Theory Assignment-2

Roll No: 2021080 **Section: A**

Name: Pranjal Bharti Specialization: CSE

1. Answer to Question 1:

Given:

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) We have to formulate this as a graph theoretic problem and show that it can be solved in linear time.

Formulation of graph theoretic problem:

We can relate the given problem to a graph theoretic problem by creating a directed graph with each vertex being the same as an intersection in the city. Since all the streets in the city are one-way streets, we will have to direct these edges accordingly. For example, if we have two intersections(vertices) A and B such that the one-way street goes from A to B, the corresponding edge representing the street (AB) will also be directed from A to B. Once we have created a directed graph according to the directions of the streets, we can now show that it is possible to legally drive from one intersection to another in linear time.

Algorithm:

In order to check whether it is possible to drive from one intersection to any other intersection in the city, we will use the concept of strongly connected components. Therefore, the problem can be rephrased in graph theoretic terms as checking whether the directed graph we created consists of one strongly connected component. If the directed graph has exactly one strongly connected component, it means that it is possible to go from one vertex to any other.

The algorithm is as follows:

1. Perform a depth-first search (DFS) starting from any vertex in the graph. Keep track of the visited vertices during the search.
2. If there are any unvisited vertices after the DFS, return false because there is at least one strongly connected component that is not reachable from the starting vertex.
3. Reverse the directions of all edges in the graph to obtain the transpose graph.
4. Perform another DFS starting from the same vertex as in step 1, but this time on the transpose graph. Keep track of the visited vertices during the search. 5. If there are any unvisited vertices after the second DFS, return false because there is at least one strongly connected component that is not reachable from the starting vertex in the transpose graph.
6. If all vertices are visited after both DFSs, return true because the graph

consists of exactly one strongly connected component.

Pseudocode:

```
function is_one_scc(graph):
 visited ← set()

 // 1: Perform a DFS on the graph
 def dfs(vertex):
 visited.add(vertex)
 for neighbor in graph[vertex] do:
 if neighbor not in visited then:
 dfs(neighbor)
     end if
     end for
 dfs(next(iter(graph)))

 // Step 2: Check if all vertices are visited
 if len(visited) != len(graph) then:
 return False

 // Step 3: Reverse the directions of edges
 transpose = {v: set() for v in graph}
 for vertex in graph do:
 for neighbor in graph[vertex] do:
 transpose[neighbor].add(vertex)
     end for
 end for
 // Step 4: Perform another DFS on the transpose graph
 visited = set()
 dfs(next(iter(transpose)))

 // Step 5: Check if all vertices are visited
 if len(visited) != len(transpose) then:
 return False

 // Step 6: Return true if the graph consists of exactly one strongly connected
component
 return True
```

Correctness:

The algorithm starts by performing a depth-first search (DFS) on the graph, starting from an arbitrary vertex. This allows the algorithm to explore all vertices that are reachable from the starting vertex. If there are any unvisited vertices after the DFS, this means that there is at least one strongly connected component that is not reachable from the starting vertex. This is because if a vertex is not visited during the DFS, it means that there is no path from the starting vertex to that vertex.

Next, the algorithm reverses the directions of all edges in the graph to obtain the transpose graph. This is necessary because the transpose graph represents the same set of strongly connected components as the original graph, but in reverse order. The algorithm then performs another DFS on the transpose graph, starting from the same

vertex as in the first DFS. This DFS allows the algorithm to explore all vertices that can reach the starting vertex in the transpose graph. If there are any unvisited vertices after this DFS, it means that there is at least one strongly connected component that cannot reach the starting vertex in the transpose graph. This is because if a vertex is not visited during this DFS, it means that there is no path from that vertex to the starting vertex in the transpose graph, and hence there is no path from the starting vertex to that vertex in the original graph.

If all vertices are visited after both DFSs, the algorithm returns true, indicating that the graph consists of exactly one strongly connected component. This is because if all vertices are visited, it means that there is a path from every vertex to every other vertex in the graph, and hence there is only one SCC in the graph. Therefore, this algorithm correctly checks if a directed graph consists of exactly one strongly connected component.

Time Complexity:

The algorithm performs two DFSs on the graph and its transpose, and each DFS takes O(V + E) time. Therefore, the total time complexity of the algorithm is O(2V + 2E), which simplifies to O(V + E).

(The time complexity of DFS has already been discussed in class)

(b) Formulation of graph theoretic problem:

The formulation of the graph theoretic problem remains more or less the same. We create the directed graph in a similar manner. However, this time we have to check whether the townhall is a node in a sink Strongly connected component.

Algorithm & Pseudocode:

We have to check whether the townhall is a node in a sink Strongly connected component. If it is, then all the nodes that are reachable from the townhall are in the same strongly connected component and therefore, there will always be a way back from these nodes to the townhall. If not, it is possible to reach a vertex from the townhall that is in a different SCC but it won't be possible to reach the townhall from that vertex.

The algorithm for the same is as follows:

- Run the algorithm to find all the strongly connected components of the graph.
- Keep a check of all the nodes that are present in the same SCC as the townhall.
- Now we run a DFS with the starting node as the townhall until it gets stuck.
- Check if all the nodes reachable from townhall are in the same SCC as the townhall.
- If so, the mayor is right. If not, the opposition is right.

Time complexity:

The above algorithm is basically a graph traversal algorithm using the Depth First-Search Algorithm. Since DFS is linear, the above algorithm also takes linear time for execution.

Correctness:

To see why this algorithm is correct, note that a sink node is a node that has no outgoing edges, i.e., it can only be reached from other nodes but cannot reach any

other node. Therefore, if the DFS starting from the given node reaches a sink node, then the given node is in a sink strongly connected component because there is a path from the given node to the sink node, and there is no path from the sink node to any other node.

On the other hand, if the DFS does not reach a sink node, it means that the given node can reach some other nodes, and those nodes can reach other nodes, and so on. This implies that there is a path from the given node to some other node, and there is a path from that node to some other node, and so on. Therefore, the given node is not in a sink strongly connected component.

2. Answer to Question 2:

Given: An edge weighted connected undirected graph G = (V, E)

Number of edged = n + 20

We must design an algorithm that runs in O(n) time (or better) and outputs an edge with smallest weight contained in a cycle of G.

Since we are already given a graph G = (V, E) with n + 20 edges, we do not have to formulate it as a graph theoretic problem.

Algorithm:

In order to solve this problem, we will use a modified version of the Kruskal's algorithm (discussed in class) for finding the minimum spanning tree of a graph. The main idea is to keep track of the edges that are not added to the minimum spanning tree and check if any of them form a cycle. If an edge forms a cycle, we can check if it has the smallest weight among all the edges in the cycle.

The reason we use Kruskal's algorithm as a basis for our modified algorithm is that it helps us find cycles in the graph efficiently. Specifically, Kruskal's algorithm maintains a set of edges S that form a forest, and each time it considers an edge (u, v) it checks whether u and v are already connected in the forest. If they are not, it adds (u, v) to the forest. If they are, it discards (u, v) because adding it would create a cycle.

By modifying this check slightly, we can use Kruskal's algorithm to find cycles in the graph efficiently. Specifically, instead of discarding (u, v) when it forms a cycle with the edges in S, we can examine the cycle and return the edge with the smallest weight in the cycle.

The algorithm is as follows:

1. Sort the edges in non-decreasing order of weight.
2. Initialize an empty set S of edges.
3. For each edge (u, v) in the sorted list of edges: a. If adding (u, v) to S does not create a cycle, add it to S. b. If adding (u, v) to S creates a cycle, check if (u, v) has the smallest weight among all the edges in the cycle. If it does, output (u, v) and terminate the algorithm. If it does not, continue to the next edge.

4. If no edge has been outputted, we will return "No cycles encountered" as the

output.

Pseudocode:

Pre-processing step-Sort the edges in non-decreasing order
Initialize an empty set S of edges
for *each edge e* ∈ *E(G)* **do:**
if *e does not create cycle* **then:**
S ← *S* ∪ *{e}*
end
else:
smallest_edge ← *find_smallest_edge(cycle)*
Return smallest_edge
end
Return "No cycles encountered"

In order to check if a cycle is created on adding a particular edge to S, we use Union-Find Data Structure (as discussed in Lecture-13), thus making our code more efficient. In case a cycle is formed, we are using an auxiliary method *find_smallest_edge(cycle)* which simply finds out the smallest-weighted edge in the cycle encountered.

Correctness:

To justify the correctness of the algorithm, we need to show that it outputs an edge with the smallest weight contained in a cycle of G.

First, note that the algorithm outputs an edge if and only if it forms a cycle with the edges already added to S. This is because Kruskal's algorithm adds edges in non-decreasing order of weight, so if an edge is not added to S, it means that there is already a path between its endpoints in S.

Second, suppose that the algorithm outputs an edge (u, v). This means that (u, v) forms a cycle with the edges already added to S, and there is no other cycle that contains (u, v) with smaller total weight. To see why this is true, suppose there is another cycle C that contains (u, v) with smaller total weight. Since Kruskal's algorithm adds edges in non-decreasing order of weight, there must be an edge e in C that was considered before (u, v) and not added to S. But adding e to S would create a cycle that contains (u, v) with smaller total weight than C, which contradicts the assumption that C has the smallest total weight among all cycles that contain (u, v).

Therefore, the algorithm outputs an edge with the smallest weight contained in a cycle of G.

Time complexity:

As discussed in Lecture-13, the time complexity of Kruskal's algorithm is *O(ElogE)* when we are using the Union-Find Data Structure. It is given to us that the number of edges (E) is equal to n + 20 (where n is the number of vertices). Thus, we can simplify the running time of the algorithm as follows.

1. Sorting the edges takes O(E log E) time, which is at most O((n + 20) log (n + 20)).
2. The for loop runs E times, which is at most n + 20 times.
3. Checking if adding an edge creates a cycle takes O(log V) time using union find data structure. Since the graph is connected, V = O(E), so this takes O(log E)

time.

4. Checking if an edge has the smallest weight among all the edges in a cycle takes constant time.

Therefore, the total time complexity of the algorithm is O((n + 20) log (n + 20)) + O((n + 20) log E) + O(constant) = *O(n log n)*.
Subsequently, we now have an algorithm that is logically correct and takes O(n) time (tightest asymptotic notation will be *O(n log n)* and outputs the smallest-weighted edge in a cycle present in a simple connected undirected graph G.

3. Answer to Question 3:

Given: A directed acyclic graph G
We have to find an algorithm that computes the probability that this random walk reaches sink ti for every i ∈ {1, . . . , k}.
Since we are already given a graph, we don't have to formulate a graph theoretic problem.

Algorithm:

The algorithm which we will use to solve this problem basically has two parts. First, we apply topological sorting on the given DAG, which sorts the vertices in a topological order. Once we are done with this, we calculate the probability of reaching each vertex. Both these algorithms are finally used in the main algorithm to give us the desired output.

Algorithm 1: Topological sorting of DAG

This part of the algorithm sorts the vertices in the given DAG in a topological order, which means that for any edge from vertex u to vertex v, u comes before v in the sorted list. This is done by performing a depth-first search (DFS) on the vertices of the DAG in a random order, and adding the visited vertices to a list in reverse order. The DFS function visits each neighbor of the current node if it has not been visited yet, and removes the current node from the set of unvisited vertices. This ensures that all vertices are visited exactly once, and the sorted list is obtained in O(n + m) time complexity, where n is the number of vertices and m is the number of edges in the DAG.

Algorithm 2: Calculate the probability of reaching each vertex

This part of the algorithm uses the sorted list of vertices obtained from Algorithm 1 to calculate the probability of reaching each vertex in the DAG starting from the source vertex s. The algorithm initializes the probability of reaching each vertex to 0, except for the source vertex which is initialized to 1. Then, it iterates over the vertices in the sorted list, skipping the sink vertices (which have a probability of 0). For each vertex v, it calculates the probability of reaching its neighbors w by adding the product of the probability of reaching v and the weight of the edge from v to w to the probability of reaching w. This process continues until all vertices have been processed. The final probability values for each vertex are returned as a dictionary with vertex keys.

Note: Algorithm 2 itself uses dynamic programming
It does not have a traditional base case like other recursive functions but,

it assumes that the probability of reaching the source vertex is 1. Therefore, this can be the base case for Algorithm 2.

As such there is no preprocessing step for Algorithm 2 however, we should note that the input format of the DAG for optimal time complexity is adjacency list or matrix. Apart from this, it takes a set of sink vertices (t) and the source vertex s as the input.

Subproblem for Algorithm 2:

Let P[v] denote the probability of reaching vertex v from the source vertex s. The subproblem is to compute P[v] for all vertices v in the graph, using the probabilities of reaching vertices that are reachable from v.

Main algorithm: Calculate probability of reaching sink vertices

This part of the algorithm simply calls Algorithm 2 to calculate the probability of reaching each sink vertex in the DAG starting from the source vertex s, and prints the resulting probabilities. The algorithm takes as input the DAG G, source vertex s, and sink vertices t (which could be a single vertex or a set of vertices).

Pseudocode:

```
// Algorithm 1: Topological sorting of DAG
// Sorts the given directed acyclic graph in topological order
function topologicalSort(G):
        L <- [] // Empty list. It will contain the vertices in topologically sorted
    order
        S <- {} // Set of all vertices
        while len(S) do> 0: //Performing DFS on all the vertices
            node <- random node from S
            dfs(node, G, S, L)
        end while
    return L
end function

// This is a helper function for the topologicalSort function
// It performs DFS and adds the nodes to the sorted list (L) in reverse order
function dfs(node, G, S, L):
    //Remove the vertex from the set to mark it as visited
    S.remove(node)
    for neighbor in G[node] do:
        // Performing DFS on the neighbours of the node if it is in S i.e.
    unvisited
        if neighbor in S then:
            dfs(neighbor, G, S, L)
        end if
    end for
    // Inserting the vertex in topological order
    L.insert(0, node)
end function
```

```
// Algorithm 2: Calculate the probability of reaching each vertex
// Calculates the probability of reaching each vertex using the given directed acyclic graph, source vertex and sink vertices
function calculateProbabilities(G, s, t):
    // Initializing the probability as 0
    P ←{v: 0 for v in G.keys()}
    // Initializing the probability of source vertex as 1
    P[s] ← 1
    // Iterating over the vertices in topological order and calculating their probabilities
    for v in topologicalSort(G) do:
        if v in t then:
            continue
        end if
        for w in G[v] do:
            P[w] ←P[w] + P[v] * G[v][w]
            // Initial value of P[w] has been already set to 0
        end for
    end for
    return P
end function


// Main algorithm: Calculate probability of reaching sink vertices  // Calculates the probability of reaching sink vertices using the given directed acyclic  graph, source vertex and sink vertices, and prints the result
def calculateSinkProbabilities(G, s, t):
    P = calculateProbabilities(G, s, t) //t includes all the sinks, s is the source vertex
    // Printing the probabilities for each sink
    for ti in t do:
    print(P[ti])
    end for
end function
```

Correctness:

Algorithm 1 performs a topological sort of the DAG, which guarantees that the order of the vertices satisfies the topological sorting property. This ensures that Algorithm 2 can apply the recursive formula for probabilities in the correct order, i.e., starting from the source vertex and proceeding in a topological order.

The probability of reaching a vertex v in a DAG can be expressed as the sum of the probabilities of reaching its predecessors multiplied by the probability of the edge connecting them. This formula can be recursively applied to all the vertices in the DAG in a topological order, starting from the source vertex.

Algorithm 2 calculates the probabilities of reaching each vertex by iterating over the

vertices in topological order and applying the recursive formula. Since the vertices are processed in a topological order, the probabilities of reaching the predecessors of a vertex have already been calculated, and the probability of reaching the vertex can be calculated using these values.
Once we have computed the probability of reaching each vertex in the graph using Algorithm 2, we can easily compute the probability of reaching any subset of vertices, including the sink vertices, by summing up the probabilities of reaching each vertex in the subset.
Thus, the algorithm is correct because it correctly applies the topological sorting property and the recursive formula for probabilities to calculate the probabilities of reaching the sink vertices in a DAG.

Time Complexity:

Assume: - Number of vertices: n

Number of edges: m

The resulting time complexity of this algorithm is *O(n + m).*

The time complexity of Algorithm 1 (Topological Sorting of DAG) is *O(n + m).* (already discussed in class)

Algorithm 2 also has a time complexity of *O(n + m),* as it iterates over the vertices in topological order and calculates the probability of reaching each neighbor. The inner loop of the algorithm visits all the neighbors of the current vertex, which gives a time complexity of *O(m).* Since each vertex and edge is processed exactly once, the total time complexity of Algorithm 2 is also *O(n + m)*

Therefore, the overall time complexity of this entire algorithm is ***O(n+m).***