NAMAN ARORA
19323369
Computer Engineering

REPORT FILE
ASSIGNMENT 4 : SEARCHING WITH GRAPHS

TASK 1: BFS AND DFS

- Graph Representation I used is **Adjacency List.**
- An array of lists is used. Size of the array is equal to the number of vertices. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs.
- Not used Adjacency Matrix because : Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.
- Adjacency List saves space O(|V|+|E|) . In the worst case, there can be C(V, 2) number of edges in a graph thus consuming O(V^2) space. Adding a vertex is easier.

BFS ::
BFS for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

DFS ::
DFS for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more

than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

OUTPUT :-

```
Adjacency list of vertex A
 head -> A-> B-> D-> E

 Adjacency list of vertex B
 head -> B-> C-> D

 Adjacency list of vertex C
 head -> C-> B-> D

 Adjacency list of vertex D
 head -> D-> F

 Adjacency list of vertex E
 head -> E-> D

 Adjacency list of vertex F
 head -> F-> C

BFS traversal :: A    B    D    E    C    F

DFS traversal :: A    B    C    D    F    E
```

TASK 2: DIJKSTRA

In Dijkstra's algorithm, two sets are maintained, one set contains list of vertices already included in SPT (Shortest Path Tree), other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in O(V+E) time using BFS. The idea is to traverse all vertices of graph using BFS and use a Min Heap to store the vertices not yet included in SPT (or the vertices for which shortest distance is not finalized yet). Min Heap is used as a priority queue to get the

minimum distance vertex from set of not yet included vertices. Time complexity of operations like extract-min and decrease-key value is O(LogV) for Min Heap.

Following are the detailed steps.

1) Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains vertex number and distance value of the vertex.

2) Initialize Min Heap with source vertex as root (the distance value assigned to source vertex is 0). The distance value assigned to all other vertices is INF (infinite).

3) While Min Heap is not empty, do following

…..**a)** Extract the vertex with minimum distance value node from Min Heap. Let the extracted vertex be u.

…..**b)** For every adjacent vertex v of u, check if v is in Min Heap. If v is in Min Heap and distance value is more than weight of u-v plus distance value of u, then update the distance value of v.

- Adjacency List is used to represent the graph
- For this I have created 3 structures :
    - 1st is the Node
    - 2nd is a pointer to the node the node to re3rd present the head.
    - 3rd is the graph structure which contains
        - No. of vertices V
        - An array of 2nd type of structure

- There is an addEdge Function to add the destination node to the list of the src node.

- **For keeping track of the previous node from which the shortest distance to the node was updated, i have taken an array prev[], and update it whenever I update a distance to a node.**

OUTPUT:-

Permanent path :: A, B, C, G, E, D, F,

Shortest Distance from Source A ::

B	1
C	2
D	7
E	5
F	7
G	3

Shortest Path for B :: B<-A
Shortest Path for C :: C<-B<-A
Shortest Path for D :: D<-E<-C<-B<-A
Shortest Path for E :: E<-C<-B<-A
Shortest Path for F :: F<-E<-C<-B<-A
Shortest Path for G :: G<-B<-A

TASK 3: ON THE BUSES

- This program is similar to the previous one.
- But for storing the vertices and edges from the CSV files, I have created two new structures verticeNode and edgeNode.
- For keeping track of the path or previous nodes from where I have updated the shortest distance to some node I have created another structure pathlist.
- So pathlist helps in binding the graph nodes with the vertice and edge nodes.
- Pathlist is first initiazed first with initPath
- It is updated whenever any weight/distance of a node gets updates.

- Other than that it functions similar to the 2nd question in finding the minimum node at every step.

OUTPUT:-

Loaded 4807 vertices
Loaded 6180 edges

start stop :: 10

end stop :: 78

Path

10	Parnell Square	53.353387	-6.265384
12	Dorset St	53.356789	-6.264623
14	Dorset St	53.358537	-6.262724
15	Dorset St	53.360251	-6.260973
1175	Gardiner St	53.358589	-6.261158
1174	Mountjoy Square	53.356294	-6.258693
51	Dorset St	53.357816	-6.263266
84	North Circular Road	53.360155	-6.263697
83	North Circular Road	53.360274	-6.265540
82	North Circular Road	53.360787	-6.271673
80	Cabra Road	53.360854	-6.276050
79	Cabra Road	53.361014	-6.280786
78	Cabra Road	53.361131	-6.283972