Design of Algorithms Assignment 1 Written Report
Liam Aharon
Student #835596

## Part 6: Graph Representation

In the event that the graph in the assignment was stored using an *adjacency matrix* rather than an *adjacency list*, some changes would have to be made to the way depth-first traversal and breadth-first traversal is performed in my implementation.

**When acquiring a vertex's next edge we need to traverse an adjacency matrix, rather than a linked list.** To find the next edge we need to traverse the row in the adjacency matrix belonging to the vertex as shown in following pseudocode:

```
vertex_edge_row = adjacency_matrix[vertexid]
for i=0, i<vertex_edge_row.length, i++
      if vertex_edge_row[i] == 1 //vertex with id i is adjacent to vertex with id vertexid
            //do something with vertex
```

**The id of the destination of an edge will also need to be obtained differently.** It is easily obtained as the index that the edge is found in vertex_edge_row seen above.

## Part 7: Design of Algorithms

**For part 3,** my algorithm performs depth first search (DFS). It starts by pushing the origin vertex to the top of a stack (with distance 0), looks at the edges of the vertex and pushes the first unvisited vertex it finds and the distance it takes to get it to the top of the stack. The process then repeats with the new vertex on the top of the stack. If at any point the algorithm reaches a vertex which doesn't contain an edge to any unvisited vertices, it pops the dead end vertex and distance off the stack and tries again with the other edges. If at any point the destination is found, the stack is reversed and every vertex in it is popped. As vertices are popped, their label is printed with a cumulative distance alongside.

**For part 4,** my algorithm performs DFS recursively to find all paths between source and destination. Starting with the source vertex, it adds the vertex to an array and marks the vertex as visited in another array. It then recurses on every unvisited destination accessible by the vertex's edges **a separate copy of the vertex array and visited information is passed along to every recursion, so that a recursive instance will never change the path and vertex information of an instance lower down in the execution stack**. If at any point the destination is found, recursion will not be performed on the destination vertex and instead the current path for that particular iteration will be printed.

**For part 5,** my algorithm is very similar to part 4, except there is an extra array and data created when the function is first called which is used to store data regarding the shortest known path at the time. Also, cumulative distance is updated at the start of each recursive instance so the length of  the current path is known at all times. When a path from origin to destination is found, instead of printing it, the current path distance is checked. If the distance is shorter than the current short path distance then short path is replaced with current path. Once every path has been traversed (last vertex popped off the stack) the short path is printed along with its distance.

## Complexity analysis

**Part 3**
The time complexity of this algorithm is O(n + e) where n is the number of vertices and e is the number of edges. This is because in the worst case every single vertex and edge must be traversed.

**Part 4**
The time complexity of this algorithm is O(n!), where n in the amount of vertices in the graph, for in every case the algorithm will traverse every possible path in the graph (without visiting a vertex more than once) in order to make sure for certain that every single path has been found.

**Part 5**
The time complexity of this algorithm is O(n!), where n in the amount of vertices in the graph, the reason being in every case the algorithm will traverse every possible path in the graph (without visiting a vertex more than once) in order to make sure for certain that every single path has been found.