1. PROBLEM

The Shortest Path Finder

The shortest path problem is to find a path between two vertices/nodes in a digraph with non-negative edge weights G= (V, E) and a distinguished source vertex s belongs to V such that the sum of the weights of its constituent edges is minimized in a graph.

The problem of finding the shortest path between two intersections on a road map may be modelled as a special case of the shortest path problems in graphs, where the vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of the segment.

2. ANALYSIS

Dijksa's algorithm is an efficient single-source shortest path algorithm. It helps us finding the shortest distance from the source vertex to all other vertices in the graph. As opposed to breadth-first search, it efficiently solves the single-source shortest path problem for weighted graphs (graph with weighted edges).

This algorithm will run until all vertices in the graph have been visited. This means that the shortest path between any two nodes can be saved and looked up later.

Rules for running Dijkstra's algorithm:

1. From the starting node, visit the vertex with the smallest known distance /cost.
2. Once we've moved to the smallest-cost vertex, check each of its neighbouring nodes.
3. Calculate the distance/cost for the neighbouring nodes by summing the cost of the edges leading from the start vertex.
4. If the distance /cost to a vertex we are checking is less than a known distance, update the shortest distance for the vertex.

3. DATA REQUIREMENTS

Problem Constant

Nil.

Problem Input

n : Total no. of cities

a[ ][ ] : The adjacency Matrix to store which cities are adjacent with what distance value.

src : The Source City from where we want to travel.

dest : The Destination City to where we want to travel.

Problem Outputs

dist[dest] : The Shortest possible DISTANCE between the given points.

parent[] : The Shortest possible PATH between the given points.

Relevant Formulas

If Vertex is not visited and dist(u, v) < the given distance(or Zero ‘0’) then new distance=dist(u, v)

4. DESIGN

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph.

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

a. INITIAL ALGORITHM

1. Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

3. While sptSet doesn't include all vertices

    a. Pick a vertex u which is not there in sptSet and has minimum distance value.

    b. Include u to sptSet.

    c. Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

b. ALGORITHM REFINEMENTS

```
// Sending original graph G along with the distances between each points
// and also sending the start location

Dijkstra_SP(G, w, s)
{
        //creating a dist[] array to find distance between two points.
        //Also creating a pred[] array to store the Shortest PATH

        for each vertex i in Vertex set V[G]
        {
                dist[i]= infinite;
                pred[i]= NULL;
        }

        dist[s] = 0;
        pred[s] = -1;
        //initialing distance from start point as 0(zero)
        //and predecessor value of start point as -1

        Q = V[G];
        //Creating a Priority Queue to store all Points in the graph
        //according to their distances from start point

        while(Q is not empty)
        {
                u= EXTRACT_MIN(Q);
                for each v in adj(u)
                {
                        //adj(u) means set of points adjacent to u
                        if ((v is in Q) AND (dist[v]>w[u,v]+dist[u]))
                        {
                                //updating distance array continuously
                                //if a shorter path from start to the vertex 'v' is found
                                dist[v] = w[u,v]+dist[u];
                                pred[v] = u;
                        }
                }
        }
}
```

5. IMPLEMENTATION

Following is the JAVA code to find out shortest path between any two cities amongst the given cities:

```
//A Java program for Dijkstra's single source
//shortest path algorithm.

//This program is based on adjacency matrix
// representation of the graph

import java.util.*;
class ShortestPath
{
static int vtx ; // for total vertices
public ShortestPath(int v){
        vtx=v;
}

//A func to find the vertex with min distance from the set
//of vertices not yet included in shortest path tree
int minDistance(int dist[], Boolean sptSet[]) {
        // Initialize min value with Largest Possible
        int min = Integer.MAX_VALUE, min_index = -1;
        for (int v = 0; v < vtx; v++)
                if (sptSet[v] == false && dist[v] <= min) {
                        min = dist[v];
                        min_index = v;
                }
        return min_index;
}

// A utility function to print the constructed distance array
void printSolution(int dist[],int src,int dest,int parent[])
{
        System.out.println("The minimum distance between source: "+src+
                " and destination: "+dest+" is: "+dist[dest]+" units.");
        System.out.println("The shortest path between the two cities is: ");
```

```
        int i=dest;
        System.out.print("City "+ i +" ");
        while(parent[i]!=-1)
        {
                i=parent[i];
                System.out.print("city"+i+" ");
        }
}
/* Function that implements shortest path between a source and a sink
vertex for a graph represented using adjacency matrix representation */
void dijkstra(int graph[][], int src, int dest)
{
        //dist[] holds the shortest distance from source to every other vertex
        int dist[] = new int[vtx];
        //Array to store the cities included in shorted path
        int parent[] = new int[vtx];
        //sptSet[i] will true if vertex i is included in shortest path tree or
        //shortest distance from src to dests is finalized
        Boolean sptSet[] = new Boolean[vtx];
        // Initializing all distances as INFINITE and
        //stpSet[] as false -> no vertexis included in shortest path
        for (int i = 0; i < vtx; i++)
        {
                dist[i] = Integer.MAX_VALUE;
                sptSet[i] = false;
        }
        dist[src] = 0;// Distance of source vertex from itself is always 0
        parent[src]=-1;// there is no parent for source vertex

        // Find shortest path for all vertices
        for (int count = 0; count < vtx-1; count++)
        {
                // Pick the minimum distance vertex from the set of vertices
                // not yet processed. u is always equal to src in first
                // iteration.
                int u = minDistance(dist, sptSet);
                // Mark the picked vertex as processed
                sptSet[u] = true;
                // Update dist value of the adjacent vertices of the
                // picked vertex.
```

```
        for (int v = 0; v < vtx; v++)
                        // Update dist[v] only if is not in sptSet, there is an
                        // edge from u to v, and total weight of path from src to
                        // v through u is smaller than current value of dist[v]
                        if (!sptSet[v] && graph[u][v] != 0
                                && dist[u] != Integer.MAX_VALUE
                                && dist[u] + graph[u][v] < dist[v])
                        {
                                parent[v]=u;
                                dist[v] = dist[u] + graph[u][v];
                        }
        }
        // print the constructed distance array
        printSolution(dist,src,dest,parent);
}



// Driver method
public static void main(String[] args) {
        Scanner sc= new Scanner(System.in);
        System.out.println("Enter the total number of cities: ");
        int n= sc.nextInt();
        int a[][]=new int[n][n];
        System.out.println("Enter the direct distance from city to city");
        for(int i=0;i<n;i++)
                for(int j=0;j<n;j++)
                {
                System.out.println("enter the distance of city "+i+" from city "+j+" : ");
                        a[i][j]=sc.nextInt();
                }
        System.out.println("Distances from each city to every other city");
        System.out.println("Adjacency matrix representation of the graph
                        with all the cities as nodes and distance between
                        every city as weighted, undirected edges");
        for(int i=0;i<n;i++)
        {
                for(int j=0;j<n;j++)
                        System.out.print(a[i][j]+" ");
                System.out.println();
```

```
        }
        int src=-1,dest=-1;
        while(!(src>=0 && src<n))
        {
                System.out.println("Enter the source city between 0 to "+(n-1)+" : ");
                src= sc.nextInt();
        }
        while(!(dest>=0 && dest<n))
        {
                System.out.print("Enter the destination city between 0 to "+(n-1)+" : ");
                dest= sc.nextInt();
        }
        // user input finishes...
        ShortestPath t = new ShortestPath(n);
        t.dijkstra(a, dest,src);
}
}
```

---

6. Testing

This is a sample test case:

Enter the total number of cities:
9

Enter the distances from each city to every other city:

00	04	00	00	00	00	00	08	00
04	00	08	00	00	00	00	11	00
00	08	00	07	00	04	00	00	02
00	00	07	00	09	14	00	00	00
00	00	00	09	00	10	00	00	00
00	00	04	14	10	00	02	00	00
00	00	00	00	00	02	00	01	06
08	11	00	00	00	00	01	00	07
00	00	02	00	00	00	06	07	00

Enter the source city between 0 to 8: 5

Enter the destination city between 0 to 8: 7

The minimum distance between source 7 and destination 5 is: 3 units.

The shortest path between the two cities is:

City 5 City 6 City 7

The above sample test case and output provide a good test for the solution because it is relatively easy to visualize and understand by drawing the graph.