**Airline Reservation System**

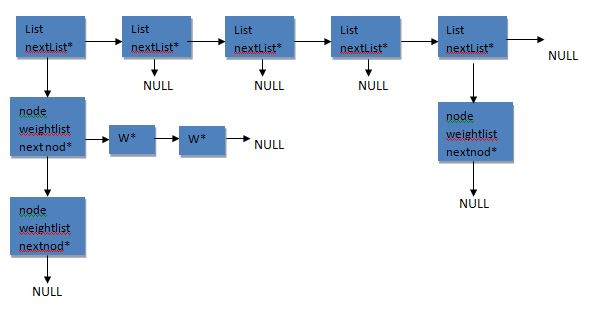
**Basic Structure of Graph:**

**Adjacency List:**

**We implemented our graph using an Adjacency List**. Our adjacency list was maintained by using a **linked list with a pointer pointing towards another linked list**. Here every linked list pointing towards another linked list represents a Vertex and its neighbors. The first element of the linked list will always be the vertex and all the other elements ahead of it will be its neighbors. **This type of structure optimizes data storage as there is no data memory being wasted. Every data member is being utilized.**

Of course the Weight of Vertex to itself is uninitialized but that is important as it represents a self edge, and since there is no Vertex with an edge to itself, the weight of vertex with itself must be zero.

**Graph Structure Diagram:**

****

**Node:**

We modified our Node class for it to have two main data members, one for obviously storing the key, and the other for its weight. Node class has a key and a list of weights**. The list of weights is important as there are multiple weights for even a direct edge.** The weights can later on be sorted with respect to cost, time of flight or earliest date to book a flight with required conditions.

**Constructor:**

There are two constructors of the graph. Default constructor creates an empty graph with zero vertices and edges. We can later on insert vertices and edges onto it.

The parameterized constructor takes two strings as a parameter, a filename with flights information and a filename with city’s hotel charges. This constructor creates a graph with each vertex representing a city and its adjacent list representing its direct edges. If a direct edge has more than one weight, then it is added into its specific weight list (i.e. a new node is not created for every edge).

Algorithm:

* Reads the file line by line.
* Performs operation on the line read.
* Since the information in that line is separated by spaces, It stores the different info in designated variables.
* After the entire line is read, it inserts already read vertices, (if vertices exist, new vertices are not created).
* Inserts edges b/w the two vertices, if edge already exists, it adds the edge onto the weight’s list.
* After the graph is created, it reads the hotel charges file.
* Stores the transit cost inside AdjList of specific vertices.

**Destructor:**

Our destructor de-allocates the memory of adjacency list one by one starting from first vertex. The de-allocation calls the destructor of List and it de-allocates contents of its list, i.e. its neighbors. These de-allocations calls the destructor of list of weights and that memory is de-allocated as well.

**Display Functions:**

* Display(): displays the Adjacency list
* DisplayWeighted(): displays the Adjacency list with weights
* BFS(): displays graph vertices with breadth first traversal
* DFS(): displays graph vertices with depth first traversal
* DisplayVertices(): Displays all the vertices.
* ShowPath(T place1,T place2, int mode): shows direct path b/w path1 and path 2 for different modes
* Mode 0: show path w.r.t. minimum cost.
* Mode 1: show path w.r.t. earliest date.
* Mode 2:show path w.r.t. shortest flight time.

**Insertion Functions:**

* insertVertex(T ver): Inserts vertex into graph
* insertEdge(T src, T dest, U weight): Inserts a weighted edge between src and dest.
* InsertEdgeUndirectional(T src, T dest, U weight): insertes an inderectional weighted edge between src and dest.

**Getters:**

* getListAt(int index): returns the list (Vertex and its neighbors) at specified index.

Algorithm: Traverses the Adjacency list till the desired index and returns the adjList.

* getList(T place): returns the list (Vertex and its neighbors) of specified place.

Algorithm: Traverses the Adjacency list till the desired data is found and returns the adjList.

* getIndex(T place): returns index of where the Vertex of place is present.

Algorithm: Returns the index of where the Vertex (place) is present.

**Traversals:**

The time complexity of traversals in graph will be |V|\*|E| i.e. Num of vertices \* Num of edges. If we take into consideration, the traversals of weight’s list aswell, then the traversal will be |V|\*|E|\*|W|, i.e. the previous product \* Num of Weights. Normally the weights will be sorted, hence the complexity will remain |V|\*|E|.

**FLOYD-WARSHALL’s ALGORITHM:**

In this Project, we have opted Floyd-Warshall’s algorithm to find the shortest path (in terms of cost and time) between two cities. We handled all scenarios related to minimum cost and minimum time,individually. For this purpose two separate functions have been used

voidfloydWarshall(strings, stringd);// gives the all-pairs shortest path between any

two vertices in terms of cost

void floydWarshall2(strings, stringd);// gives the all-pairs shortest path between any

two vertices in terms of time

where ‘s’ and ‘d’ are source and destination of type string.

Each of these functions first maps the entire graph onto a size x size(size is the number of vertices) 2D matrix of Edges of type Node<T,U>. Then we sort the weights of each Node w.r.t. cost or time difference between the arrival and departure time such that the weight on zeroth Node of Weights list is always providing minimum travel cost or minimum time difference between arrival time and departure time.

After this Floyd Algorithm is applied on the matrix of Edges such that the matrix we obtain on the final iteration contains only the minimum possible costs or time differences bwteen any two cities.

A parent matrix is also maintained to keep the track of path in case of connecting Flights.

Floyd’s Algorithm works as follows:

**STEPS:**

For a graph with N vertices:

* Initialize the shortest paths between any 2 vertices with Infinity.
* Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1 intermediate vertex and so on..until using all N vertices as intermediate nodes.
* Minimize the shortest paths between any 2 pairs in the previous operation.
* For any 2 vertices (i,j) , one should actually minimize the distances between this pair using the first K nodes, so the shortest path will be: min(dist[i][k]+dist[k][j],dist[i][j]).

dist[i][k] represents the shortest path that only uses the first K vertices, dist[k][j] represents the shortest path between the pair k,j. As the shortest path will be a concatenation of the shortest path from i to k, then from k to j.

**COMPLEXITY**

“O(|V^3|)”,where V is the number of vertices

**WHY IS IT BETTER THAT DIJKSTRA AND BELL-FORD’s ALGORITHM?**

If we have to find the shortest path between all pairs of vertices, Dijkstra and Bell Ford’s Algorithms would be expensive in terms of time. Dijkstra’s algorithm finds the shortest path from one vertex to all others with a time complexity O(V^2).**If applied on all vertices it’s complexity will rise to O(V^3).** Whereas, the complexity of BellFord’s algorithm O(V.E) if E=V,O(V^3), where V is the number of vertices and E is number of edges.

But both these algorithms do not provide all pairs shortest paths unless applied on each and every vertex. **Whereas, Floyd’s algorithm helps to find the shortest path in a weighted graph with positive or negative edge weights. A single execution of the algorithm is sufficient to find the lengths of the shortest paths between all pairs of vertices. It is easy to modify the algorithm and use it to reconstruct the paths**

**Non-member Functions:**

* getSubGraph(string airline, string flights, string hotelCharges): Returns a subgraph of main graph, with only the desired airlines.

Algorithm: Reads the file again and creates a sub-graph of specific airline.

* Sort\_wrt\_cost(List weight): Sorts the weight list w.r.t minimum cost at beginning.
* Sort\_wrt\_time(List weight): Sorts the weight list w.r.t minimum travel time at beginning.
* Sort\_wrt\_days(List weight): Sorts the weight list w.r.t earliest day at beginning.

**Scenarios Covered:**

* Minimum Cost with direct flights.
* Minimum travel time with direct flights.
* Earliest flight with direct flights.
* Minimum Cost with connected flights.
* Minimum travel time with connected flights
* Shows the whole path followed. Be it direct or connected flight.
* Will stop by at transit location if specified.

**User Interface:**

The user interface function requires two parameters, Flights file and hotel charges file. It creates the required graph and a menu for the user.

* The user can view their flights and related information.
* The user can choose any special airline they want to travel with, if they want to.
* The user can book their flights.
* Flights will be booked according to user’s preferred selected requirements.
* After flight, user will land onto the desired destination, after which user can keep on travelling.

**Bonus:**

Animation of the airplane leaving its destination and flying into the sky has been added.

Resources: (For Justifying Floyd’s ALgorithm)

<https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/>

<https://brainly.in/question/14003026>