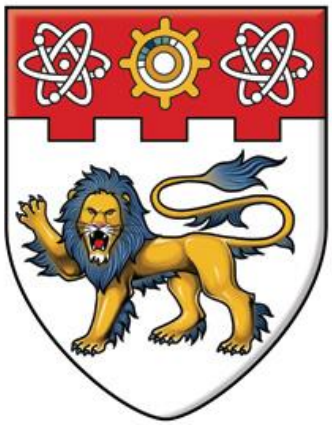

NANYANG TECHNOLOGICAL UNIVERSITY

COURSE CODE	**:**	**CZ2001**
COURSE NAME	**:**	**ALGORITHMS**
GROUP MEMBERS	**:**	

1. **Alvin Lee Yong Teck (U1620768F)**
2. **Htet Naing (U1620683D)**
3. **Liu Jia Hao, Bryan (U1621401C)**
4. **Lim Kian Hock, Bryan (U1620949L)**
5. **Luke Lee (U1620850A)**
6. **Zacharias Tay (U1620760H)**

LAB GROUP	**:**	**SS4**
COURSEWORK	**:**	**EXAMPLE CLASS 4 REPORT**
TITLE	**:**	**APPLICATION OF BFS TO FLIGHT SCHEDULING**

Contents

Introduction to Breadth First Search (BFS)

BFS starts at the first node of a graph, and explores the neighbouring nodes before moving on to the next level of neighbours, finding every other neighbour that is at the distance **k** before exploring any vertices that are distance **k+1**. There is no order involved when we explore the neighbouring nodes.

Each node can only be visited **EXACTLY** once, and every edge, twice, in a forward direction as it is visiting neighbouring nodes, and once backwards, when it is backtracking after it's done searching through all the neighbouring nodes.

A queue system, which utilises the **First-Come First-Serve** implementation, is used to store the neighbouring vertices which will be visited after the current vertex.

Every instance of visiting a vertex, it will be marked as "visited", such that it will not be put in the queue and be visited again.

BFS will be done till all vertices in the graph has been visited, that is, when the queue becomes empty.

The pseudo code for BFS that was implemented can be seen below:

```
Queue <Vertex> q = new LinkedList<Vertex>();
Map<Vertex,Integer> tree = new HashMap<Vertex,Integer>();
tree.put(startCity, value);
q.add(startCity);
startCity.mark();

//BFS
outerloop:
while (!q.isEmpty()) {
    Vertex v = q.remove();
    value = tree.get(v) + 1;
    LinkedList<Vertex> neighbours = Graph.getNeighbours(v);
    for (Vertex n : neighbours) {

        if (!n.getMarkedStatus()) {
            n.mark();
            q.add(n);
            tree.put(n , value);
            if (n.getCity().equals(endCity.getCity()))
                break outerloop;
        }
    }
}
```

Figure 1. BFS Code

```
//find shortest path
route.add(endCity);
Vertex start = endCity;

outerloop1:
while (!start.getCity().equals(startCity.getCity())) {
    for (Vertex v : tree.keySet()) {
        start = route.get(route.size()-1);
        if (start.getCity().equals(startCity.getCity()))
                break outerloop1;
        if (g.isNeighbour(v, start) && (tree.get(v).intValue() ==  (tree.get(start).intValue() - 1))) {
            route.add(v);
        }

    }
}
```

Figure 1. Code for finding the shortest path

Generating a Graph for BFS to use

In our implementation, we will be using an adjacency list (LinkedHashMap) to populate the graph of all the vertices and its edges after reading through a file that states all the links from each city to another.

```
Vertex [] vertices;
private static LinkedHashMap<Vertex, LinkedList<Vertex>> neighbours;
private static LinkedHashMap<String, Vertex> referenceToVertex;

public Graph(String [] cities) {
    neighbours = new LinkedHashMap<Vertex, LinkedList<Vertex>>();
    referenceToVertex = new LinkedHashMap<String, Vertex>();

    for (int i = 0; i < cities.length; i++) {
        Vertex vertex = new Vertex(cities[i]);
        neighbours.put(vertex, new LinkedList<Vertex>());
        referenceToVertex.put(cities[i],vertex);
    }
}
```

Figure 3. Method to populate the adjacency list

```
public static void addEdge (String srcCity, String destCity) {

    if (referenceToVertex.containsKey(srcCity) && referenceToVertex.containsKey(destCity)) {
        Vertex startVertex = referenceToVertex.get(srcCity);
        Vertex endVertex = referenceToVertex.get(destCity);

        //since it's a undirected graph, each of the vertex is added as the neighbour of the other
        neighbours.get(startVertex).add(endVertex);
        neighbours.get(endVertex).add(startVertex);
    }
}
```

Figure 4. Method to add edges of each vertex into the graph

```
public static void generateGraph(String [] edges) {
    String [] edge;
    for (int i = 0; i < edges.length; i ++) {

        edge = edges[i].split(",");
        addEdge(edge[0], edge[1]);
    }
}
```

Figure 5. Method to generate the graph

Graphs

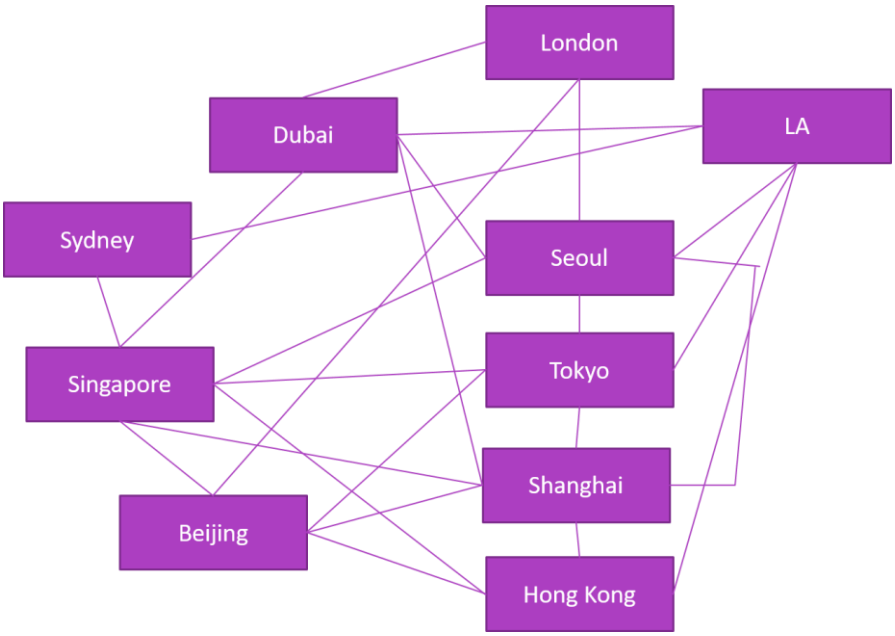

Figure 6. Graph for 10 cities

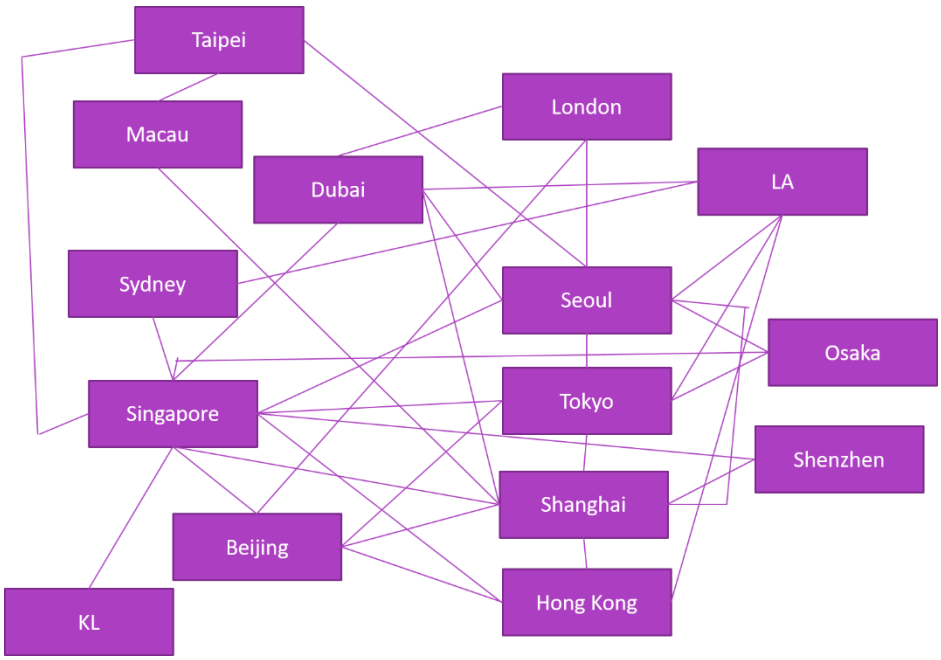

Figure 7. Graph for 15 cities

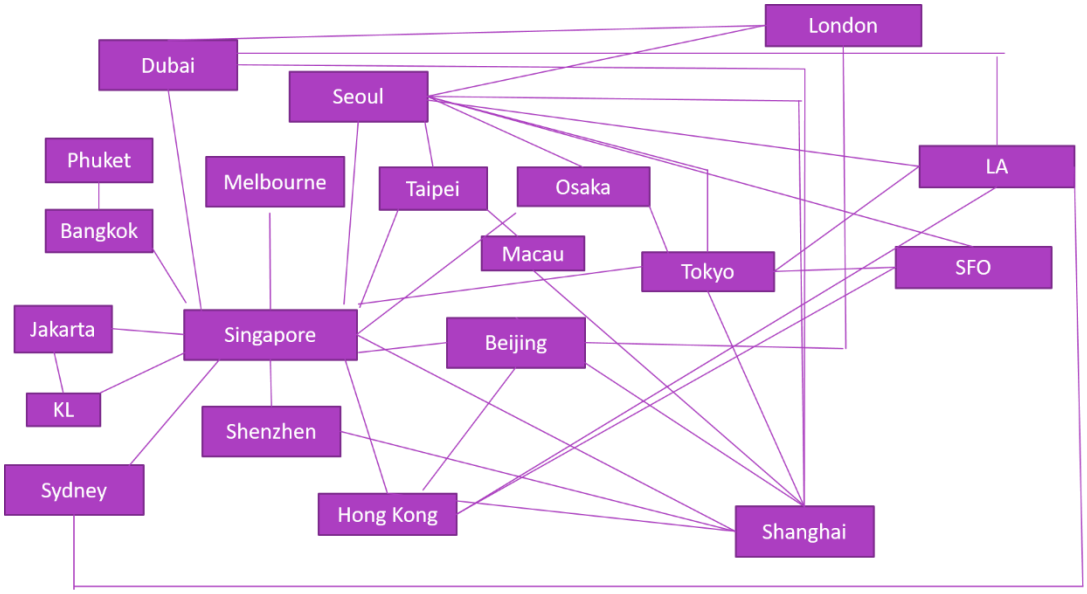

Figure 8. Graph for 20 cities

CPU Time based on different amount of edges and vertices

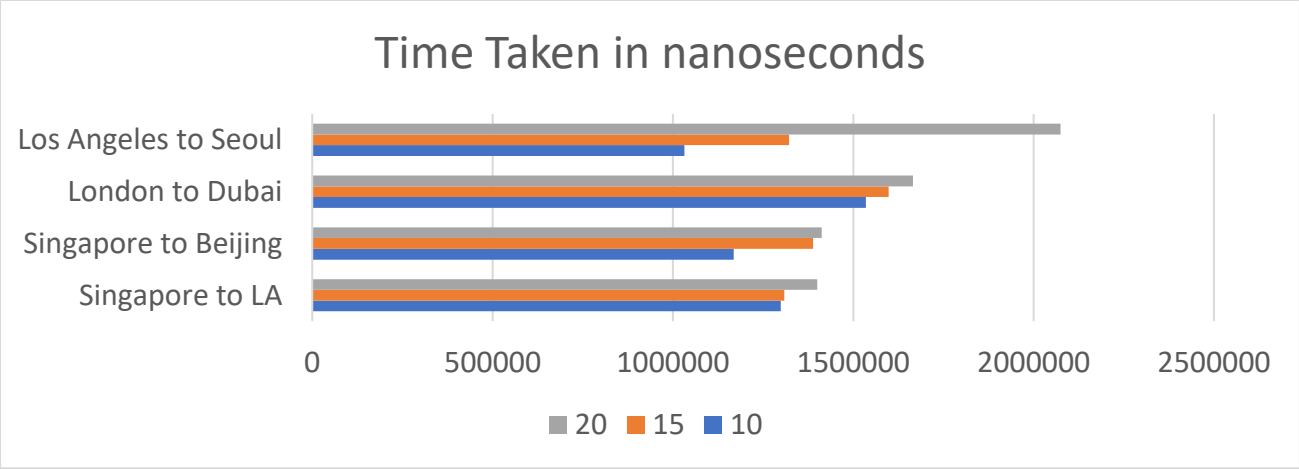

Figure 9. CPU Time of BFS

As shown in the graph above, the running time of BFS increases as the number of vertices increases (cities in this scenario). With the amount of non-stop flights increasing together with the number of cities, the running time also increases if the algorithm cannot find a direct flight between the cities that the user has input into the program.

Can Depth-First Search (DFS) algorithm be used in place of BFS?

No, DFS cannot be used in place of BFS. DFS does not guarantee that it returns a shortest path. For DFS, we go deeper first. After visiting the first neighbour, we will visit the neighbour of the first neighbour, instead of the second neighbour of the source vertex. The difference of the 2 algorithms are that BFS and DFS give priorities to different dimensions, breadth (which is horizontal) versus depth (which is vertical).

In our scenario, take for example that a user wishes to search for a flight from London to Phuket. When we are using BFS, the shortest path that is calculated would be London → Beijing → Singapore → Bangkok → Phuket. However, if we are to use DFS (assuming that we transverse in alphabetical order), the path that would be calculated will be London → Beijing → Hong Kong → Los Angeles → Dubai → Seoul → Osaka → Singapore → Bangkok → Phuket which does not satisfy the shortest path condition.