Theory Assignment-2: ADA Winter-2023

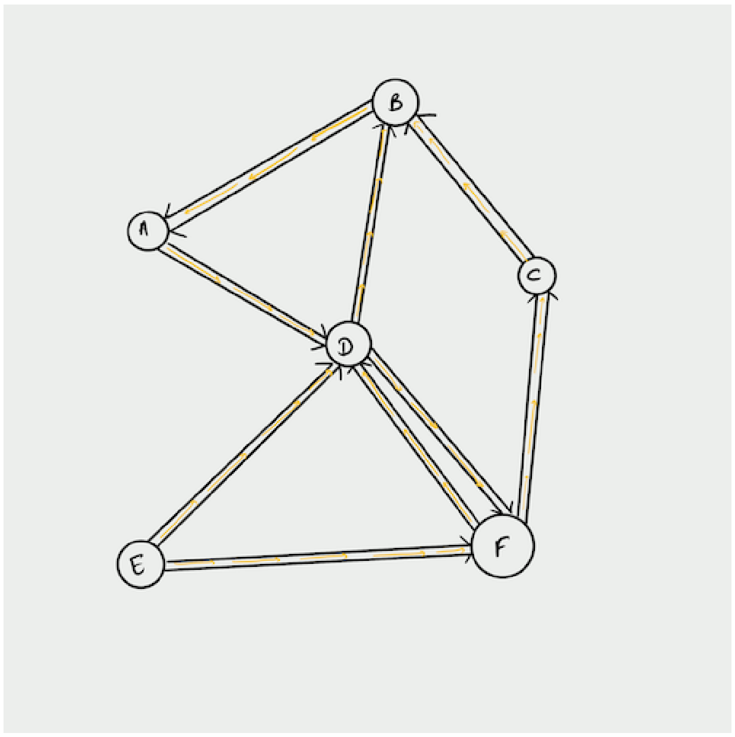
Shivoy Arora (2021420) Chehak Malhotra (2021141)

1. The police department in the city of Computopia has made all the streets one-way. But the mayor of the city still claims that it is possible to legally drive from one intersection to any other intersection.

   (a) Formulate this problem as a graph theoretic problem and explain why it can be solved in linear time.

Ans:

(a) We can formulate this problem as a graph theory problem by representing each **intersection as a vertex** in the graph and each **one-way street as a directed edge** connect the first intersection(vertex) to the second intersection(vertex).



Like in the example above A, B, C, D, E, F are the intersections that are represented as vertices in the graph such as $V(G) = \{A, B, C, D, E, F\}$ and the streets like $A \to D$ are the edges in the graph such as $E(G) = \{A \to D, C \to B, \cdots\}$

(b) We can use the **kosaraju's algorithm**(done in class) in this **DAG**(Directed Acyclic Graph) to find the strongly connected components in the graph in linear time. If all the vertices are strongly connected then it is possible to drive from any intersection to any other intersection.

The output by the kosaraju algorithm should be only one SCC(Strongly Connected Component) as kosaraju give all the SCC

(c) The **kosaraju's algorithm** for checking mayor's claim in linear time works as the mayor claimed that it is possible to legally drive from one intersection to another in, i.e., it is possible to reach from any vertex to any other vertex in the graph, where the roads are one way, i.e., the edges are directed. So, we have to check whether all the vertices in the graph are strongly connected.

(d) **Run Time**

Time Complexity of the algorithm is $O(V+E)$ (linear) as this algorithm is DFS based. It does DFS two times on the graph, and also changes the the direction of all the edges that would require $O(E)$ time, and DFS takes $O(V+E)$ time, so the total time complexity is $O(V+E)$.

Space Complexity of the algorithm is $O(V)$ as we use a stack to store the vertices while performing the first DFS

(b) Suppose it was found that the mayor's claim was wrong. She has now made a weaker claim: "if you start driving from town-hall, navigating one-way streets, then no matter where you reach, there is always a way to drive legally back to the town-hall. Formulate this weaker property as a graph theoretic problem and explain how it can be solved in linear time.

Ans:

(a) The formulation to the graph problem would be the same as part (a) for the this question

(b) First, in this we will use DFS to find all the vertices that are reachable from town hall(TH) and mark all the vertices in a list of vertices named *reachable*.

$DFS(G, TH, \&reachable)$

Then, we will flip all the edges in the graph G

Then, we will use DFS again to find all the vertices that are reachable from town hall(TH) and mark all the vertices in a list of vertices named *reach_back*.

$DFS(G, TH, \&reach\_back)$

Finally, we will check if all the vertices reachable from TH can also go back to TH by checking if $reachable[i] = \textbf{true}$ and $reach\_back[i] = \textbf{true}$ for all $i \in V(G)$

(c) This algorithm in the first part check all the vertices that are reachable from TH, i.e., it checks all the intersections that are reachable from town hall

And then after flipping the edges, it checks all the vertices that has atleast one path to reach TH, i.e., all the intersections that can reach the town hall legally.

And then finally check if all the vertices in both the part are same, i.e., all the intersections that are reachable from town hall can also reach back to the town hall legally.

(d) **Run Time**

Time Complexity of the algorithm is $O(V+E)$ (linear) as this algorithm is DFS based. It does DFS two times on the graph, and also changes the the direction of all the edges that would require $O(E)$ time, and DFS takes $O(V+E)$ time, so the total time complexity is $O(V+E)$.

Space Complexity of the algorithm is $O(V)$ as we use a use two arrays of size V to store the vertices visited in both of the DFS runs

Algorithm 1 DFS algorithm

```
Require: $G = (V, E)$, $TH$ is the town hall vertex
Ensure: $reachable$ is the list of all the vertices and intially all the vertices are marked false
Ensure: $G$ is given in adjacency list form
  function DFS($G, root, \&reachable$) then
    # $\&reachable$ means that the reference to reachable ios sent like in the C++
    $reachable[root] \leftarrow$ true
    for $v \in Adj(root)$ do
      if $reachable[v] = true$ then
        continue
      end if
      $DFS(G, v, \&reachable)$
    end for
  end function
```

2. Given an edge-weighted connected undirected graph G = (V, E) with n + 20 edges. Design an algorithm that runs in O(n)-time and outputs an edge with smallest weight contained in a cycle of G. You must give a justification why your algorithm works correctly.

Ans:

(a) Formulation into graph is not needed as it is already given in the question.

The graph G is assumed to be in adjacency list form and a function $w[v_1 \rightarrow v_2]$ is assumed to give the weight of the edge form v_1 to v_2

(b) In this question we will use a modified version of DFS function and also maintain a dictionary, i.e., *visited* in the global scope that stores the vertex as key and the steps taken by the DFS algorithm to reach that vertex, there is also a gobla variable *minEdge* that stores the wait of the minimum edge which would be the answer to the question, which would initially be ∞. Also a global variable *currSteps* to keep the track of steps taken(nodes reached) by the DFS algorithm, with initial value as 0.

(c) In this we will run DFS and check if a cycle is formed, i.e., if the vertex is already visited is visited again, we will back track to that vertex that was visited again and check what is the minimum edge wait in that, and while back-tracking we find another cycle we will use the longer cycle while back tracking. The best way to explain this in my opinion is dry running my algorithm with an example.

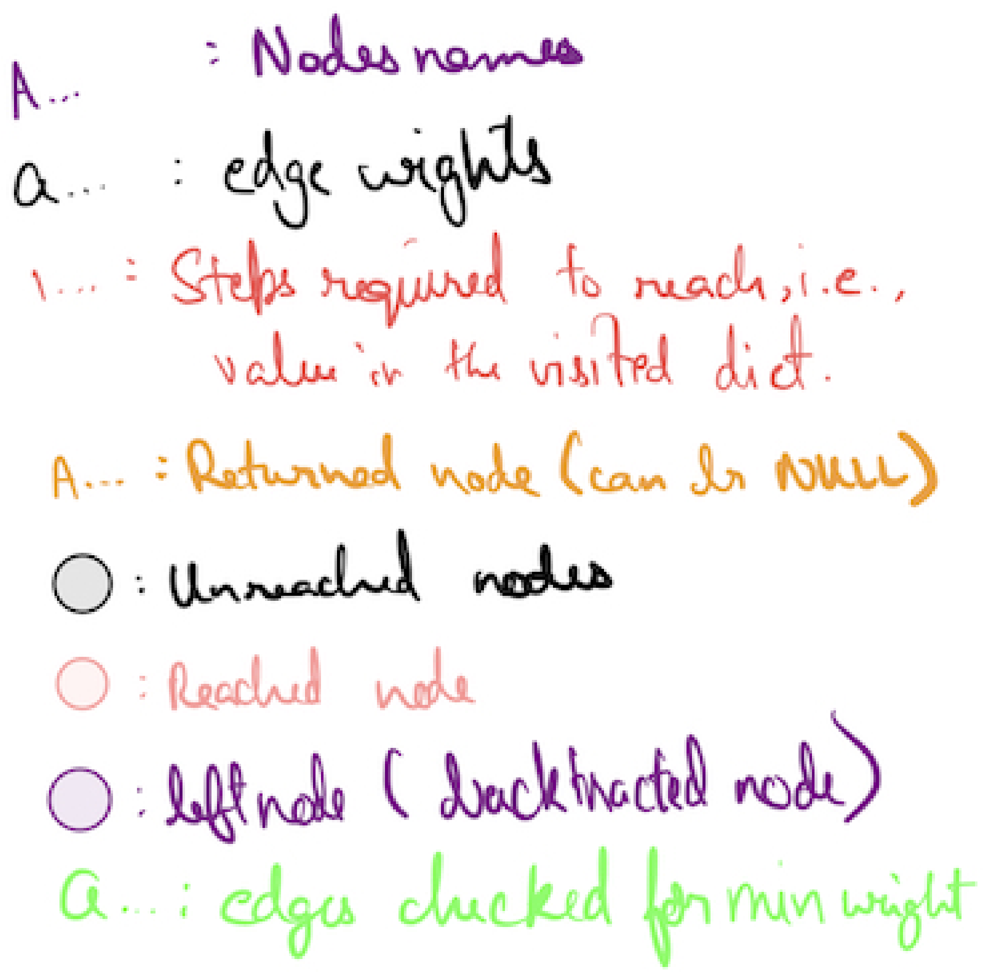

A... : Nodes names
a... : edge weights
1... : Steps required to reach, i.e., value in the visited dict.
A... : Returned node (can be NULL)
: Unreached nodes
: Reached node
: left node (backtracked node)
a... : edges checked for min weight

Figure 1: Colour Codes

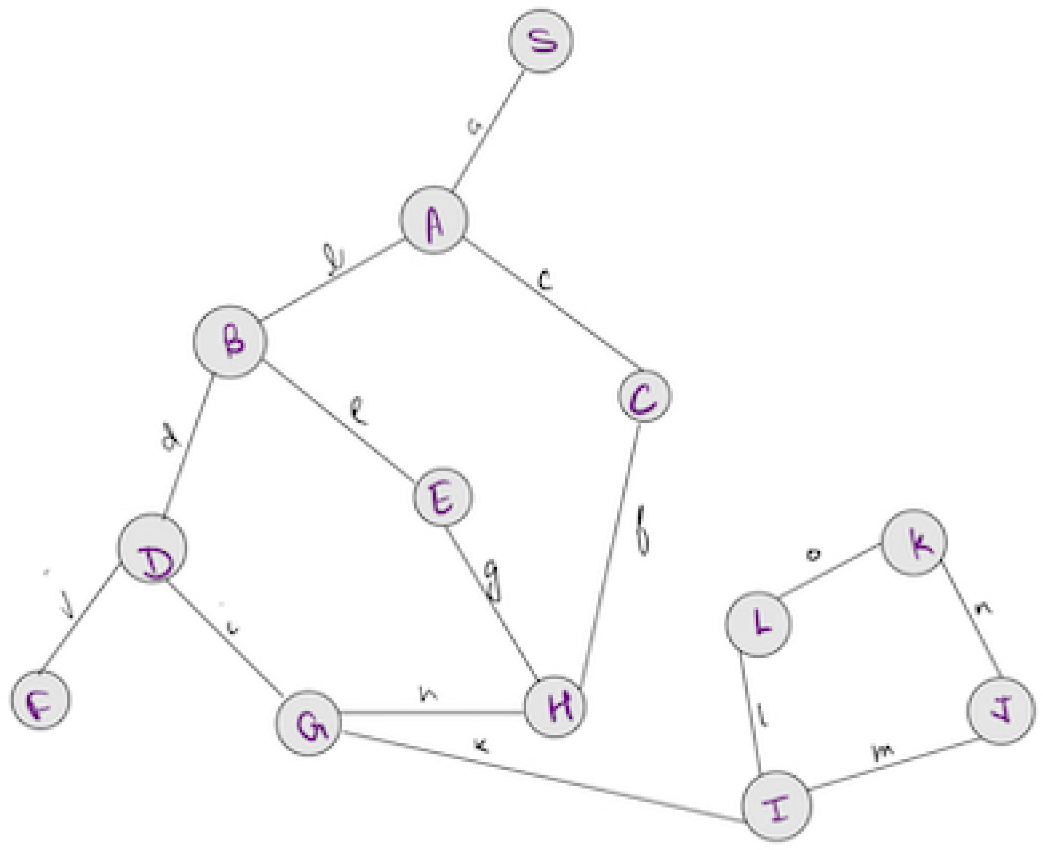


Figure 2: Steps 1

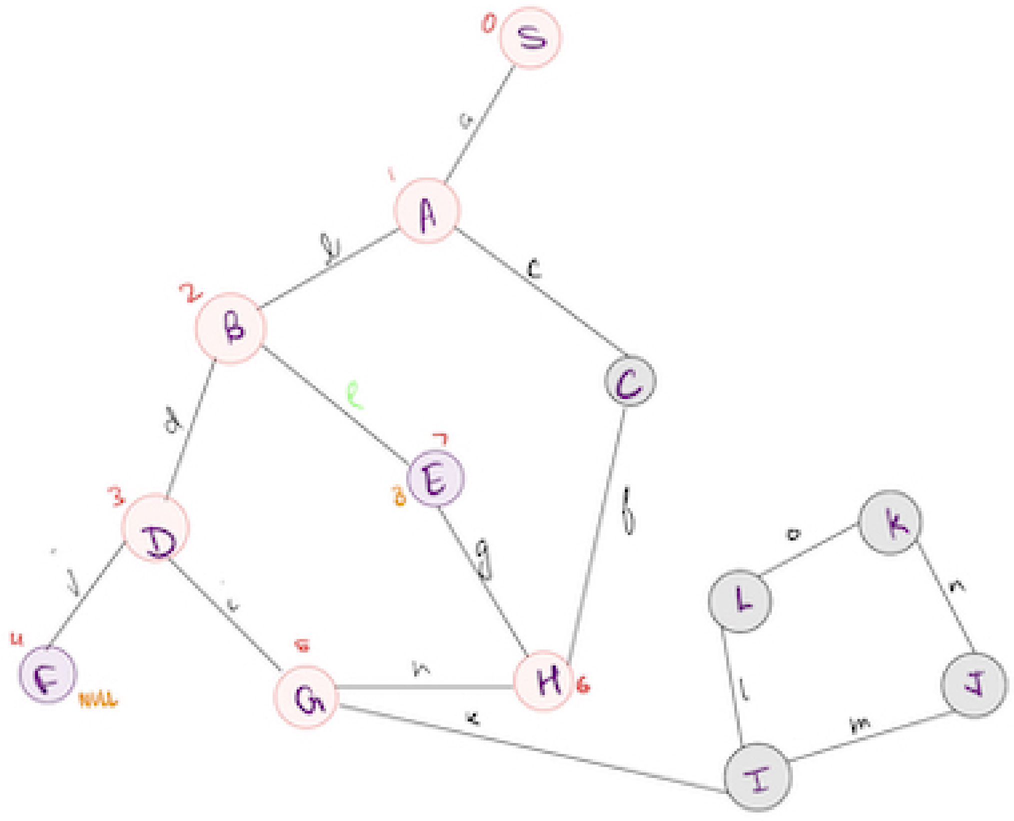


Figure 3: Step 2

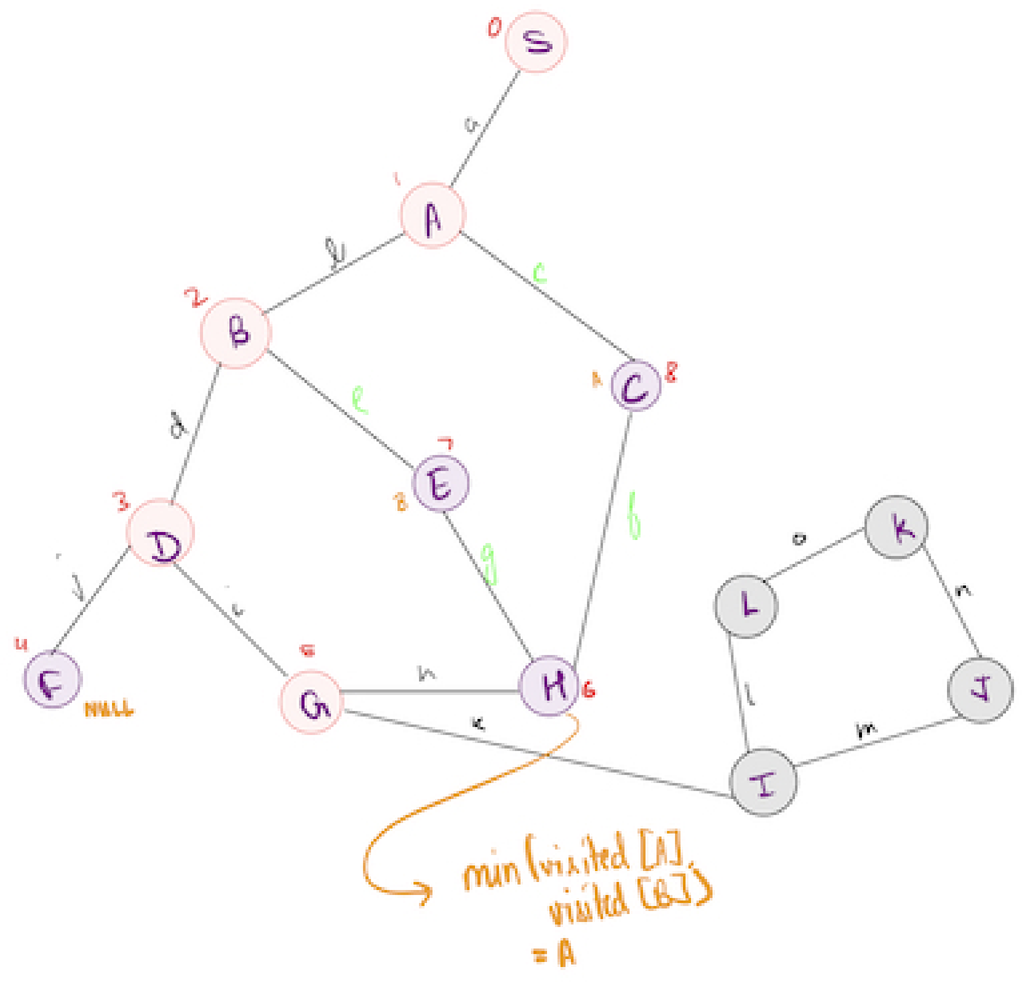


Figure 4: Step 3

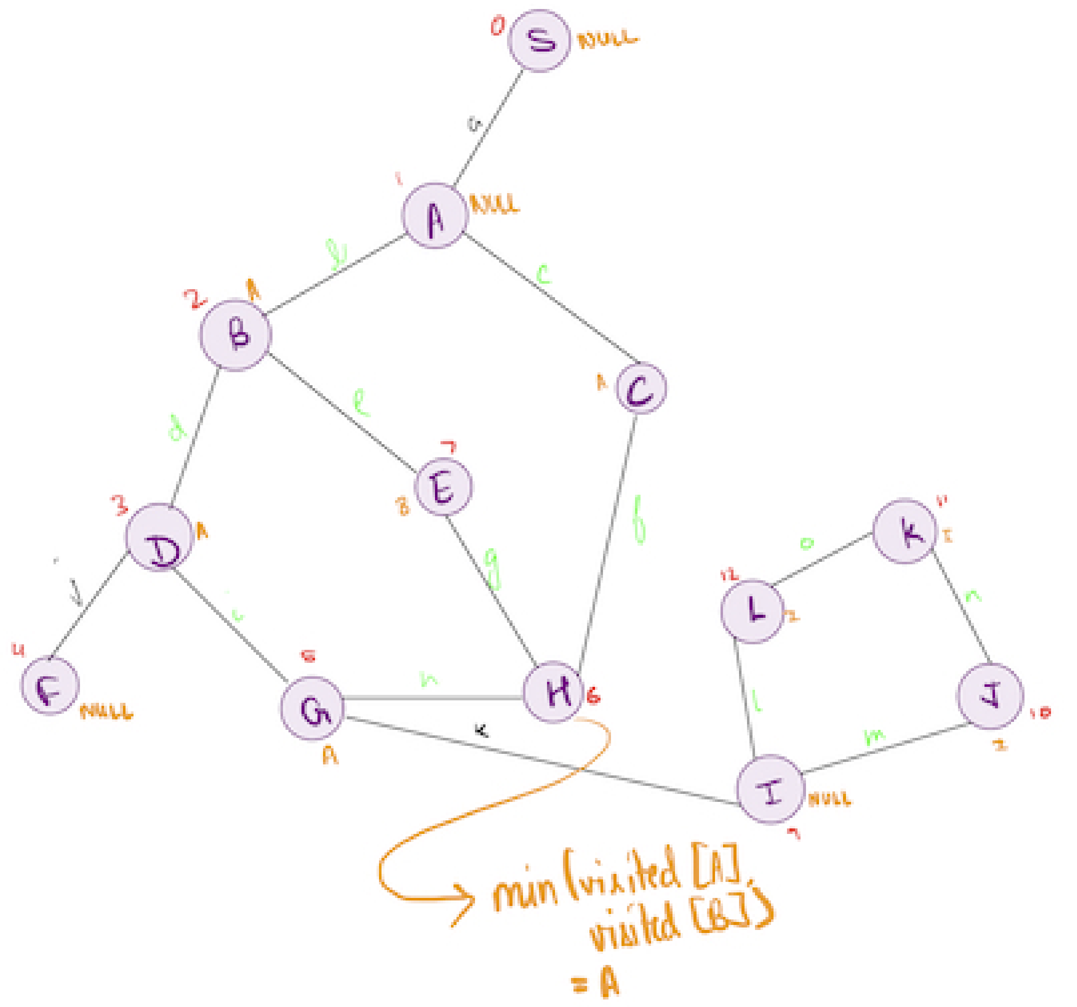


Figure 5: Step 4

(d) **Run Time**

Time Complexity of this algorithm is $O(V+E)$ as we are using only $O(1)$ functions to modify the DFS algorithm, therefore the time complexity of the algorithm will be the same as DFS, i.e., $O(V+E)$

Space Complexity of this algorithm is $O(V)$ as we are making a dictionary, i.e., *visited* of max size V, and rest of the other variables only take $O(1)$ space so the overall complexity is $O(V)$

Algorithm 2 Modified version of DFS as mentioned above

```
Require: Graph $G(V, E)$
Ensure: $G$ is in the adjacency list form
  function DFS($G, root$) then
    $visited[root] \leftarrow currSteps$
    $currSteps \leftarrow currSteps + 1$
    $retNode \leftarrow$ NULL
    for $v \in Adj(root)$ do
      if $reached[v] \neq \infty$ then
          if $retNode =$ NULL or $visited[retNode] > visited[v]$ then
              $retuNode \leftarrow v$
          end if
          $minEdge \leftarrow min(minEdge, w[root \to v])$
          continue
      end if
      $n \leftarrow DFS(G, v, \&reachable)$
      if $n \neq$ NULL and ($retNode =$ NULL or $visited[retNode] > visited[n]$) then
          $retNode \leftarrow n$
          if $n \neq$ NULL then
              $minEdge \leftarrow min(minEdge, w[root \to v])$
          end if
      end if
    end for
    if $retNode = root$ then
      return NULL
    end if
    return $retNode$
  end function
```

3. Suppose that G be a directed acyclic graph with following features.

- G has a single source s and several sinks t1, . . . , tk
- Each edge (v → w) (i.e. an edge directed from v to w) has an associated weight p(v → w) between 0 and 1.
- For each non-sink vertex v, the total weight of all the edges leaving v is

$$\sum_{(v\to w)\in E} Pr(v \to w) = 1$$

The weights Pr(v → w) define a random walk in G from the source s to some sink ti; after reaching any non-sink vertex v, the walk follows the edge v → w with probability Pr(v → w). All the probabilities are mutually independent. Describe and analyze an algorithm to compute the probability that this random walk reaches sink ti for every i ∈ {1, . . . , k}. You can assume that an arithmetic operation takes O(1)-time.

Ans:

0.1 DP part explaination

(a) In the **preprocessing step**, we create a 2D matrix *dp* of size $V \times t$.

(b) The **subproblem** is the probability of reaching the sink t_j from v_i. The entries of the dp matrix, dp[i][j], represent this.

(c)
$$dp[node][j] = weight(node- > adjacentnode) \times dp[adjacentnopde][j]$$

(d)
$$dp[s][t_1], ..., dp[s][t_n] is\ the\ solution\ to\ the\ problem\ where\ s\ is\ the\ start\ vertex$$

(e) Algorithm

(f) **Runtime**

Time Complexity: $O(V + E)$, where V is the number of vertices and E is the number of edges. We are using DP for ease of computation but we are following the basic DFS algorithm, hence the time complexity.

Space Complexity: $O(V * t)$, since only a new array *dp* with $V \times t$ elements is created

0.2 Graph part explaination

(a) A directed acyclic graph is given where s is the start vertex and the sinks are the end vertices.

(b) DFS on a directed graph is being used to find all paths from a source to end vertex and $O(1)$ computations are being used to calculate the probabilities.

(c) DFS, a basic graph traversal algorithm is sufficient to traverse through all possible paths and we are simply multiplying the probabilities of traversing through a path and optimising the process.

Algorithm 3

```
Require: s is the source vertex, t's are the sinks, edge weights (here, probabilities) between
  all vertices
Ensure: For each non-sink vertex, the total weights of all edges leaving it is 1.
  for i in range 0 to v do
    for j in range 0 to k do
      dp[i][j]=-1
    end for
  end for

  function dfs($node$) then
    if node is sink then
      probability[*]=0
      probability[node]=1
      return probability
    end if
    if $dp[node][t_1] \neq -1$ then
      return dp[node]
    end if
    for j in range 0 to t do
      probability[j]=0
    end for
    for $v \in Adj(node)$ do
      arr=dfs(v)
      for i in range 0 to k do
          $probability[i] += weight(node \rightarrow v) * arr[i]$
      end for
    end for
    dp[node]=probability
    return probability
  end function
```