# NANYANG TECHNOLOGICAL UNIVERSITY

# SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

## CZ2001/CE2001 SE1 ALGORITHMS

## PROJECT 2


**Celeste Chin Khar Min (U1921916B)**

**Elayne Tan Hui Shan (U1921730C)**

**Seow Jing Hng Aloysius (U1920159K)**

**Soh Qian Yi (U1922306C)**

**Introduction**

In our implementation, we used the adjacency list to represent the undirected, unweighted, Graph G. We utilized the Breadth-First Search (BFS) algorithm to locate the shortest path for each node to a location (i.e Hospital). BFS systematically explores the edges directly connected to the chosen node before visiting further vertices.

**Analysis of Algorithm and Asymptotic Time Complexity**

In finding our asymptotic time complexity, we mainly focus on the time complexity given by the graph algorithm and assume that the time complexity given by the file reading, file processing and graph creating processes are negligible compared to the graph algorithm's time complexity.

<u>GraphAlgorithm</u>

<table>
<tr>
<td>
<pre>
System.setOut(o);
for (int i=0; i<edges.size();i++) {
    searchForHosp(edges, i, hospitals, 1);
}
</pre>
</td>
<td>This for loop answers part (a) where we create an output file including every node, its nearest hospital information and information regarding the distance to the nearest hospital. This for loop runs through all the edges and hence gives a time complexity of O(E).</td>
</tr>
</table>

<u>searchForHospital</u>

<table>
<tr>
<td>
<pre>
public static void searchForHosp(ArrayList<ArrayList<Integer>> edges, int start,
        int[] hospitals, int k) throws FileNotFoundException
    {
    System.out.println("Node " + (start));
    int[] predecessor = new int[edges.size()];
    int[] dist = new int[edges.size()];

    int[] destination_hospitals = BFS(edges, start, hospitals, k, predecessor, dist);
</pre>
</td>
<td>This method calls the BFS method and stores the results in an array, destination_hospital.</td>
</tr>
<tr>
<td>
<pre>
// find route for every hospital that start managed to reach in range of k
for (int i = 0; i < destination_hospitals[k]; i++) {
    System.out.println("Hospital Node ID: " + destination_hospitals[i]);
    // forming route
    LinkedList<Integer> route = new LinkedList<Integer>();
    curr = destination_hospitals[i];
    route.add(curr);
    while (predecessor[curr] != -1) {
        route.add(predecessor[curr]);
        curr = predecessor[curr];
    }

    // printing out distance
    System.out.println("Shortest distance is: " + dist[destination_hospitals[i]]);

    // printing out route
    System.out.println("Route is:");
    for (int j = route.size()-1; j >= 0; j--) {
        System.out.print(route.get(j) + " ");
    }
</pre>
</td>
<td>The method then uses a for loop to print the hospital node ID, shortest distance and exact route from the selected start node to the top-k nearest hospital.</td>
</tr>
</table>

BFS

```java
private static int[] BFS(ArrayList<ArrayList<Integer>> edges, int start,
        int[] hospitals, int k, int[] predecessor, int[] dist)
    {
    int[] result = new int[k+1];
    // last element is a counter of no. of destination hospitals, stop when reach k
    result[k] = 0;

    LinkedList<Integer> queue = new LinkedList<Integer>();
    boolean[] visited = new boolean[edges.size()];
    // initialise
    for (int i = 0; i < edges.size(); i++) {
        visited[i] = false;
        dist[i] = Integer.MAX_VALUE; //infinity
        predecessor[i] = -1;
    }
```

This method uses a queue to monitor which vertices to visit next. It initializes all vertices as unvisited so visited[i] for all i is false and as no path is yet constructed, dist[i] for all i set to infinity.

The initialising for loop runs through every edge and gives a time complexity of O(E) where E = number of edges.

```java
    // for starting point
    visited[start] = true;
    dist[start] = 0;
    queue.add(start);

    while (!queue.isEmpty()) {
        int curr = queue.remove();
        for (int j = 0; j < edges.get(curr).size(); j++) {
            if (visited[edges.get(curr).get(j)] == false) {
                visited[edges.get(curr).get(j)] = true;

                // curr's dist + 1 = curr's "child"'s dist
                dist[edges.get(curr).get(j)] = dist[curr] + 1;
                predecessor[edges.get(curr).get(j)] = curr;
                queue.add(edges.get(curr).get(j)); // add curr's "child" to queue

                // checking if curr's "child" is destination (hospital)
                for (int h = 0; h < hospitals.length; h++)
                    if (edges.get(curr).get(j) == hospitals[h]) {
                        result[result[k]] = hospitals[h];
                        result[k] += 1;

                        // destination has k number of hospitals, stop
                        if (result[k] == k) return result;
```

In the *while* loop and first *for* loop, each edge is processed once for a total cost of O(|E|) where E is the number of edges. Then, it runs through all (if not almost all - for cases where certain edges are not accessible from the starting node) vertices and gives a time complexity of O(|V|).

Each vertex is visited at most one time as the distance is null at the start and each vertex is enqueued and dequeued at most one time. Since we examine the edges incident on a vertex only when we visit it, each edge is examined at most twice, once for each of the vertices it's incident on. Thus, BFS spends O(|V|+|E|) time visiting vertices.

The second *for* loop checks if the node is a hospital and runs through the hospitals array until it finds the node to be a hospital or if it runs through the entire array (which means the node is not a hospital). Hence, the maximum time the loop will run each time is h, where h is the number of hospitals. However, h is much smaller than E (especially in real road networks whereby the number of hospitals is normally much smaller) and hence the time complexity is negligible.

When working with large graphs (e.g. real road network graphs), to find nodes that are distance *d* from the start node (measured in number of edge traversals), BFS takes O(b^(d+1)) time and memory, where *b* is the branching factor of the graph.

**Design of Algorithm**

<u>Assumptions</u>

- Value of k should be less than or equals to number of hospitals (h) in the Hospital Information File
- The output will only show relevant hospital nodes (h) based on the Node IDs in the Node File being used {e.g. if k=4 nearest number of hospitals chosen (3,10,19,34) and the Node File only contains Node IDs up to 10, the output will only print the nearest 2 hospitals (3,10) since (19,34) are irrelevant hospital nodes}
- Number of Hospitals (h) is very negligible compared to the number of Edges (E) for large roadmap cases.
- There would be different run times for different computers used.

**Experiments**

Our code allows us to answer part (a)-(d) of the task requirements.

- For part (a), entering ‘1’ will output the distance and the shortest path for each node to a file. Outputs for part (c) or (d) will be generated by entering ‘2’ or ‘3’ respectively.
- Part (b) is skipped as the time complexity for our code does not depend on the total number of hospitals h.
- Outputs for parts (b), (c) and (d) will be written in a text file - *output.txt*.

1. **Using our random generated graph’s dataset - *random.txt*, and *rando.txt*, let starting node = 1:**

| | | |
|---|---|---|
| Hospital Node ID: 3<br>Shortest distance is: 3<br>Route is:<br>5 1 0 3<br><br>For Node 6,<br>Hospital Node ID: 9<br>Shortest distance is: 1<br>Route is:<br>6 9<br><br>For Node 7,<br>Hospital Node ID: 3<br>Shortest distance is: 1<br>Route is:<br>7 3<br><br>For Node 8,<br>Hospital Node ID: 7<br>Shortest distance is: 1<br>Route is:<br>8 7 | Nearest 2 hospitals to Node 1 are:<br>For Node 1,<br>Hospital Node ID: 3<br>Shortest distance is: 2<br>Route is:<br>1 0 3<br>Hospital Node ID: 9<br>Shortest distance is: 2<br>Route is:<br>1 0 9 | Nearest 4 hospitals to Node 1 are:<br>For Node 1,<br>Hospital Node ID: 3<br>Shortest distance is: 2<br>Route is:<br>1 0 3<br>Hospital Node ID: 9<br>Shortest distance is: 2<br>Route is:<br>1 0 9<br>Hospital Node ID: 4<br>Shortest distance is: 2<br>Route is:<br>1 2 4<br>Hospital Node ID: 7<br>Shortest distance is: 2<br>Route is:<br>1 2 7 |
| *Partial output for part (a)* | *Output for part (c)* | *Output for part (d) when k = 4* |

2. **Using a real road network graph - *roadNet-CA.txt*, and *txtfile.txt*, let starting node = 1:**

```
Node 406727
Hospital Node ID: 446637
Shortest distance is: 51
Route is:
406727 406728 406734 406741 406742 406717 4

Node 406728
Hospital Node ID: 446637
Shortest distance is: 50
Route is:
406728 406734 406741 406742 406717 406718 1

Node 406729
Hospital Node ID: 446637
Shortest distance is: 52
Route is:
406729 406727 406728 406734 406741 406742 4

Node 406730
Hospital Node ID: 446637
Shortest distance is: 52
Route is:
406730 410843 406733 406734 406741 406742 4
```

*Partial output for part (a)*

```
Nearest 2 hospitals to Node 1 are:
For Node 1,
Hospital Node ID: 5
Shortest distance is: 2
Route is:
1 6 5
Hospital Node ID: 133
Shortest distance is: 5
Route is:
1 385 386 387 139 133
```

*Output for part (c)*

```
Nearest 5 hospitals to Node 1 are:
For Node 1,
Hospital Node ID: 5
Shortest distance is: 2
Route is:
1 6 5
Hospital Node ID: 133
Shortest distance is: 5
Route is:
1 385 386 387 139 133
Hospital Node ID: 132
Shortest distance is: 6
Route is:
1 385 386 387 139 133 132
Hospital Node ID: 426
Shortest distance is: 7
Route is:
1 0 469 380 183 182 181 426
Hospital Node ID: 190
Shortest distance is: 7
Route is:
1 385 384 468 442 441 191 190
```

*Output for part (d) when k = 5*

**Empirical Study -** Effects of h and k on the performance of various algorithm

Using roadNet-CA.txt and random_hospital.txt,

| k | Time Taken in milliseconds |
|---|---|
| 10 | Execution time in milliseconds : 49 |
| 100 | Execution time in milliseconds : 212 |
| 200 | Execution time in milliseconds : 245 |

| k | h (number of hospitals) | Time Taken in milliseconds |
|---|---|---|
| 2 | 2 | 30 |
| 2 | 3 | 46 |
| 2 | 7 | 33 |

In conclusion, as *k* increases with constant *h*, time taken also increases while there is no clear relationship when *k* is kept constant. Hence, our time complexity is dependent on *k*.

**References**

https://www.geeksforgeeks.org/shortest-path-unweighted-graph/
https://en.m.wikipedia.org/wiki/Breadth-first_search
https://www.khanacademy.org/computing/computer-science/algorithms/breadth-first-search/a/analysis-of-breadth-first-search

**Statement of Contribution**

Code (Research and Analysis):
Celeste Chin Khar Min (U1921916B)
Elayne Tan Hui Shan (U1921730C)
Seow Jing Hng Aloysius (U1920159K)
Soh Qian Yi (U1922306C)

Report (Documentation and Analysis):
Celeste Chin Khar Min (U1921916B)
Elayne Tan Hui Shan (U1921730C)
Seow Jing Hng Aloysius (U1920159K)
Soh Qian Yi (U1922306C)

Everyone in the team contributed equally as we attended every group meeting and worked effectively as a team, with good communication and camaraderie.