Siyi Guo, 737008

Part 6 and bonus (Remove Recursion):

Suppose the adjacency matrix is imported in the form of a two dimension int array **Adjt_Matrix[Number of Rows][Number of Column]:** we need to change how we go through all nodes that connected to current node.

Originally, we were using **while(edge != NULL){store node index in queue(bfs) or stack(dfs); edge = edge->next;}** to access the neighbours of current node. This is was used because the neighbour information were stored in a linked list.

Now since every node's neighout information is stored in **Row_Array** accessed by **Adjt_Matrix[node_index].** We can replace our while loop by a for loop: **for(int i = 0; i < number of node; i++){if Row_array[i] > 0{store i in queue(bfs) or stack(dfs);}}**

This replacement go through every column of current node's row, and if there are any column entry in this row that is greater than 0, that means the node represented by this column is a neighbor, we therefore should put it into stack(for dfs search) or queue(for bfs search).

Just replacing the way we access every neighbor of current node, the program will adapted to the adjacent matrix.

Part 7, Design of Algorithm:

Part 4: In finding all path from Source to Destination, we define function **dfs**, which takes 6 argument: **graph, source_id, destination_id, path array, explored array and path length integer**. We recursively used dfs as the problem from A (Source) to An(Destination) can be view as many sub problems that from Ai to An, where 0<i<n.

Every time we recursively call **dfs**, we do 3 things:

1, Record current node into the **path_array** at position **path_length**.

2 increase **path_length** by 1 as the length has increased.

2, mark this node as visited in **explored_array** to avoid path going backward.

Then we check whether we have reach our destination, if so, we print out the information in our **path_array** from first element to **path_length** th element. And if we have not reach, we apply **dfs** function to this node's neighbour, except those node that already in current path, the use neighbor node as **source_id** argument for **dfs.** At the end of **dfs** we mark current node as unvisited, because this node might become a part of path for other paths. And we marked at first simply to avoid backward edge.

Part3 and Part 5: both of this part using Dijkstra Algorithm as a shorted path from source to destination is also a detailed path. The algorithm's aim is to find a minimum spanning tree. Algorithm is as followed:

1 define a data structure for each node: an int for distance from source, a bool for permanent label or not and an int for parent id of this node in the spanning tree. A node with permanent label has the distance from source to this node, and temporary label means this node's distance from source is infinity

2 creating an array with size = graph->manx of the defined data structure, initialize all nodes distance to infinity, label to temporary and adjacent node to nothing.

3 set source node's **u0** distance as 0, permanently label it. This is the roost for our spanning tree.

4 while **i** < number of vertex of this graph, do:

(a) Choose a vertex with temporary label that has minimum distance. Set this node as pivot point **ui** and scan its neighbor

(b) For all edge **ui v** with vertex v that has temporary label:

a) If distance of **v** > distance of **ui** + weight of edge **ui v,** then:

i. Set the distance of v to distance of **ui** + weight of edge **ui v,** set v's partent id to **ui**

b) Change **ui's** label to perment; **i** = **i**+1;

Note: neighbor can simply accessy by using array[neighbor_id]

5 go to array[destination_id] and traced back every node's parent node, put node into stack, then we have a path from source_id to destination_id by pop up stack.

Time Complexity analyse

For part 4:

n is the number of vertices in graph

Function Dfs: T(n)

if source_id == destination_id:

prin_out_path (O(n))

else:

Go through every neighbor of current node: Loop m times, m= number of neighbor

Call DFS T(n-1)

Therefore, we have equation T(n) = m*(T(n-1) + O(n), by expanding the recursion, we eventually have equation for complexity of out algorithm:

$$\mathrm{T(n)} = \sum_{i=0}^{n} (\prod_{j}^{i} m_j) * O(n-i)$$

Therefore, in the worst case, algorithm will have complexity T(n) = O(n!) if the graph input is a complete graph.

And in the base case, algorithm will have complexity T(n) = O(n) it the input graph is a trial.

There is no average case as the neighbor of every nodes in graph can varied from 1 to n-1.

For part 5 and part 3:

Since both parts using Dijkstra algorithm, the time complexity of them are analyzed as follows:

Initialize the record array. O(n)

Find the smallest label in an array of at most n nodes each time,

This operation will run n times. O(n^2)

Each edge is examined exactly once. O(m), m is number of edges in G

Since the number of edges in a complete graph is n*(n-1)/2, we know m <= n*(n-1)/2 < n^2

Therefore, this algorithm's complexity is O(n^2)