysis, the time taken for the algorithm to complete its search decreases in a negative exponential as more hospitals indicates that more nodes are hospitals and that increases the likelihood that the immediate neighbour of each node is a hospital.

The second algorithm considered a brute force method (although slightly improved). While we are able to alter the algorithm towards solving the problems laid out to us in this project, the algorithm devised is considered slow and inefficient as proven by our empirical analysis. While it is expected, the increased number of hospitals, h and k value increases the time lapsed linearly for this algorithm when it is performing the top-K nearest hospital search. As the number of k and h increases, more runs are required for the search to be completed which results in a long search time.

6 Reference

[1] T. Cormen & D. Balkcom Analysis of breadth-first search (n.d). Retrieved from https://www.khanacademy.org/computing/computer-science/algorithms/breadth-first-search/a/analysis-of-breadth-first-search

[2] PyTech Vision. (2020, April 6th). 2.1 BFS: Breadth First Search Implementation in Python | Graph Data Structure. Retrieved from https://www.youtube.com/watch?v=PQhMkmhYZjQ&t=511s

Statement of Contribution

In this project, the workload was divided among the five of us following the breakdown below:

Chang Heen Sunn – Conducted the empirical analysis of the two algorithms applied in this project.

Luo Wenyu – Utilised the random graph generator for shortest distance and path problem and output them in a text file.

Ooi Wei Chern – Prepared the presentation slides and wrote the project rerport.

Saw William Joseph – Investigated the time complexities of the two algorithms used.

Tan Ching Fhen – Modified and written the code for the two algorithms used.