CZ2001: Project 2 Report

Goh Wei Pin, Tan Keng Kai Luke, Lynn Enhua Masillamoni, Jonathan Chang Ji Ming

1.0 Introduction

Breadth-first search (BFS) is an algorithm for traversing graph data structures. It starts at a selected node, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. In our case study, our graph represents a city, and given a node, we are tasked to find k nearest nodes which have been labeled as hospitals.

2.0 Data Structures

Data Structures used are Queue: List, Visited: List and Adjacency List: Dictionary.

We chose to use an adjacency list over an adjacency matrix due to the graph being sparse and having few edges compared to its number of nodes, as seen from the sample real road network graph.

Time complexity

- In an adjacency matrix, for every known node A, it will take O(|V|) to search for its neighbours, where |V| is the total number of vertices. Thus, to find the path from node A to the nearest hospital node, the time complexity is $O(|V|^2)$. As we would be running the BFS function |V| number of times to check every node, the total time complexity for the entire program will be $O(|V|^3)$.
- The access time for the adjacency list is at most O(|V|+|E|) per node. As we run the BFS function |V| number of times, the time complexity will be of $O(|V|^2)$ since |V| is of the same order as |E|.

Space complexity

- The space complexity for the adjacency matrix is of $O(|V|^2)$. This would be extremely resource intensive, especially when used on large scale real road network graphs.
- However, the space complexity for the adjacency list is only O(|V| + |E|), though the worst case where every single node is connected to each other causes the space complexity to increase to $O(|V|^2)$.

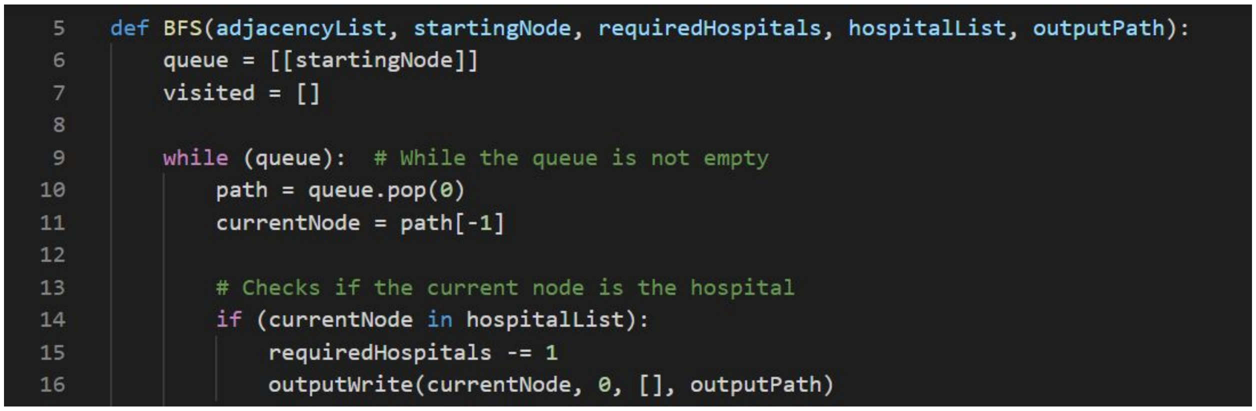
Hence, we decided to use an adjacency list for its superior time and space complexity.

3.0 Relevant Functions

We will first explain the relevant functions throughout our entire program:

1. generateRandGraph(): Generates a random graph from the ‘networkx’ library, by selecting a specific n (number of nodes) and p (probability of edge between two nodes). The graph is generated and printed into a text file located at the path provided.
2. getGraph(): Retrieves the selected graph from (1), and puts it into the form of an adjacency list (python dictionary).
3. readHospitals(): Retrieves the list of nodes that are hospitals from the hospitals text file.
4. outputWrite(): Outputs the results of the program in a text file.
5. wipeTextFile(): Deletes all the contents of the output text file to ensure that the file will not have old and irrelevant data.

Lastly, the most important function in our program is the BFS() function. <u>We use the same BFS algorithm to achieve all 4 tasks that were presented to us as it fulfills all the requirements.</u>

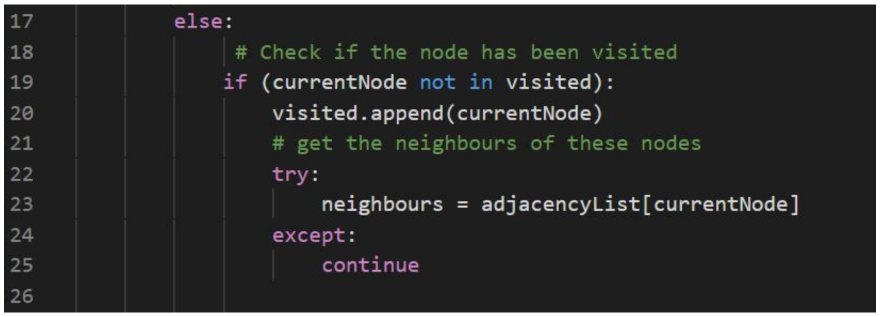


```python
5   def BFS(adjacencyList, startingNode, requiredHospitals, hospitalList, outputPath):
6       queue = [[startingNode]]
7       visited = []
8   
9       while (queue):  # While the queue is not empty
10          path = queue.pop(0)
11          currentNode = path[-1]
12  
13          # Checks if the current node is the hospital
14          if (currentNode in hospitalList):
15              requiredHospitals -= 1
16              outputWrite(currentNode, 0, [], outputPath)
```

In our BFS function, we take in a variety of parameters:

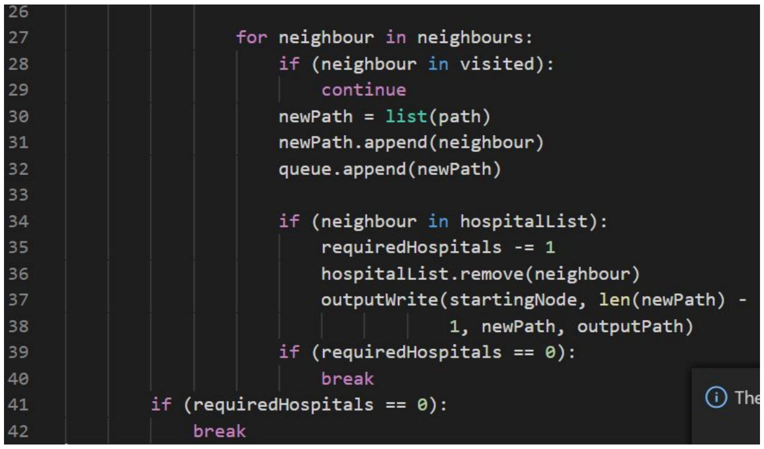
1. adjacencyList: the graph, in the form of an adjacency list
2. startingNode: the starting node, for which we want to find *k* nearest hospitals to it
3. requiredHospitals: the variable that stores the *k* nearest hospitals we want to find, the user is able to input value k
4. hospitalList: this is the list of hospitals in the graph
5. outputPath: this is the path which allows us to print out our results in the output text file

We start by placing the startingNode in the queue, and declaring an empty list for the visited nodes. Using a while loop, we pop the first element from the queue into the path and label it as the currentNode. Lines 14-16 checks if the currentNode is already a hospital, if it is a hospital, it deducts 1 from the number of nearest hospitals we need to find, and uses the outputWrite function to print out the path in the output text file.



```python
17          else:
18              # Check if the node has been visited
19              if (currentNode not in visited):
20                  visited.append(currentNode)
21                  # get the neighbours of these nodes
22                  try:
23                      neighbours = adjacencyList[currentNode]
24                  except:
25                      continue
26  
```

If the currentNode is not a hospital, we check if it is within the list of visited nodes. If it has not been visited, we add it into the visited list, and get the neighbours of these nodes. We use exception handling, such that if there are no neighbours, this portion of code is skipped.

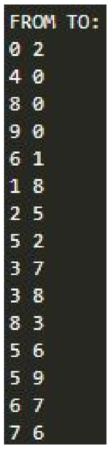


```python
26  
27                  for neighbour in neighbours:
28                      if (neighbour in visited):
29                          continue
30                      newPath = list(path)
31                      newPath.append(neighbour)
32                      queue.append(newPath)
33  
34                      if (neighbour in hospitalList):
35                          requiredHospitals -= 1
36                          hospitalList.remove(neighbour)
37                          outputWrite(startingNode, len(newPath) -
38                                      1, newPath, outputPath)
39                      if (requiredHospitals == 0):
40                          break
41          if (requiredHospitals == 0):
42              break
```

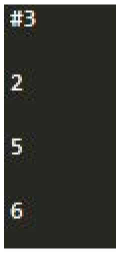
Lines 28-40 will execute for each neighbour found in Line 23. If the neighbour has been visited, we skip this neighbour and move on to the next. If not, a newPath is set to be the current path, the neighbour is appended and the queue is updated. The queue data structure is a list of the current path taken to the respective nodes where the node at index -1 is the node we are looking at. If the neighbour is in the list of hospitals, we deduct 1 from the number of nearest hospitals we need to find, and remove the node from the list of hospitals. Then, the output file is updated. This portion is looped until we have found all *k* nearest hospitals.

4.0 Test Cases and Time Complexity

For the following test cases, the graph (left), is randomly generated where n = 10, p = 0.3, and a list of hospitals (right) is used.

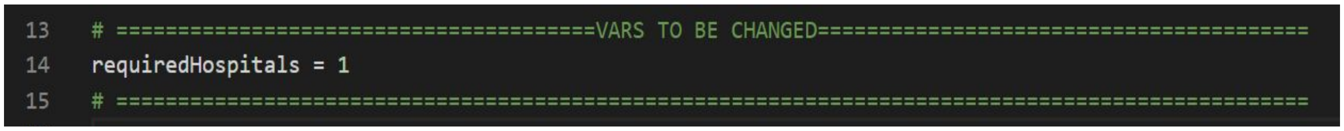


```
FROM TO:
0 2
4 0
8 0
9 0
6 1
1 8
2 5
5 2
3 7
3 8
8 3
5 6
5 9
6 7
7 6
```



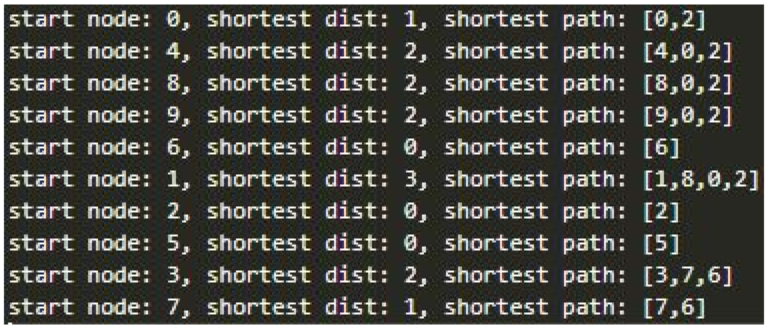
```
#3

2

5

6
```

(a & b) Design an algorithm for computing the distance from each node in G to its nearest hospital, its time complexity should not depend on the total number of hospitals h. Output the distance and the shortest path for each node to a file.
To find the nearest hospital from each node, we set the requiredHospitals variable to 1.



```
# ======================================VARS TO BE CHANGED=========================================
requiredHospitals = 1
# =================================================================================================
```

When we run this interface, the output will be as follows.

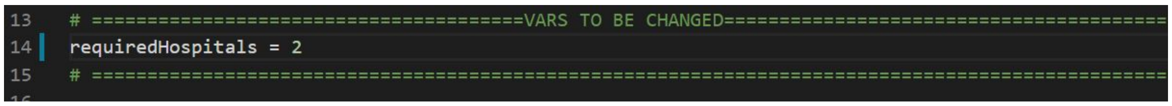


```
start node: 0, shortest dist: 1, shortest path: [0,2]
start node: 4, shortest dist: 2, shortest path: [4,0,2]
start node: 8, shortest dist: 2, shortest path: [8,0,2]
start node: 9, shortest dist: 2, shortest path: [9,0,2]
start node: 6, shortest dist: 0, shortest path: [6]
start node: 1, shortest dist: 3, shortest path: [1,8,0,2]
start node: 2, shortest dist: 0, shortest path: [2]
start node: 5, shortest dist: 0, shortest path: [5]
start node: 3, shortest dist: 2, shortest path: [3,7,6]
start node: 7, shortest dist: 1, shortest path: [7,6]
```

The worst case time complexity for each BFS function call is O(|V|+|E|), where |V| is the number of vertices and |E| is the number of edges, where the function has to search through the entire graph to find the closest hospital.Since we assumed the graph is sparse, where |E| is on the same order as |V|, this allows us to reduce the complexity of each BFS function call to O(|V|). Since the function is called |V| times to search for the closest hospital to each node, the overall worst case time complexity will be O(|V|^2) for |V| nodes. On the other hand, the best case time complexity for each BFS function call is O(1), where the node is already the hospital itself. Since the function will still get called |V| times, the overall best case time complexity will be O(|V|). Thus, the time complexity is purely dependent on |V| and |E|, and is not dependent on the number of hospitals h.

(c) Instead, we are interested in finding the distances to top-2 nearest hospitals from each node. Revise your algorithm in (b) to accommodate this new requirement.

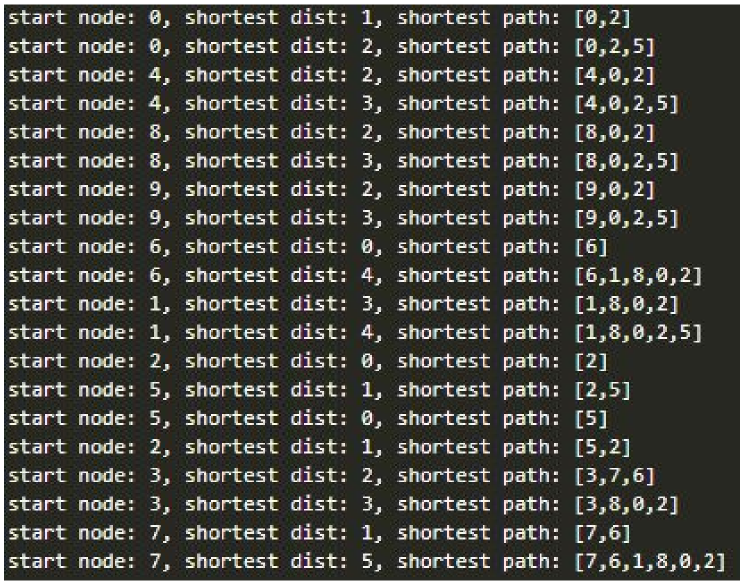
To fulfill the requirement, we set the requiredHospitals variable to be equal to 2 and run the algorithm.



```
# ====================================VARS TO BE CHANGED=====================================
requiredHospitals = 2
# ============================================================================================
```

When we run this interface,

the output will be as follows.



```
start node: 0, shortest dist: 1, shortest path: [0,2]
start node: 0, shortest dist: 2, shortest path: [0,2,5]
start node: 4, shortest dist: 2, shortest path: [4,0,2]
start node: 4, shortest dist: 3, shortest path: [4,0,2,5]
start node: 8, shortest dist: 2, shortest path: [8,0,2]
start node: 8, shortest dist: 3, shortest path: [8,0,2,5]
start node: 9, shortest dist: 2, shortest path: [9,0,2]
start node: 9, shortest dist: 3, shortest path: [9,0,2,5]
start node: 6, shortest dist: 0, shortest path: [6]
start node: 6, shortest dist: 4, shortest path: [6,1,8,0,2]
start node: 1, shortest dist: 3, shortest path: [1,8,0,2]
start node: 1, shortest dist: 4, shortest path: [1,8,0,2,5]
start node: 2, shortest dist: 0, shortest path: [2]
start node: 5, shortest dist: 1, shortest path: [2,5]
start node: 5, shortest dist: 0, shortest path: [5]
start node: 2, shortest dist: 1, shortest path: [5,2]
start node: 3, shortest dist: 2, shortest path: [3,7,6]
start node: 3, shortest dist: 3, shortest path: [3,8,0,2]
start node: 7, shortest dist: 1, shortest path: [7,6]
start node: 7, shortest dist: 5, shortest path: [7,6,1,8,0,2]
```

In this case, the worst case time complexity is the same as in part (a & b), which is O(|V|^2). The best case time complexity for each time the BFS function is called is O(2k-1) which is equivalent to O(3) since k=2. This is because the algorithm will search through at least 2 nodes and 1 edge to come to this conclusion. Since the function will still get called |V| times, the overall best case time complexity will also be equals to O(|V|), since 2k-1 is a constant.

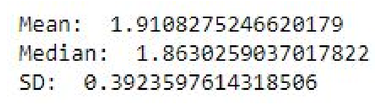
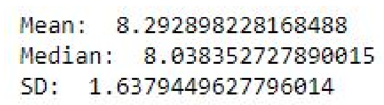
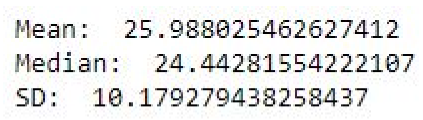
(d) Propose an algorithm that works generally for computing the distances from each node to top-*k* nearest hospitals for an input value of *k*.

Similar to parts (a) to (c), to find the distances from each node to top-k nearest hospitals, we set the requiredHospitals variable to *k* to find the desired output. The time complexities do not differ from that explained in the previous parts.

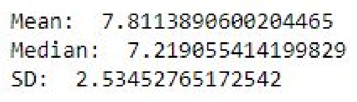
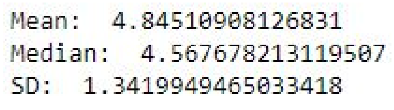
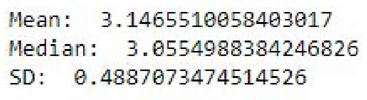
5.0 Empirical Study and Findings

For the following experiments, we generated a graph of 1000 nodes with the probability of an edge being 0.01 using our generateRandGraph function. Each experiment is run 100 times and the mean, median and standard deviation(SD) is recorded. For each iteration of the 100 runs, we randomly generate the hospital nodes according to h, the number of hospitals required.

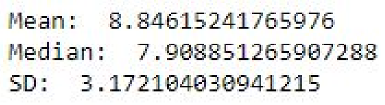
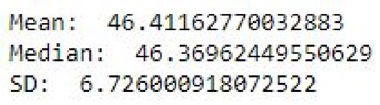
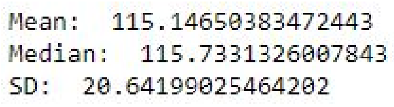
The table below shows the statistics on the experiment we ran 3 times where h is a constant 6, and k is varied, taking values 1, 3 and 5. Generally, the algorithm takes longer when they are required to find more hospitals. Additionally, the standard deviation and variance is also larger as k increases. The algorithm performs faster on smaller numbers of k due to the algorithm stopping earlier. This is because the algorithm would have to traverse through more of the graph as the number of nearest hospitals that it has to find increases.

h=6, k=1	h=6, k=3	h=6, k=5
 Mean: 1.9108275246620179 Median: 1.8630259037017822 SD: 0.3923597614318506	 Mean: 8.292898228168488 Median: 8.038352727890015 SD: 1.6379449627796014	 Mean: 25.988025462627412 Median: 24.44281554222107 SD: 10.179279438258437

The results below show the time taken for the algorithm to find the shortest path from each node to the nearest k hospitals. The experiment is run 3 times where k is a constant 2 and h is varied, taking values 4, 6 and 10. Generally, the algorithm is able to find the shortest path faster when h increases because the probability that the visited node is a hospital increases. The standard deviation remained relatively low, showing the consistency of our algorithm. The decrease in time taken could be due to the higher number of hospitals resulting in a higher probability of a nearby node being a hospital, thus reaching the required number of hospitals faster. However, it is important to take note that h will not affect the best and worst case time complexities as explained above in our time complexity analysis.

h=4, k=2	h=6,k=2	h=10, k=2
 Mean: 7.8113890600204465 Median: 7.219055414199829 SD: 2.53452765172542	 Mean: 4.84510908126831 Median: 4.567678213119507 SD: 1.3419949465033418	 Mean: 3.1465510058403017 Median: 3.0554988384246826 SD: 0.4887073474514526

The results below show the time taken for the algorithm to find the shortest path from each node to the nearest k hospitals when we vary the density of the graph. This is done by adjusting p, the probability of an edge existing between 2 nodes, in the generateRandGraph() function. The values of h and k are kept constant at 4 and 2 respectively. The results show a trend that the algorithm takes a longer running time to find the shortest path in a dense graph as compared to sparse graphs.

Probability of edge = 0.01	Probability of edge = 0.2	Probability of edge= 0.5
 Mean: 8.84615241765976 Median: 7.908851265907288 SD: 3.172104030941215	 Mean: 46.41162770032883 Median: 46.36962449550629 SD: 6.726000918072522	 Mean: 115.14650383472443 Median: 115.7331326007843 SD: 20.64199025464202

As we increased the density of the graphs, we can see that there is an increase in the time taken. We previously concluded the time complexity to be O(|V|^2) for a sparse graph where |V| is of the same order as |E|. As we increase the density of the graph, we are increasing the total number of edges while keeping the number of nodes the same.

As the graph becomes more dense, the advantages of the adjacency list diminishes in comparison to an adjacency matrix, as the number of nodes linked from the parent node comes close to |V|. Hence, the run time increases drastically. In the case of a full graph, the number of edges per node is equals to |V|-1. The time complexity will be equals O(|V|+|E|) = O(|V|+|V|(|V|-1)) = O(|V|^2) per node. As BFS will be run |V| times which results in the overall time complexity to be O(|V|^3).

6.0 Real World Applications

We previously decided to use adjacency list over the adjacency matrix due to the advantage in time complexity. This is only true as we made the assumption on the graph being sparse where |V| and |E| are of similar orders. This assumption is very valid in our problem, due to the nature of checkpoints, roads and junctions in our roadmap, we can expect each node to be connected to only a limited number of other nodes, thus resulting in a very sparse graph. However, this assumption may not hold true in every problem presented to us. If we were to change the context to a social networking graph, which results in a much denser due to its nature, we can no longer make the same assumption. In a situation like this, the benefits of using an adjacency list over an adjacency matrix will then start to diminish.

Contributions
Tan Keng Kai, Luke →Writing of report, analysis of algorithms
Goh Wei Pin → Implementation of algorithms to python code, analysis of algorithms
Lynn Enhua Masillamoni → Implementation of algorithms to python code, writing of report
Jonathan Chang → Implementation of algorithms to python code, analysis of algorithms