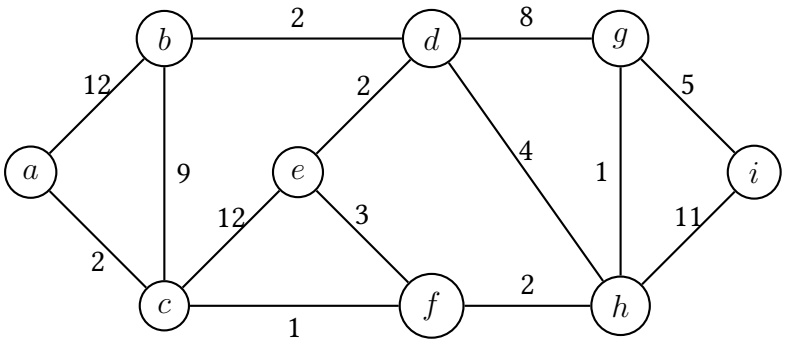
CS6033 Design and Analysis of Algorithms I
Homework Assignment 4
10.15.2022

Group Members:
rk4305 (Sai Rajeev Koppuravuri)
sg7372 (Sriharsha Gaddipati)
vt2184 (Veeravenkata Raghavendra Naveenkumar Tata)
vt2182 (Venu Vardhan Reddy Tekula)

---

1. (10 points) Run Dijkstra's algorithm on the following graph where the source vertex is a.



Ans:

Dijkstra's shortest path algorithm works as follows:

Visited = []
Unvisited = [a, b, c, d, e, f, g, h, i]

The start vertex be a
Distance from a to a is 0
Distances from a to all the other vertices is not known, let mark it ∞

Vertex	**Shortest Distance from a**	**Previous vertex**
a	0	
b	∞	None
c	∞	None
d	∞	None
e	∞	None
f	∞	None
g	∞	None
h	∞	None
i	∞	None

Visit the unvisited vertex with the smallest known distance from the start vertex
First time, this is the start vertex itself, a

For the current vertex, examine its unvisited neighbours
We are currently visiting a and its unvisited neighbours are b and d

For the current vertex, calculate the distance of each neighbour from the start vertex
From the graph, the distance from
- a to b is 0+12=12
- a to c is 0+2=2

If the calculated distance of a vertex is less than the known distance, update the shortest distance

Update the previous vertex for each of the updated distances
In this case we visited b and c via a

Visited = [a]
Unvisited = [b, c, d, e, f, g, h, i]

Vertex	**Shortest Distance from a**	**Previous vertex**
a	0	
b	12	a
c	2	a
d	∞	None
e	∞	None
f	∞	None
g	∞	None
h	∞	None
i	∞	None

Add the current vertex to the list of the visited vertices

The next step is to visit the unvisited vertex with the smallest known distance from the start vertex
This time, it is c

For the currrent vertex, examine its unvisited neighbours
We are visting c and its unvisited neighbours are b, e and f

For the current vertex, calculate the distance of each neighbour from the start vertex
From the graph, the distance from
- c to b is 2+9=11
- c to e is 2+12=14
- c to f is 2+1=3

If the calculated distance of a vertex is less than the known distance, update the shortest distance

The shortest distance from a to b is 11 via c, so update it

Update the previous vertex for each of the updated distances
In this case we visited e and f via c

Visited = [a, c]
Unvisited = [b, d, e, f, g, h, i]

Vertex	**Shortest Distance from a**	**Previous vertex**
a	0	
b	11	c
c	2	a
d	∞	None
e	14	c
f	3	c
g	∞	None
h	∞	None
i	∞	None

The next unvisited shorted distance vertex is f
We are visting f and its unvisited neighbours are e & h and the distances from
- f to e is 3+3=6
- f to h is 3+2=5

The shortest distance from a to e is 6 via e, so update it
Update the previous vertex for each of the updated distances
In this case we visited e and h via f

Visited = [a, c, f]
Unvisited = [b, d, e, g, h, i]

Vertex	**Shortest Distance from a**	**Previous vertex**
a	0	
b	11	c
c	2	a
d	∞	None
e	6	f
f	3	c
g	∞	None
h	5	f
i	∞	None

The next unvisited shorted distance vertex is h
We are visting h and its unvisited neighbours are d, g & i and the distances from

- h to d is 5+4=9
- h to g is 5+1=6
- h to i is 5+11=16

Update the previous vertex for each of the updated distances

Visited = [a, c, f, h]
Unvisited = [b, d, e, g, i]

Vertex	**Shortest Distance from a**	**Previous vertex**
a	0	
b	11	c
c	2	a
d	9	h
e	6	f
f	3	c
g	6	h
h	5	f
i	16	h

The next unvisited shorted distance vertex is g
We are visting g and its unvisited neighbours are d & i and the distances from
- g to d is 6+8=14
- g to i is 6+5=11

Update the previous vertex for each of the updated distances

Visited = [a, c, f, h, g]
Unvisited = [b, d, e, i]

Vertex	**Shortest Distance from a**	**Previous vertex**
a	0	
b	11	c
c	2	a
d	9	h
e	6	f
f	3	c
g	6	h
h	5	f
i	11	g

The next unvisited shorted distance vertex is e
We are visting e and its unvisited neighbours are d and the distances from e to d is 6+2=8

Update the previous vertex for each of the updated distances

Visited = [a, c, f, h, g, e]
Unvisited = [b, d, i]

Vertex	**Shortest Distance from a**	**Previous vertex**
a	0	
b	11	c
c	2	a
d	8	e
e	6	f
f	3	c
g	6	h
h	5	f
i	11	g

The next unvisited shorted distance vertex is d
We are visting d and its unvisited neighbours are b and the distances from d to b is 8+2=10

Update the previous vertex for each of the updated distances

Visited = [a, c, f, h, g, e, d]
Unvisited = [b, i]

Vertex	**Shortest Distance from a**	**Previous vertex**
a	0	
b	10	d
c	2	a
d	8	e
e	6	f
f	3	c
g	6	h
h	5	f
i	11	g

The next unvisited shorted distance vertex is b
There are no unvisited neighbours for b, there is nothing to update

Visited = [a, c, f, h, g, e, d, b]
Unvisited = [i]

Vertex	Shortest Distance from a	Previous vertex
a	0	
b	10	d
c	2	a
d	8	e
e	6	f
f	3	c
g	6	h
h	5	f
i	11	g

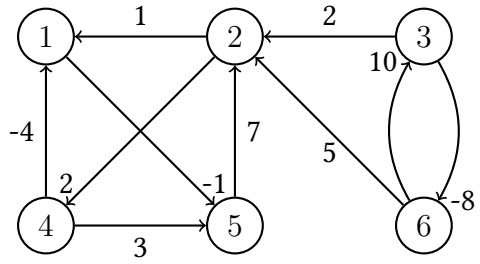
The next unvisited shorted distance vertex is i
There are no unvisited neighbours for i, there is nothing to update

Visited = [a, c, f, h, g, e, d, b, i]
Unvisited = []

Vertex	Shortest Distance from a	Previous vertex
a	0	
b	10	d
c	2	a
d	8	e
e	6	f
f	3	c
g	6	h
h	5	f
i	11	g

Since all the vertices are visited, we can end the algorithm.

2. (20 points) Run $SLOW - ALL - PAIRS - SHORTEST - PATHS$ on the weighted, directed graph given below, showing the matrices that result for each iteration of the loop. Then do the same for $FASTER - ALL - PAIRS - SHORTEST - PATHS$.



Ans:

Initial

$$\begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & 3 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{pmatrix}$$

SLOW ALL PAIRS SHORTEST PATHS

$m = 2$:
$$\begin{pmatrix} 0 & 6 & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & 0 & \infty \\ 3 & -3 & 0 & 4 & \infty & -8 \\ -4 & 10 & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & \infty & 0 \end{pmatrix}$$

$m = 3$:
$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -2 & -3 & 0 & -1 & 2 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 5 & 0 \end{pmatrix}$$

$m = 4$:
$$\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -3 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$$

$m = 5$: $\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$

FASTER ALL PAIRS SHORTEST PATHS

$m = 2$: $\begin{pmatrix} 0 & 6 & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & 0 & \infty \\ 3 & -3 & 0 & 4 & \infty & -8 \\ -4 & 10 & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & \infty & 0 \end{pmatrix}$

$m = 4$: $\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -3 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$

$m = 8$: $\begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{pmatrix}$

3. (10 points) Run the Floyd-Warshall algorithm on the weighted, directed graph given for $Q2$. Show the matrix $D^{(k)}$ that results for each iteration of the outer loop.

Ans:

$D0$ $\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 0 & \infty & \infty & \infty & -1 & \infty \\ 2 & 1 & 0 & \infty & 2 & \infty & \infty \\ 3 & \infty & 2 & 0 & \infty & \infty & -7 \\ 4 & -4 & \infty & \infty & 0 & 3 & \infty \\ 5 & \infty & 7 & \infty & \infty & 0 & \infty \\ 6 & \infty & 5 & 10 & \infty & \infty & 0 \end{bmatrix}$

$$D1 \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 0 & \infty & \infty & \infty & -1 & \infty \\ 2 & 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & \infty & 2 & 0 & \infty & \infty & -7 \\ 4 & -4 & \infty & \infty & 0 & -5 & \infty \\ 5 & \infty & 7 & \infty & \infty & 0 & \infty \\ 6 & \infty & 5 & 10 & \infty & \infty & 0 \end{bmatrix}$$

$$D2 \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 0 & \infty & \infty & \infty & -1 & \infty \\ 2 & 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 3 & 2 & 0 & 4 & 2 & -7 \\ 4 & -4 & \infty & \infty & 0 & -5 & \infty \\ 5 & 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 6 & 5 & 10 & 7 & 5 & 0 \end{bmatrix}$$

$$D3 \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 0 & \infty & \infty & \infty & -1 & \infty \\ 2 & 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 3 & 2 & 0 & 4 & 2 & -7 \\ 4 & -4 & \infty & \infty & 0 & -5 & \infty \\ 5 & 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 6 & 5 & 10 & 7 & 5 & 0 \end{bmatrix}$$

$$D4 \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 0 & \infty & \infty & \infty & -1 & \infty \\ 2 & -2 & 0 & \infty & 2 & -3 & \infty \\ 3 & 0 & 2 & 0 & 4 & -1 & -7 \\ 4 & -4 & \infty & \infty & 0 & -5 & \infty \\ 5 & 5 & 7 & \infty & 9 & 0 & \infty \\ 6 & 3 & 5 & 10 & 7 & 2 & 0 \end{bmatrix}$$

$$D5 \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 0 & 6 & \infty & 8 & -1 & \infty \\ 2 & -2 & 0 & \infty & 2 & -3 & \infty \\ 3 & 0 & 2 & 0 & 4 & -1 & -7 \\ 4 & -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 5 & 7 & \infty & 9 & 0 & \infty \\ 6 & 3 & 5 & 10 & 7 & 2 & 0 \end{bmatrix}$$

$$D6 \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 0 & 6 & \infty & 8 & -1 & \infty \\ 2 & -2 & 0 & \infty & 2 & -3 & \infty \\ 3 & -4 & -2 & 0 & 0 & -5 & -7 \\ 4 & -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 5 & 7 & \infty & 9 & 0 & \infty \\ 6 & 3 & 5 & 10 & 7 & 2 & 0 \end{bmatrix}$$

$$D7 \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 0 & 6 & \infty & 8 & -1 & \infty \\ 2 & -2 & 0 & \infty & 2 & -3 & \infty \\ 3 & -4 & -2 & 0 & 0 & -5 & -7 \\ 4 & -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 5 & 7 & \infty & 9 & 0 & \infty \\ 6 & 3 & 5 & 10 & 7 & 2 & 0 \end{bmatrix}$$

4. (15 points) Suppose you are given a connected weighted undirected graph, G, with n vertices and m edges, such that the weight of each edge in G is an integer in the interval $[1, c]$, for a fixed constant $c > 0$. Show how to solve the single-source shortest paths problem, for any given vertex v, in G, in time $O(n + m)$.

   Hint: Think about how to exploit the fact that the distance from v to any other vertex in G can be at most $O(cn) = O(n)$.

   Ans: There are many ways to solve this problem in linear time. We will talk about the easiest to understand.

   The main idea is to transform the weighted graph into an unweighted one. If the number of nodes and edges of the transformed graph remains linear to n and m, then we can run BFS to solve it in $\mathcal{O}(n + m)$.

   The transformation is as follows: for each edge (u, v) of weight w, create $(w1)$ nodes to connect (u, v). The transformation is as follows: for each edge (u, v) of weight w, create $(w1)$ nodes to connect (u, v). For example an edge (a, b) of length 3, we create two $(edgeweight - 1)$ dummy nodes to normalize the edge weights and replace the original edge with $a \to d1 \to d2 \to b$

   It is easy to check that this transformation preserves the shortest path between any pairs of nodes. The number of nodes is now $\mathcal{O}(n + (c - 1)m) = O(n + m)$ and the number of edges is $\mathcal{O}(cm) = \mathcal{O}(m)$. Since BFS is linear to the number of nodes and edges, we have a runtime of $\mathcal{O}(n + m + m) = \mathcal{O}(n + m)$.

5. (10 points) How can we use the output of the Floyd-Warshall algorithm to detect the presence of a negative-weight cycle?

   Ans:

   The output matrix of the Floyd-Warshall algorithm helps to find whether there is any negative-weight cycle in a graph.

   Floyd-Warshall algorithm is used to find all-pairs shortest paths. That is, for each pair of vertices (i, j), a shortest path will be calculated. Floyd-Warshall algorithm returns a matrix $(D(n))$ that represents all the shortest paths between each pair of vertices (i, j).

   Since a shortest path is a simple path that has no repeating vertices, if there is a diagonal value with $i = j$ (repeating vertex) and if the weight is less that 0, then there is a cycle with negative weight in the graph.

   Thus, check the main-diagonal entries of the resultant matrix to find whether there is any negative value. If there is a negative value, for some vertex i, then there is a path weight

from i to itself. If it is a negative value, there is a path from i to itself (i.e., cycle), with negative weight.

Therefore by observing diagonal entries of the output matrix of floyd warshall algorithm, it is possible to detect whether there is a negative-weight cycle or not.

```
for i in range(V):
    if (dist[i][i] < 0):
        return True
return False
```

6. (10 points) Dijkstra's algorithm is not used in graph's which contain negative weights since a negative edge can break the algorithm. A workaround to this is that when a graph contains negative weights, we increase all the edge weights by adding to each edge the absolute value of the lowest edge weight of the graph. This makes all the weights positive.

   For example, if the weights are $w1 = -7$, $w2 = -3$, $w3 = 2$, we can ensure no edge has a negative weight by adding 7 to each edge. Thus $w1 = 0$, $w2 = 4$, $w3 = 9$. As the weights are non-negative, we can now apply Dijkstra's.

   The transformation can be written as:

   $G = (V, E) \implies G' = (V', E')$
   $E' = E$
   $V' = V$
   $w'(u, v) = w(u, v) + |m|$
   where m is the smallest weight of any edge in the graph.

   (a) (2 points) Does this approach give correct results? (Yes/No)

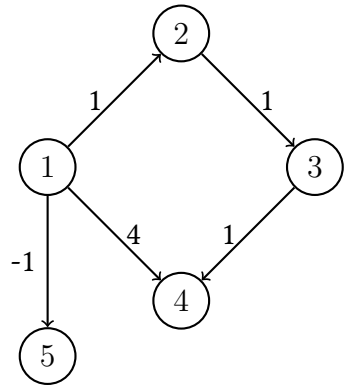
   Ans: No

   (b) (8 points) If yes, what is the complexity of the transformation and if no, provide reasoning or counter-example why it doesn't work.

   Ans:

   Explanation

   We are selecting a straightforward directed network with just one node and two edges leading to source vertex 1. So that we are merely demonstrating why strategy fails, we do not need to implement Djikstra.

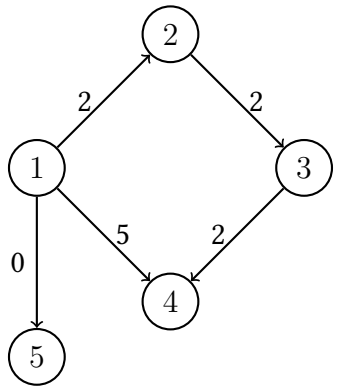


From 1 to 4, we have 2 paths

- 1st path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ with weight 3
- 2nd path $1 \rightarrow 4$ with weight 4

The 1st path is shorter here.

Also, as -1 is minimum negative edge weight, we add +1 to all the edges. The graph after modification is below.



From 1 to 4, we have 2 paths again

- 1st path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ with weight 6
- 2nd path $1 \rightarrow 4$ with weight 5

The 2nd path is shorter in the above case.

In the given graph, the shortest path is the 1st. And in the modified graph, the shortest path is the 2nd.

Shortest path should not change if we apply modifications to negative edge weight graph in djisktra but its changing in our example. This is contradiction.

Hence, proved.

7. (15 points) Suppose you are given a timetable, which consists of the following:

   - A set A of n airports and for each airport $a \epsilon A$, a minimum connecting time c(a).
   - A set F of m flights and the following attributes for each flight $f \epsilon F$:
     - Origin airport $a1(f) \epsilon A$
     - Destination airport $a2(f) \epsilon A$
     - Departure time $t1(f)$
     - Arrival time $t2(f)$

   (a) (12 points) Describe an efficient algorithm/pseudocode for the flight scheduling problem. In this problem, we are given airports a and b and time t. We wish to compute a sequence of flights that allows one to arrive at the earliest possible time in b when departing from a at or after time t. Minimum connecting times at intermediate airports should be observed.

   Ans:

   With the following properties, this problem can be transformed into a graph traversal problem with :

i. Let us consider airports as nodes and each flight as an edge in a graph. The flight time would be considered as weight of the edge.

ii. Now we need to find sequence of flights from source (which starts after time t) to destination with that allows to arrive at earliest possible time.

iii. To solve this we can apply Dijkstra's Algorithm to get sequence of flights between a and b with earliest possible time.

iv. While applying Dijkstra's, when we reach a particular airport, we would check the outgoing flights start time from that airport with the time to reach this airport. Consider only flights which have start time greater than the time taken to reach this airport.

v. Upon reaching the destination, we abort the algorithm as we would have got the time to reach target from source.

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:
        dist[v] := infinity
        //initial distance from source to vertex v
        // is set to infinite
        previous[v] := undefined
        // Previous node in optimal path from source
    dist[source] := t   // starting time at source
    Q := the set of all nodes in Graph
    while Q is not empty:        // main loop
        u := node in Q with smallest dist[ ]
        if (u == b):
            // if the node is our target airport,
            // then we return the distance
            return dist[u]
        remove u from Q
        for each neighbor v (edge e) of u:
        // where v has not yet been removed from Q.
            // consider only flights which have start time stamps
            // greater than current_time sofar to reach this airport
            if(dist[u] < e[0]) // e[0] gives start time of flight
                // e[1] gives end time of flight
                // c(v) gives connecting time at airport v
                alt := dist[u] + e[1] - e[0] + c(v)
                if alt < dist[v]        // Relax (u,v)
                    dist[v] := alt
                    previous[v] := u
```

(b) (3 points) What is the running time of your algorithm as a function of n and m?

Ans: $\mathcal{O}(E + V(Log(V)))$

8. (10 points) You are working for a ISP providing service in a relatively undeveloped country. A wealthy customer wants a highly reliable and stable connection from their house to the router that connects your network with the rest of the internet. In order to please this valued customer without spending money on expanding the ISP's infrastructure, your boss has tasked you with finding a dedicated path through the network to route this customer's traffic. You, naturally, want this channel to have only the most reliable links. You have a directed graph, labeled with the chance of a given router (vertex) or link (edge), to go down on a given day.

   Design a algorithm/pseudocode so that you can quickly compute this and other reliable routes through the network.

   Ans:

   Here, each router is different vertex/node and the connection between any two routers is a link(edge). Weight of each link states that possibility/chances of network going down on given day.

   Any path that connects customer's router to ISP is called as reliable path and there might be many such paths/reliable paths. We can say it as **most reliable path** if and only if the total chance/possibility of network going down is minimal.

   We can use DFS to compute all the reliable paths between given source node (ISP) and customer's router.

```
def printAllReliablePaths(self, u, d, visited, path):
        visited[u]= True
        path.append(u[0])
        weight+= u[1]
        if u == d:
            print (path)
            print(weight)
        else:
            for i in self.graph[u]:
                if visited[i]== False:
                    self.printAllPathsDFS(i, d, visited, path)

        weight-=u[1]
        path.pop()
        visited[u]= False
```

   After computing all the reliable paths from the above function(printAllReliablePaths), return the path which has total sum of edge (link) weights as minimum and that would be the most reliable path (that is the minimum chance of internet going down).

   If we want only the most reliable path, we can use Dijikstra algorithm alone to compute the path between ISP and customer's/enduser router.