Design Manual for Assignment 4

The purpose of this manual is to provide a descriptive description of the structure of the problem. This manual will comprises of 5 sections each section corresponding to the four major functions coded and the graph representation

Section 0 : Representation of the graph and other structures

0.1. The graphs are represented using adjacency List -> adj_List

I utilized link-list for this purpose, the list adj_list is a pointer array which would store the adjacent vertex from any Vertex V in Graph G.

In order to get the transpose of the graph i.e. reversing the edges, we have stored those values in List -> rev_List.

Function that is responsible for construction :: adj_construction.
Time complexity :: O(V+E)

0.2 For storing the bus stops we are using an approach similar to hashing, where each bus stop has an index value and corresponding to it we will store its name.

We are using an array of structure :: stopList which stores this.

Function that performs this task is :: stop_names
Time complexity :: O(N) where N are names of the stops.

0.3 We use a Link-list to store the edges and their weight from reading the bus routes file.

Here again we use a pointer array where each pointer points to a edge to the destination and its corresponding weight.

Function :: reading_bus_routes.
Time complexity :: O(N)

Section 1 : StronglyConnectivity

In this function we will check if the given graphs is strongly connected or not.
First we will create the graph and the given stop names using the functions.

1. ***Stop_names ::*** for reading the stop names file.
2. ***Reading_bus_routes ::*** for getting the information of the weight.
3. ***Adj_construction ::*** for constructing the adjacency list of the matrix.
    a) This function would return the number of the vertex for the graph.

4. ***StronglyConnected ::*** Responsible for performing the check for connectivity.

    a) This function uses the Kosaraju's algorithm for checking the connectivity of the graph.
    b) First performs a DFS on the graph.
    c) Then it checks if all the vertex were reached from this DFS from the visited array.
    d) If all the vertex were visited, then we will reverse the edges thus, we took transpose of the graph, and have stored reverse_list.
    e) Perform second DFS.
    f) If one can reach all the vertex then this is a connected graph.

Reasons for using the above algorithm.

1. Connectivity is a two way street where we must ensure that taking any of the edges one can reach all other Vertex of the graph.
2. The DFS by default traverses for the longest path before backtracking. Thus taking into account all the vertexes that can be reached.
3. The reversal of the edges makes sure to check whether we can reach back to this vertex from any or all of the vertexes present in the graph.

4. ***Time complexity of this function is ::***
   i. We used adjacency list for graph representation, thus :: O(V+E)

Section 2 : MaximalStronglyComponents

MaxStrongly for finding the graph's strongly connected components.

This function is another implementation of Kosaraju's algorithm.
We have similar visited vertex array which is created dynamically.

This function will first fill the visited vertex using DFS.

This would create a Stack(we implemented one using arrays) and keeping count of the l length of the last element inserted.

Then we perform DFS on the stack created by the fill_vertex function.
This would ensure to provide the connected components of the graph G.

The Time complexity of the function is :: O(V+E)

Section 3 : ReachableStops

This function would find the reachable vertex from the graph G.
The helper function for this is reachable.

We created a Structure Queue which being used to implement ADT queue.

Again a visited array is created for the vertex visited in the graph.
We first find the stop code using the hash created for the stop names.

We use BFS for this function using the ADT queue created earlier.
This is an iterative implementation of BFS and checking if we can reach other vertex from this function.

Note ::

1. We utilize both the adjacency list and the stop names created in the Section 1. Refer Section 1 and Section 0 for details of the same.
2. The architect of the program kept the name of the BFS as dfs_max, it should not be confused with DFS.

a) The articulation of the function name was in effect with the flow where this function performs the operation of reaching from one Vertex to Another.

Section 4 :: TravelRoute

Here we have used Priority Queue using Binomial heaps, along with Dijkstra's Algorithm to Find the minimum cost/distance from source to destination.

This function first creates a edge path and assign the value of Inf to the vertex.
The we run the function min_distance (which runs Dijkstra's algorithm)

Proof Of correctness for the Dijkstra's is by induction.

Namely its stated that the min cost if exits for K (Vertex ie. From X->K)
Then if there exists a path from K to K', then finding the least cost path from K to K' and then adding it to previous path for K would enable to find the least cost path.

Why we used Priority Queue and BH ?

We have to get the least paths I.e. the cost reaching to one vertex. The PQ provides the most optimum approach where we would base the key of the Queue as weight of the Vertex.

Thus, easily providing the implementation and also keeping in check the complexity.

Note ::

For all of the functions the names of the file-names are given from the main function.
There is no input taken from the console.

If the parameters are changed then change them from the main function.

The default values for Section 3 is :: Bridge Street for Source

The default values for Secton 4 is :: Bridge Street for Source and Bathurst Street for destination

The values will have to be changed in the main function for testing.