Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

1) Answer to Question 1:

Part (a)

Formulating the problem as a Graph Theoretic Problem:
We can convert the given problem into a graph problem as follows.
We imagine the intersections in the city to be the vertices of our graph.
If there exists a road between two intersections, we denote the roads as edges between the corresponding two vertices. Moreover, since the roads are given to be one-way, our edges will be directed. We assume that there are no roads from an intersection to itself, i.e., there are no self loops.
Therefore, we have constructed a directed graph where the edges denote a one way road between two vertices or intersections. Let the resultant graph be G(V,E).
We store the graph using adjacency lists.
Also note that if there are multiple edges between two vertices, we may remove the duplicate edges as they do not affect the reachability between two vertices.

The mayor claims that it is possible to legally drive from one intersection to any other intersection.
In terms of graphs, she claims that - our directed graph has one and only one strongly connected component (there exists a path between any two pair of vertices in the graph).

Explanation of Graph Algorithms and How They're Used:
We will use Kosraju's Algorithm to find the number of strongly connected components in the graph.
If we get 1 strongly connected component from Kosaraju's algorithm, then we return true (the mayor's claim is correct) otherwise we return false (the mayor's claim is false).

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

Explanation of Kosaraju's Algorithm -

1. Run DFS on the graph G and sort the vertices according to their finishing times (in descending order) by pushing each vertex onto a stack when it is completely explored.
2. Find the transpose (G^T) of graph G by reversing the directions of all the edges in the graph.
3. Run DFS on G^T from each vertex according to the order in which they are stored on the stack (by popping the stack until it is empty).

If in 3. we have to run DFS from more than one vertex, then it means that our graph does not have only one strongly connected component and therefore the mayor's claim is false. Otherwise, the claim is true.

Pseudocode:

```
DFS_Step1(G)
    stack ← $\boldsymbol{\Phi}$
    For every u $\in$ V(G) do
        Mark u as unvisited
        parent(u) = NIL;
    End
    For every u $\in$ V(G) do
        If u is unvisited do
            DFS-Visit_Step1(G,u,stack)
        End
    End
    return stack
}
```

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

```
DFS-Visit_Step1(G,s,stack){
    time ← time + 1
    start[s] = time
    Mark s visited
    Colour s with grey
    For every u ∈ Adj(s) do
        If u is unvisited do
            parent(u) = s
            DFS-Visit_Step1(G,u,stack)
        End
    End
    Mark s as explored
    Colour s black
    time ← time + 1
    finish[s] = time
    push(stack,s)
}


Rev-Graph(G){
    For all u ∈ V do
        Initialize $Adj_r[u]$ ← $\Phi$
    End
    For all u ∈ V do
        For all v ∈ Adj[u] do
            Insert u into $Adj_r[v]$
        End
    End

    return (vertices set V, $Adj_r[v]$ for all v ∈ V)
}
```

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

```
DFS_Step2(G,stack){
    countscc ← 0
    For every u ∈ V(G) do
        Mark u as unvisited
        parent(u) = NIL;
    End
    while stack ≠ Φ do
        u ← pop(stack)
            If u is unvisited do
                DFS-Visit_Step2(G,u)
                countscc ← countscc + 1
            End
        End
    End
    return countscc
}


DFS-Visit_Step2(G,s){
    Mark s visited
    Colour s with grey
    For every u ∈ Adj(s) do
        If u is unvisited do
            parent(u) = s
            DFS-Visit_Step2(G,u)
        End
    End
    Mark s as explored
    Colour s black
}
```

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

```
main(G){
    stack  ← DFS_Step1(G)
    G_rev ← Rev-Graph(G)
    countscc ← DFS_Step2(G_rev,stack)
    If countscc = 1 do
        return true
    End-if
    Else do
        Return false
    End-else

}
```

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

Explanation Of Why the Algorithm is Correct:

The graph must have only one strongly connected component for the mayor's claim
to hold true by the following contradiction:

Let us assume that our graph does not have only one strongly connected component and there exists a path between any two vertices.
Let us consider the case that we have two strongly connected components G_1 and G_2, then by the fact that there is not a single strongly connected component only, there exist at least one pair of vertices $v_1 \in G_1$ and $v_2 \in G_2$ such that at least one of the paths from (v_1 to v_2) or (v_2 to v_1) does not exist.
Thus, there exists a pair of vertices such that there is no legal path to reach from one to the other. [*]
However, this is a contradiction. Therefore, our claim that our graph does not have only one strongly connected component is false.
The same logic can be used to prove why we cannot have more than $n \geq 2$ strongly connected components.

Hence, the graph must have only one strongly connected component.

Now, Kosaraju's algorithm gives us the strongly connected components present in the graph G. Therefore, while running the final DFS, we simply have to check how many times we are calling the DFS-Visit function. If we call it only once, we have one strongly connected component and we return true, otherwise we return false.

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

Running Time:

1. In Kosaraju's Algorithm we sort the edges in the decreasing order of their finishing times using DFS which has time complexity O(|V| + |E|).
2. Then we find G^T by iterating over all vertices and their edges which has time complexity O(|V| + |E|).
3. Then we run DFS again to find the number of strongly connected components which has time complexity O(|V| + |E|).

4. Then we just check how many strongly connected components we have in O(1).

Therefore, our final time complexity is O(|V| + |E|).

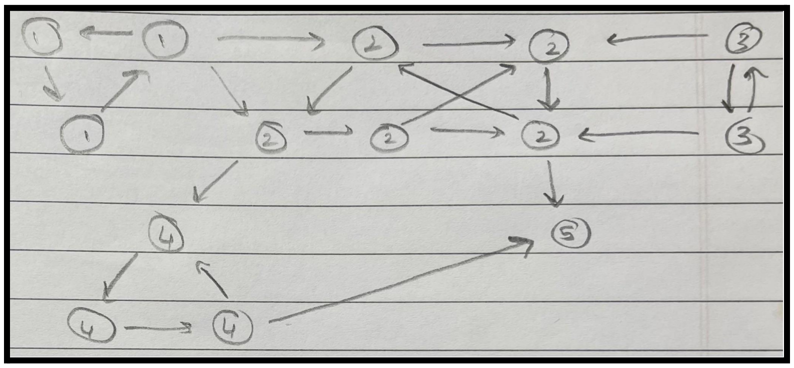
Theory Assignment-1

Roll No: 2021055 **Section: A**

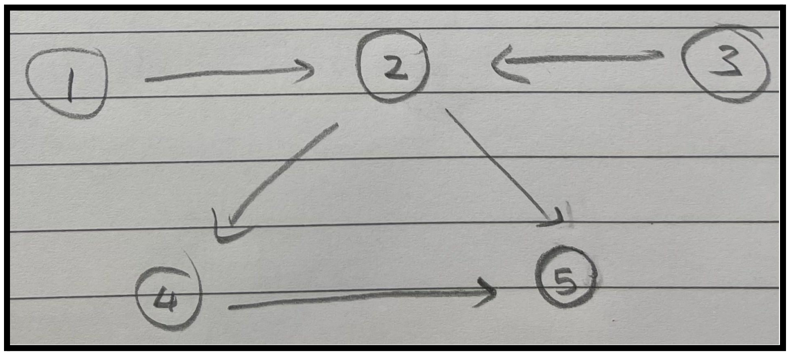
Name: Jyotirmaya Singh **Specialisation: CSE**

Part (b)

Formulating the problem as a Graph Theoretic Problem:
Graph is constructed the same way as done in (a).
The mayor claims that if we start driving from the town hall, then no matter where we reach, there is always a way to legally drive back to the town hall.
We define a condensation graph as follows -
The condensation graph of a directed graph G is defined as the directed graph G^{SCC} whose nodes are the SCCs (strongly connected components) of G and whose edges are defined as (C_i,C_j) (where C_i and C_j are vertices in the G^{SCC}) is an edge in G^{SCC} iff there is an edge in G from any node in C_i to any node in C_j.
In terms of graphs, she claims that - the town hall belongs to an SCC say C_k in G^{SCC} such that there is no outgoing edge from C_k.



A directed graph with SCCs marked using numbers



Corresponding condensation graph

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

Explanation of Graph Algorithms and How They're Used:

1. We will use Kosaraju's algorithm to find all the SCCs of the given graph G. Explanation of Kosaraju's Algorithm -
   (a) Run DFS on the graph G and sort the vertices according to their finishing times (in descending order) by pushing each vertex onto a stack when it is completely explored.
   (b) Find the transpose (G^T) of graph G by reversing the directions of all the edges in the graph.
   (c) Run DFS on G^T from each vertex according to the order in which they are stored on the stack (by popping the stack until it is empty). Store the vertices visited in each DFS run into a list and then store each of these lists into a list of lists.
2. We find out the number of vertices in the SCC of the town hall.
3. We run DFS from the town hall vertex and count the number of vertices it visits.
4. If the number of vertices in (2) and (3) are equal then we return true (mayor's claim is correct), otherwise we return false (mayor's claim is not correct).

Pseudocode:

```
DFS_Step1(G)
    stack ← $\Phi$
    For every u ∈ V(G) do
        Mark u as unvisited
        parent(u) = NIL;
    End
    For every u ∈ V(G) do
        If u is unvisited do
            DFS-Visit_Step1(G,u,stack)
        End
    End
    return stack
```

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

```
}
```

```
DFS-Visit_Step1(G,s,stack){
    time ← time + 1
    start[s] = time
    Mark s visited
    Colour s with grey
    For every u ∈ Adj(s) do
        If u is unvisited do
            parent(u) = s
            DFS-Visit_Step1(G,u,stack)
        End
    End
    Mark s as explored
    Colour s black
    time ← time + 1
    finish[s] = time
    push(stack,s)
}
```

```
Rev-Graph(G){
    For all u ∈ V do
        Initialize Adj_r[u] ← Φ
    End
    For all u ∈ V do
        For all v ∈ Adj[u] do
            Insert u into Adj_r[v]
        End
    End

    return (vertices set V, Adj_r[v] for all v ∈ V)
}
```

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

```
DFS_Step2(G,stack){
    big_list ← $\Phi$
    For every u ∈ V(G) do
        Mark u as unvisited
        parent(u) = NIL;
    End

    while stack ≠ $\Phi$ do
        small_list ← $\Phi$
        u ← pop(stack)
        If u is unvisited do
            DFS-Visit_Step2(G,u,small_list)
        End
        append(big_list,small_list)
    End
    return big_list
}


DFS-Visit_Step2(G,s,small_list){
    Mark s visited
    Colour s with grey
    For every u ∈ Adj(s) do
        If u is unvisited do
            parent(u) = s
            DFS-Visit_Step2(G,u)
        End
    End
    Mark s as explored
    Colour s black
    append(small_list, s)
}
```

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

```
DFS_Step3(G, v)
    stack ← $\Phi$
    For every u $\in$ V(G) do
        Mark u as unvisited
        parent(u) = NIL;
    End
    DFS-Visit_Step1(G,v,stack)
    return stack
}

main(G, $v_{townhall}$){
    stack  ← DFS_Step1(G)
    G_rev ← Rev-Graph(G)
    list ← DFS_Step2(G_rev,stack)

    For each $G^{SCC}$ $\in$ list do
        For each v $\in$ $G^{SCC}$ do
            If v = $v_{townhall}$ do
                $length_{SCC}$ ← length($G^{SCC}$)
            End-If
        End
    End

    stack ← DFS_Step3(G,$v_{townhall}$)

    If $length_{SCC}$ = length(stack) do
        return true
    End-if
    Else do
        Return false
    End-else
}
```

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

Explanation Of Why it is Correct:

Let us prove a few properties of G^{SCC}s for clarity and ease.

Claim I: The G^{SCC} is a DAG (directed acyclic graph).
Proof: Let us prove this by contradiction. We assume that there exist at least two distinct vertices C_i, C_j in G^{SCC} such that there exists a cycle between them. However, now due to the cycle we can reach from any node in C_i to any node in C_j and vice versa.
Therefore, C_i and C_j are not distinct SCCs which is a contradiction.
Hence G^{SCC} is a DAG (directed acyclic graph).

Claim II: There must be at least one vertex in G^{SCC} such that there is no outgoing edge from it.
Proof: Since G^{SCC} is a DAG, we must be able to find a topological ordering for it. Since a topological ordering must end at some vertex, there exists at least one vertex in G^{SCC} such that there is no outgoing edge from it. Another way to look at it, is that there may be more than one vertices such that there is no outgoing edge from them, however, the last vertex in the topological ordering must be a vertex with no outgoing edge.

We claim that the town hall must belong to a vertex in the G^{SCC} which has no outgoing edge from it.
Proof: The town hall must belong to a vertex say C_t in the G^{SCC}. We look at the two possible cases of outgoing edges-

(a) There is no outgoing edge from C_t.
(b) There is an outgoing edge from C_t to another vertex, say C_k.
If C_k does not have an outgoing edge, then it is clear that there exists a vertex $v \in C_k$ such that there does not exist a path from v to the town hall.
Assume there is an outgoing edge from C_k, we know that there will exist at least one vertex in the G^{SCC} say C_n, with no outgoing edge (proved in Claim II) which will either be connected to C_k directly or through some intermediary set of vertices say $C_1, C_2, \ldots, C_I$ which are connected to C_k. Graphically the three cases are -
$C_t \rightarrow C_k$ OR $C_t \rightarrow C_k \rightarrow C_n$ OR $C_t \rightarrow C_k \rightarrow C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_I \rightarrow C_n$
Therefore, there exists a vertex $v \in C_n$ such that there exists a path from the town hall to v, but there does not exist a path from v to the town hall.

Therefore, the mayor’s claim is proved to be false.

Thus, the only case where the mayor’s claim may be true is in (a) when there is no outgoing edge from C_t. Since C_t is an SCC, then by definition we can find a path between any two nodes belonging to C_t. Furthermore, since we cannot travel outside the SCC (due to no outgoing edge from C_t), it is evident that we cannot reach any node from the town hall, such that we cannot find a way back to the town hall from that node.
Hence, the town hall must belong to a vertex in G^{SCC} such that there is no outgoing edge from it.

Once we have found the town hall’s SCC, we count the number of vertices present in it. Say we found ‘n’ vertices.
When we run a DFS from the town hall we are guaranteed to visit at least n vertices. If we visit more than n vertices, it means that the SCC has at an outgoing edge.
Therefore, if the number of vertices in the town hall’s SCC and the number vertices visited in running a DFS from the town hall are the same, then the mayor’s claim is true, otherwise it is false.

Running Time:

1. In Kosaraju’s Algorithm we sort the edges in the decreasing order of their finishing times using DFS which has time complexity O(|V| + |E|).
2. We first find out which SCC the town hall belongs to by iterating over all the SCCs and checking which one the town hall is in. This takes O(|V|) time.
3. We find out the number of vertices in the town hall’s SCC in O(|V|) time.
4. We run DFS from the town hall and count the number of vertices it visits. DFS takes O(|V| + |E|).
5. We find out the number of vertices visited in the DFS in (4) in O(|V|) time.
6. We compare the two values in (3) and (5) in O(1).

Therefore, our final time complexity is O(|V| + |E|).

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

2) Answer to Question 2:

Formulating the problem as a Graph Theoretic Problem:
Already given. We are assuming the graph to be simply connected.

Explanation of Graph Algorithms and How They're Used:
As we know, in an undirected connected graph a bridge edge is an edge such that removing it disconnects the graph.
Due to this property of bridge edges, a bridge edge can't belong to a cycle in a graph as if it does belong to a cycle, removing that edge won't disconnect the graph.
Since, in the question we have been asked to find the minimum weighted edge out of all the edges belonging to a cycle, we will use an algorithm that will remove all the bridge edges in the graph using the **Tarjan's algorithm** as we don't need to consider those edges for calculating the minimum weighted edge and then find the edge having minimum weight out of all the remaining non-bridge edges as all of these edges belong to one or more of the available cycles in the graph.

Explanation of Algorithm:

1) First we'll create 4 arrays, each of length equal to the number of vertices. The first array would be storing the arrival time of every vertex (arr1), the second array would be storing the minimum arrival time of the adjacent vertices (other than parent) for a particular vertex (arr2) , a third array for checking whether the vertex has been previously visited or not and there also will be a fourth array for tracking the parent of each vertex. Apart from these arrays we'll also maintain a global time counter initialised to 0 and an empty hashMap which will store the bridge edges of the graph (if any).
2) After the above initializations, we'll start the DFS traversal from any unvisited vertex. For each DFS call, we'll first mark the vertex (say, v) as visited and then mark the arrival time and the min arrival time as the same value as the global time counter of the vertex in the respective array and

increment the global time counter. Then we'll check for all univisited neighbours of the vertex and call the DFS traversal function for that vertex.

3) After all the neighbours have been explored, the vertex checks the minimum arrival time for each of its neighbours except the parent (say, u) and selects the minimum of all those times as its minimum arrival time. So, it basically marks the min arrival time of the vertex as the min of the min arrival time of all its neighbours excluding its parent.
4) Now, while returning back from the function call, for each edge connected to the current vertex (v), it compares the min arrival time of the neighbour (u) to the arrival time of the current vertex (v) itself, if the min arrival time of the neighbour (u) is larger than the arrival time of the current vertex (v), that indicates that it is a bridge edge. So, store this edge in the bridgeEdge hashMap to preserve this information.
5) After the DFS traversal is completed, we now have a list of all the bridge edges that are present in the graph. Since, we know that a cycle can't contain a bridge edge, the minimum weighted edge in the graph that is part of a cycle is the minimum of all the edges except the bridge edges in the graph.

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

Pseudocode:

```
Initialize visited array of size V
Initialize arrivalTime array of size V
Initialize minArrivalTime array of size V
Initialize parent array of size V
Initialise an empty bridgeEdge hashMap
time ← 0
For each vertex v do
    if(!visited[v]) do
        DFS(v)
    End
End

DFS(v, visited, arrivalTime, minArrivalTime, parent){
    Mark v as visited
    Mark the arrivalTime of v as time
    Mark the minArrivalTime of v as time
    time ← time + 1
    For each neighbour u of v do
        if(u is parent[v]) do
            continue
        End
        if(!visited[u]) do
            parent[u]=v
            DFS(u, visited, arrivalTime, minArrivalTime, parent)
            minArrivalTime[v] ← min(minArrivalTime[v], minArrivalTime[u])
            if (minArrivalTime[u]>ArrivalTime[v]) do
                Add the edge (u, v) to the bridgeEdge hashMap
        End
        Else do
            minArrivalTime[v] ← min(minArrivalTime[v], minArrivalTime[u])
        End

}
```

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

```
min=INFINITY
For each edge (u, v) in E not in bridgeEdge do
    if(weight(u, v)<min) do
        min=weight(u, v)
print(min)
```

Explanation Of Why it is Correct:

Claim: A bridge edge cannot belong to a cycle.
Proof: Let us prove our claim by contradiction. Consider two graph sub-components $G_1(V_1,E_1)$ and $G_2(V_2,E_2)$ such that $V_1,V_2 \subset V$ and $E_1,E_2 \subset E$. Assume that a bridge edge

E

between G_1 and G_2 belongs to a cycle. Then there exists another edge *E'* between G_1 and G_2 such that we can start from G_1 (or G_2) and return back to the same sub-component. However, if E' exists, then removing E from the graph does not disconnect G_1 and G_2. Hence, E is not a bridge edge. This is a contradiction.
Therefore, our assumption that a bridge edge can belong to a cycle was wrong.

Thus, a bridge edge cannot belong to a cycle.

Claim: A non-bridge edge must belong to a cycle.
Proof: Consider a non-bridge edge E between two graph sub-components $G_1(V_1,E_1)$ and $G_2(V_2,E_2)$ such that $V_1,V_2 \subset V$ and $E_1,E_2 \subset E$. Since E is a non-bridge edge, it implies that removing *E* from the graph will not disconnect G_1 and G_2. Therefore, there must exist at least one other edge (say *E'*) between G_1 and G_2 so that the graph remains connected upon the removal of E. Since E and E' are edges between G_1 and G_2, it implies that we can start from any vertex in G_1 (or G_2) traverse through one of the edges (say E) to G_2 and back to G1 via the other (say E'). Therefore, a non bridge edge must belong to a cycle.

Therefore, since all bridge edges do not form a part of a cycle and all non-bridge edges do form a part of some cycle, removing all bridge edges from the graph and then finding the minimum weighted edge from the remaining non-bridges edges will give us the

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

required answer.

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

Claim: If for an edge (u, v), the minArrivalTime[v]>arrivalTime[u], then (u, v) is a bridge edge.

Definitions used:
Back Edge - We define a back edge as an edge that is connected to a vertex that is discovered before its parent in the DFS traversal.

Justification: The claim given above means that the arrivalTime or in other words, the discovery time of the vertex u is lesser than the minArrivalTime of all the neighbours of the vertex v. Had it been not the case, we could have concluded that the vertex v has a descendant that connects itself to one of the ancestors of vertex u thus signifying that it has a back edge. Since the existence of a back edge proves that the edge (u, v) can't be a bridge edge, and by the given situation we know that there is no back edge, we can conclude that the edge (u, v) is one of the bridge edges of the graph.

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

Running Time:

1. Initializing the parent, visited, arrivalTime and minArrivalTime of size V takes O(|V|) time.
2. Since, we are performing a simple DFS traversal with some slight modifications in the conditions and computations which are of constant time complexity, the time complexity effectively boils down to that of the DFS algorithm which is O(|V| + |E|).
3. Finally, calculating the minimum valued edge that is not in *bridgeEdge* is done in O(|E|) since we are iterating over all edges in O(|E|) time and checking whether an edge belongs to *bridgeEdge* or not in O(1) using the hashmaps.

Since, in the question it is given that the number of edges is the number of vertices + 20, The time complexity becomes O(|V| + |V| + 20) which equals O(2|V|+20) which approximates to O(|V|).
Therefore, our final time complexity is O(|V|) or O(n) where n or V represents the n of vertices in the graph which is linear time.

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

3) <u>Answer to Question 3</u>:

Preprocessing Stage: Let the given DAG graph be G(V, E).
Store the topological sort of the graph G as $\sigma = (v_1, v_2, \dots, v_n)$.
We store the edges of the graph using adjacency lists where
$Adj[v] = [\ (u_1,w_1), (u_2,w_2), \dots, (u_n,w_n)\ ]$
denotes that a directed edge exists between $(v, u_1), (v, u_2), \dots, (v, u_n)$ having weights $w_1, w_2, \dots, w_n$ respectively.
No self loops are allowed since the graph is acyclic.
Also, if there are multiple edges from a vertex v to a vertex u, we sum the weights of all the edges and assign it to a single edge between v and u.

Subproblem: A brute force approach would be to calculate the probability of each possible path to a sink and then summing up the probabilities of all paths ending at (say) sink t_k to get the total probability of reaching sink t_k.
We define $dp[v_i]$ as the total probability of reaching vertex v_i from any possible path starting from source *s* and ending at v_i.

Recurrence:
Let (u_i,v_i) be a directed edge from u_i to v_i having weight/probability = $Pr[u_i \rightarrow v_i]$.

$$dp[v_i] = dp[u_i] * Pr[u_i \rightarrow v_i] + dp[v_i]$$

Base Cases:

$$dp[s] = 1$$

Subproblem that Solves the Original Problem:

It is $dp[t_k]$ depending on which sink's probability we want to find.

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

Pseudocode:

```
For all $v \in V$ do            - - - O(|V|)
    dp[v] = 0           // Initialize dp to 0
End
dp[s] = 1                       //base case

$\sigma = (v_1, v_2, \dots, v_n) \leftarrow$ a topological ordering of G        - - - O( |V| + |E| )
For all $v \in \sigma$ do                          - - - O(|E|)
    For all pairs (u,w) $\in$ Adj[v] do
        dp[u] = dp[v] * w + dp[u]
    End
End

return dp        // return the array
OR
return dp[$t_k$]    // depending on which sink we want, return the answer
```

Explanation:

Say we have non sink vertices $v_1, v_2, \dots, v_m$ from which we can reach sink t_k in one step. In order to calculate the total probability of reaching sink t_k, we must have the total probabilities of reaching each of $v_1, v_2, \dots, v_m$.

The total probability of reaching t_k is then =

TotalProb(v_1) * weight(v_1, t_k) + TotalProb(v_2) * weight(v_2, t_k) + … + TotalProb(v_k) * weight(v_m, t_k)

In general, the total probability of reaching a vertex ‘v’ must be calculated by taking into account all the edges that are of the form (u,v), i.e., all incoming edges to v.

Therefore, finding a topological sort makes intuitive sense as for every edge of the form (u,v) , u comes before v in the topological sort. We then calculate the total probability of reaching each non-source vertex using the recurrence relation. Also, since the probabilities of taking any edge are given to be mutually independent, we

can simply multiply the probabilities to calculate the probability of taking that particular path.

Running Time:

1. Initializing the dp array to 0 takes O(|V|) time as we are iterating over |V| vertices.
2. Topological sort takes O(|V| + |E|) time.
3. The outer For loop iterates over all vertices and the inner For loop iterates over all the edges of each vertex. Therefore, we are essentially iterating over all the edges of the graph, the outer For loop is only for iterating over the edges according to the topologically sorted order. Therefore, both the For loops take a total of O(|V| + |E|) time.

Therefore, our final time complexity is O(|V| + |E|).