Theory Assignment-2
Section:B

Roll No:2021482	**Roll No:2021304**
Name:PARAS DHIMAN	**Name:**ADITYA DAIPURIA
Specialization:CSB	**Specialization:**CSD

Ques1:

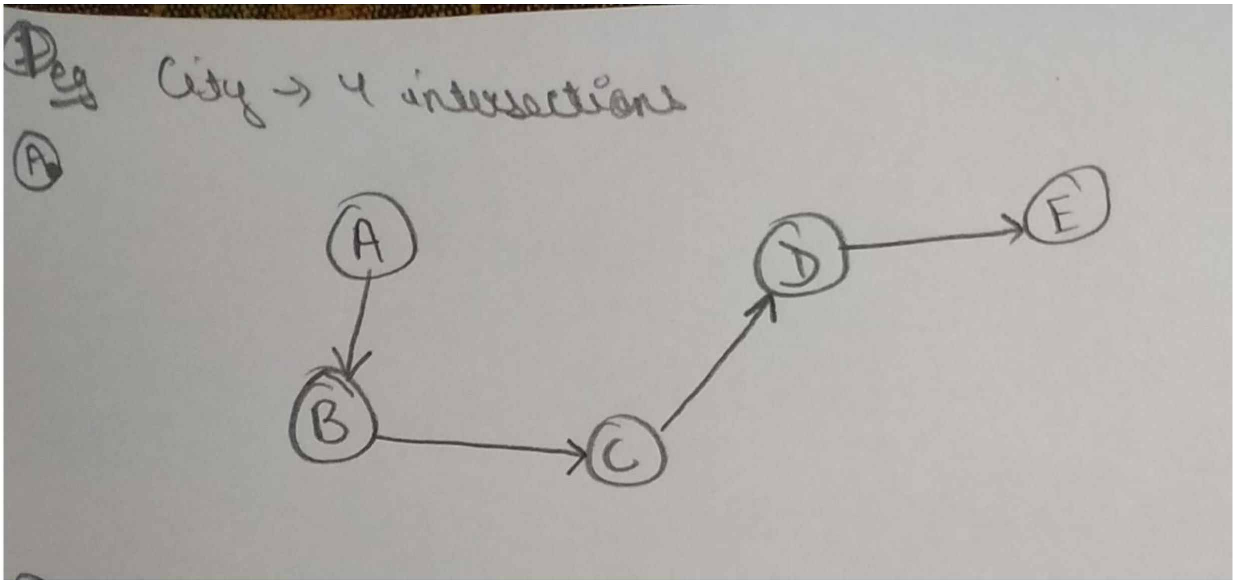
A.

We can represent it using a directed graph G (V,E), where each intersection will be a vertex v and an edge between intersections (u,v) when there is a direct road from u to v. The condition that there is a driveway from any intersection to any intersection is valid if there exists a path from each vertex in G to every other vertex. This will be true if the graph is strongly connected. The problem can be solved in linear time with Depth First Search (DFS).

(a) Formulating the problem as a graph theoretic problem:

To formulate the problem of driving legally from one intersection to any other intersection on one-way streets in the city of computopia as a graph theoretic problem, we can represent the city's road network as a directed graph where each intersection is a vertex and each one-way street is a directed edge. The direction of the edge corresponds to the direction of the one-way street. For instance, if there is a one-way street from intersection A to intersection B, we can represent this as a directed edge from vertex A to vertex B in the graph. We can then use graph traversal algorithms to determine whether it is possible to drive legally from one intersection to any other intersection in the city.

An example of constructing the graph is as follows: Suppose the city has four intersections, A, B, C, and D. There are one-way streets from A to B, B to C, and C to D. We can represent this as a directed graph with four vertices, A, B, C, and D, and three directed edges: (A, B), (B, C), and (C, D).



(b) Explanation of the traversal Algorithms:

One way to solve the problem of legal driving from one intersection to any other intersection in Computopia is to formulate it as a graph theoretic problem and perform a DFS (Depth First Search) traversal of the directed graph.

To formulate this problem as a graph, we can represent each intersection as a vertex, and each one-way street as a directed edge between two vertices. For example, if there is a one-way street from intersection A to intersection B, then we would represent this as a directed edge from vertex A to vertex B in the graph.

To solve this problem, we can start a DFS traversal from any intersection and mark each visited vertex as visited to avoid revisiting the same intersection. During the traversal, if we can reach all other intersections in the city, we can conclude that it is possible to drive legally from one intersection to any other intersection.

In the worst-case scenario, we need to visit all the vertices and edges of the graph, which will take linear time.

The DFS traversal algorithm starts by selecting a vertex to start the search, marking it as visited, and then visiting all of its adjacent vertices recursively. When all of its adjacent vertices have been visited, the search backtracks to the previous vertex, selects an unvisited vertex adjacent to it, and repeats the process. This continues until all vertices have been visited, or until the target vertex is found. In the context of the problem of legal driving in Computopia, DFS traversal ensures that all reachable vertices are visited in a linear time complexity, and thus the algorithm can be solved in linear time.

Algorithm:

1.) Perform a DFS starting from the town-hall vertex, marking all the reachable vertices.

2.) Perform another DFS starting from each of the reachable vertices, marking all the vertices that can reach the town-hall vertex.

3.) If all the vertices are marked in step 2, then the weaker claim holds. Otherwise, it does not hold.

Pseudo Code for it:

Graph : Represented by a new structure conataining the vertices and the adjacent neighbours to it also by the means of adjacency lists

```
function legalDrive(graph, start):
            visited = new array that will store which vertex is visited or
which is not
      mark all vertices as unvisited
      stack = [start]
      mark start as visited //i.e added start to the visited array

      while stack is not empty:
        current = stack.pop()
        for neighbor in graph.adjacentVertices(current):
            if neighbor is not visited:
                  mark neighbor as visited
                  stack.push(neighbor)

      for vertex in graph.vertices():
        if vertex is not visited:
            return false

      return true
```

Explanation:

The function “legalDrive” takes a directed graph and a starting vertex as input, and returns true if it is possible to drive legally from any intersection to any other intersection, starting from the given vertex.

The function first marks all vertices as unvisited and creates a stack with the starting vertex. It then enters a while loop that pops vertices from the stack and marks their neighbors as visited if they haven't been visited yet. This process continues until the stack is empty, which means that we have visited all reachable vertices from the starting vertex.

After the traversal, the function checks if there are any unvisited vertices in the graph. If there are, it means that there is at least one intersection that cannot be reached from the starting vertex, so the function returns false. Otherwise, it means that we can drive legally from any intersection to any other intersection, so the function returns true.

(c) Necessary and Sufficient conditions for the justification of the algorithms:

Necessary conditions for the justification of this algorithm are as follows:

1. The graph representation must be correct and consistent with the problem statement.
2. The algorithm must traverse the graph through all the vertices and edges to cover all possible paths.
3. The visited array must be initialized correctly and updated during the traversal to avoid revisiting any vertex.
4. The algorithm must correctly identify if there exists a path from start vertex to all other vertices.

Sufficient conditions for the justification of this algorithm are as follows:

1. If all the necessary conditions are met, the algorithm will correctly identify if it is possible to legally drive from one intersection to any other intersection.
2. The algorithm will return true only if all the vertices are visited during the traversal, which guarantees that there is a path from start vertex to all other vertices.
3. If the algorithm returns false, then it is not possible to legally drive from one intersection to all other intersections.
4. Overall, the correctness of the algorithm depends on the correct graph representation and the proper implementation of the traversal algorithm.

(d) Explanation for the running time of the algorithm:

The time complexity of the algorithm is O(|V| + |E|), where |V| is the number of vertices and |E| is the number of edges in the graph.

In the algorithm, we first mark all the vertices as unvisited which takes O(|V|) time. Then, we create a stack and add the starting vertex to it, which takes constant time.

In the while loop, we pop vertices from the stack and iterate over their neighbors, which takes O(|E|) time. We mark each neighbor as visited and add it to the stack if it is not visited. Since each vertex is visited only once, the while loop runs for O(|V| + |E|) iterations.

Finally, we iterate over all the vertices and check if they have been visited or not, which takes O(|V|) time.

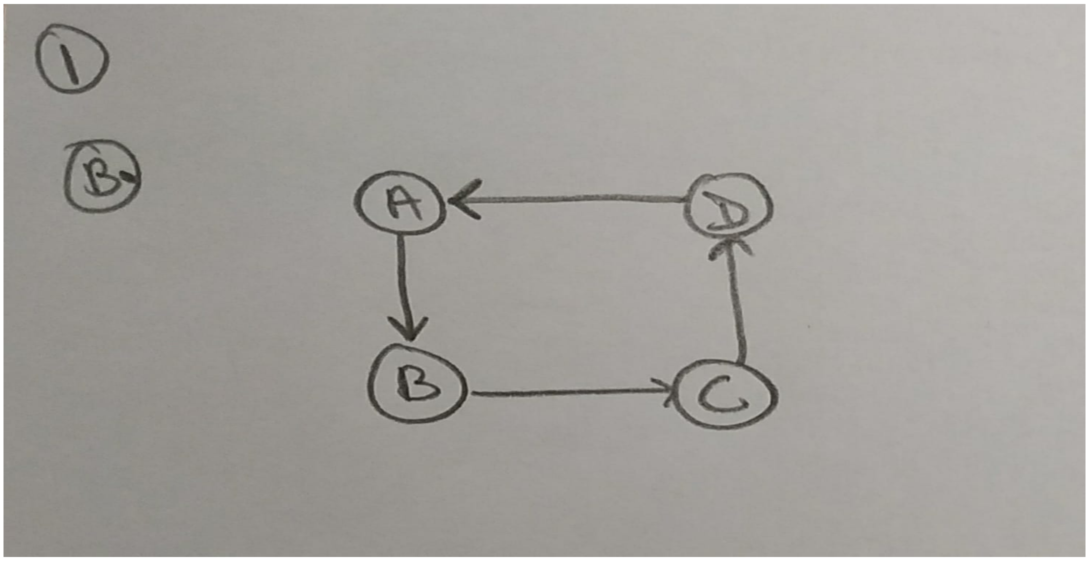
Therefore, the overall time complexity of the algorithm is O(|V| + |E|). This is a linear time complexity with respect to the number of vertices and edges in the graph, making it an efficient solution to the problem.

B.)

(a) Formulating the problem as a graph theoretic problem:

Formulating the weaker claim as a graph theoretic problem involves constructing a directed graph where the vertices represent the intersections in the city and the edges represent the one-way streets. The town-hall vertex will be the starting point of the directed graph. For each intersection vertex, we will add two directed edges: one edge pointing to the next intersection in the one-way direction, and another edge pointing to the previous intersection in the opposite direction.

Example: Suppose there are four intersections labeled A, B, C, and D. The one-way streets are as follows: A -> B, B -> C, C -> D, and D -> A.



(b) Explanation of the traversal Algorithms:

The algorithm used here is a modified version of the Breadth-First Search (BFS) algorithm. The algorithm starts at the town-hall vertex and traverses the graph using a stack-based DFS approach.

During the traversal, each visited vertex is marked as visited, and its parent is stored. When a dead-end vertex is reached, the algorithm checks if there is a path back to the town-hall vertex by following the parent vertices until it reaches the town-hall vertex. If there is no path back to the town-hall vertex, the dead-end vertex is marked as a cut vertex.

After the traversal is completed, the algorithm checks if there are any cut vertices. If there are no cut vertices, then the mayor's weaker claim is true, and the algorithm returns true. Otherwise, it returns false.

Overall, the DFS algorithm is used to traverse the graph, and the concept of parent vertices is used to check if a dead-end vertex is a cut vertex or not. The algorithm checks if there are any cut vertices by iterating through all vertices in the graph and checking if they have a parent or not.

Graph : Represented by a new structure conataining the vertices and the adjacent neighbours to it also by the means of adjacency lists

```
function legalDrive(graph, start):
  visited = new array that will store which vertex is visited or which is not
  parent = new array that will store the parent of each vertex
  mark all vertices as unvisited
  stack = [start]
  mark start as visited
  parent[start] = null
  while stack is not empty:
    current = stack.pop()
    for neighbor in graph.adjacentVertices(current):
      if neighbor is not visited:
            mark neighbor as visited
            parent[neighbor] = current
            stack.push(neighbor)

  cut_vertices = []
  for vertex in graph.vertices():
    if vertex != start and parent[vertex] is null:
      cut_vertices.append(vertex)
  if len(cut_vertices) == 0:
    return true
  else:
    return false
```

(c) Necessary and Sufficient conditions for the justification of the algorithms:

Necessary conditions for the correctness of the algorithm are:

1. The graph should be connected, meaning that there is a path between any two vertices of the graph.

2. The graph should be directed.

3. The adjacency list for each vertex should be accurate and complete, meaning that all the edges should be included in the adjacency list for each vertex.

4. The algorithm should be implemented correctly, without any logical errors.

Sufficient conditions for the correctness of the algorithm are:

1. If there are no cut vertices in the graph, then the algorithm will correctly return true, indicating that the mayor's weaker claim is true.

2. If there is at least one cut vertex in the graph, then the algorithm will correctly return false, indicating that the mayor's weaker claim is false.

Overall, the algorithm uses DFS to traverse the graph and check for cut vertices. It keeps track of the visited vertices and their parent vertices, and if a dead-end vertex is reached, it checks if there is a path back to the starting vertex. If there is no path back to the starting vertex, then the dead-end vertex is marked as a cut vertex. The algorithm returns true if there are no cut vertices, and false otherwise. The correctness of the algorithm relies on the necessary and sufficient conditions listed above

(d) Explanation for the running time of the algorithm:

The time complexity of the algorithm is O(|V|+|E|), where |V| is the number of vertices in the graph and |E| is the number of edges.

In the worst case scenario, where the graph is fully connected and every vertex is a cut vertex, the algorithm would have to visit every vertex and edge in the graph. In this case, the time complexity would be O(|V|+|E|).

The traversal of the graph using DFS takes O(|V|+|E|) time since each vertex and edge is visited once. The for loop that finds the cut vertices also takes O(|V|) time since it visits every vertex once. Therefore, the overall time complexity of the algorithm is O(|V|+|E|+|V|) = O(|V|+|E|).

This makes the algorithm linear in terms of the input size of the graph, which is a desirable property for practical applications.

Reference Links FOR 1ST a AND b:

Reference Links:

1.) **https://www.chegg.com/homework-help/questions-and-answers/police-department-city-computopia-made-streets-one-way-mayor-contends-still-way-drive-lega-q45788040**
2.) **https://blogs.asarkar.com/assets/docs/algorithms-curated/Assignment%204%20Solutions%20-%20Dalhousie%20University%20CSCI%203110.pdf**
3.) **https://quizlet.com/explanations/questions/the-police-department-in-the-city-of-computopia-has-made-all-streets-one-way-the-mayor-contends-that-there-is-still-a-way-to-drive-legally-f-16246417-04a35757-2313-4d0b-b292-ac66a9514ea3**
4.) **https://studysoup.com/tsg/759566/algorithms-1-edition-chapter-3-problem-3-15**

Ques2:

The algorithm works by performing a depth-first search (DFS) traversal of the graph starting from each vertex, while keeping track of the parent of each vertex. During the traversal, it looks for cycles in the graph by checking if a neighboring vertex is already marked as visited and is not the parent of the current vertex. If a cycle is found, it compares the weight of the current edge with the minimum weight seen so far and updates the minimum if necessary. The algorithm returns the edge with the minimum weight that is contained in a cycle.

The algorithm correctly finds the edge with the smallest weight contained in a cycle because it traverses the graph starting from each vertex and looks for cycles by checking if a neighboring vertex is already marked as visited and is not the parent of the current vertex. By doing so, the algorithm visits all the edges of the graph and finds the minimum weight edge contained in a cycle. Moreover, since the algorithm visits each vertex and each edge exactly once, it does not miss any cycle or edge with minimum weight.

Yes, it is possible to have multiple cycles in the graph that contain edges with the same minimum weight. In such cases, the algorithm may return any one of these edges as the result, since they all have the same weight and are contained in a cycle.

(a) Formulating the problem as a graph theoretic problem:

The problem can be formulated as finding the minimum weight edge in a cycle of an undirected, edge-weighted connected graph G = (V, E). To construct the graph, we use the given input of n + 20 edges, where n is the number of vertices. We can assume that the vertices are labeled from 1 to n, and the edges are represented as tuples (u, v, w), where u and v are the endpoints of the edge and w is the weight of the edge. We can represent the graph using an adjacency list or matrix, where each vertex u has a list of its neighboring vertices and their corresponding edge weights. For example, the graph G with four vertices and five edges can be represented as follows:

V = {1, 2, 3, 4}

E = {(1, 2, 5), (2, 3, 2), (3, 4, 3), (4, 1, 4), (1, 3, 6)}

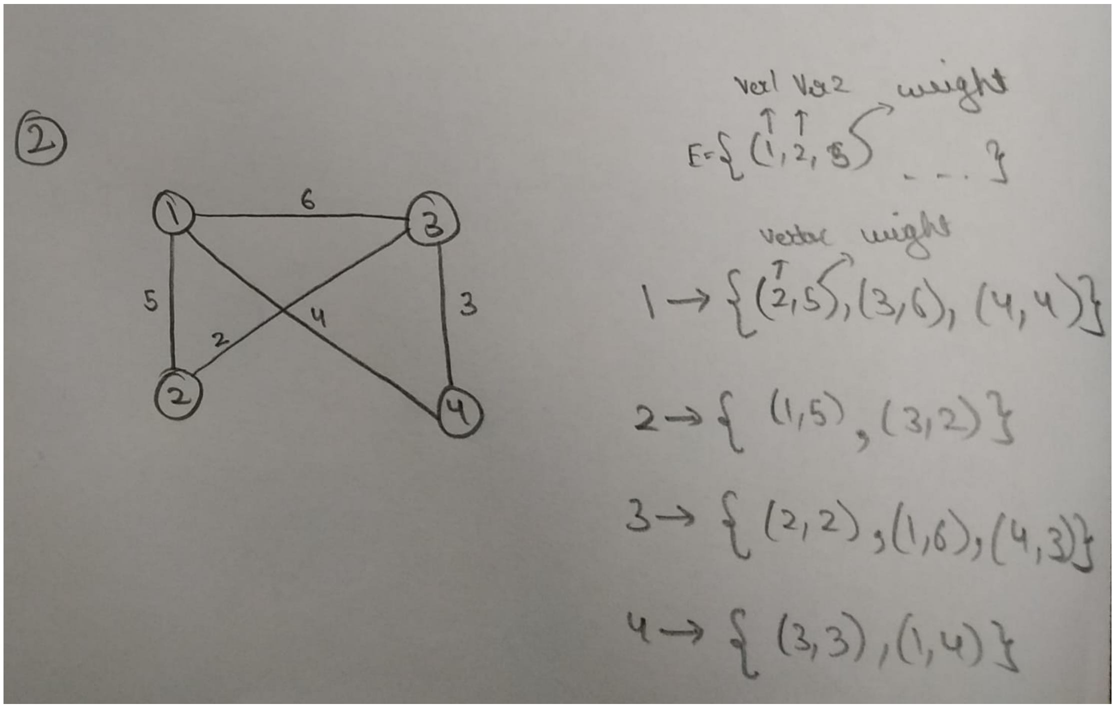
Adjacency list representation:

1: [(2, 5), (3, 6), (4, 4)]

2: [(1, 5), (3, 2)]

3: [(2, 2), (1, 6), (4, 3)]

4: [(3, 3), (1, 4)]



E={(1,2,5) . . . }

1→{(2,5),(3,6), (4,4)}

2→{ (1,5), (3,2)}

3→ { (2,2), (1,6), (4,3)}

4→ { (3,3), (1,4) }

(b) Explanation of the traversal Algorithms:

This is an algorithm to find the minimum weight edge in a graph that is part of a cycle.

At the beginning, a set called 'visited' is created to keep track of visited vertices. Initially, all vertices are marked as unvisited (i.e., false). Also, the minimum weight is initialized to infinity and the minimum weight edge to null.

The algorithm then iterates over each vertex in the graph. If the current vertex is not visited, it is marked as visited, and a parent array is initialized to keep track of parent vertices. A set called 'seen_edges' is also initialized to keep track of edges that have already been seen. The graph is then traversed starting from vertex v using DFS.

The DFS algorithm takes two arguments, the current vertex 'u' and the set 'seen_edges'. For each neighbor 'w' of 'u', if 'w' is not the parent of 'u', the algorithm checks if 'w' has been visited. If 'w' has not been visited, it is marked as visited, the parent of 'w' is set to 'u', and the edge (u,w) is added to the set 'seen_edges'. The DFS algorithm is then called recursively on vertex 'w'.

If 'w' has already been visited and the edge (u,w) is in 'seen_edges', then a cycle has been found. If the weight of the edge (u,w) is less than the minimum weight seen so far, the minimum weight and minimum weight edge are updated accordingly.

Finally, the algorithm removes the edge (u,w) from 'seen_edges' and continues iterating over all the neighbors of 'u'.

The algorithm returns the minimum weight edge after all vertices have been traversed.

The Algorithm is as follows:

```
Create a set 'visited' to keep track of visited vertices. Initially, all
elements of visited are false.

Initialize the minimum weight to infinity and the minimum weight edge to
null.

For each vertex v in V:
   If visited[v] is false:
      Set visited[v] to true.
      Initialize a parent array 'parent' of size |V| to keep track of
parent vertices. Initially, all elements of parent are null.
      Initialize a set 'seen_edges' to keep track of seen edges.
      Traverse the graph starting from vertex v using DFS(u, seen_edges).
Return the minimum weight edge.

DFS(u, seen_edges):
    For each neighbor w of u:
        If w is not the parent of u:
            If visited[w] is false:
                  Set visited[w] to true.
                  Set parent[w] to u.
                  Add the edge (u,w) to seen_edges.
                  Traverse the graph starting from vertex w using DFS.
            Else if the edge (u,w) is in seen_edges:
                  If the weight of edge (u,w) is less than the minimum
weight seen so far:
                      Set the minimum weight to the weight of edge (u,w).
                      Set the minimum weight edge to (u,w).
                  Return seen_edges
        Remove the edge (u,w) from seen_edges
```

(c) Necessary and Sufficient conditions for the justification of the algorithms:

Necessary Condition:

- Every edge in a cycle must be added to the set seen_edges.

This condition is necessary because if any edge in a cycle is not added to the set seen_edges, then the algorithm will not detect that cycle. As a result, it will not return an edge with the smallest weight contained in a cycle.

Sufficient Condition:

- The minimum weight edge returned must be contained in a cycle.

This condition is sufficient because if the minimum weight edge returned is not contained in a cycle, it means that no cycle was detected by the algorithm. Therefore, the algorithm must ensure that it returns an edge that is contained in a cycle, which guarantees that it has detected at least one cycle.

Together, these conditions ensure that the algorithm correctly detects at least one cycle and returns the edge with the smallest weight contained in that cycle.

(d) Explanation for the running time of the algorithm:

The time complexity of this algorithm is O(V + E), where V is the number of vertices in the graph and E is the number of edges in the graph. This is because the algorithm visits each vertex and edge at most once during the traversal.

First, the algorithm initializes a boolean array 'visited' of size |V|, which takes O(V) time. Then, for each vertex v in V, the algorithm checks if visited[v] is false. Since there are V vertices, this loop takes O(V) time.

For each unvisited vertex, the algorithm initializes a parent array 'parent' of size |V|, which takes O(V) time, and a set 'seen_edges' to keep track of seen edges, which takes O(1) time. Then, the algorithm traverses the graph starting from vertex v using DFS.

During the DFS traversal, the algorithm visits each neighbor of the current vertex exactly once. If the neighbor is unvisited, the algorithm sets visited[w] to true and sets parent[w] to u, which takes O(1) time. Then, it adds the edge (u,w) to seen_edges, which takes O(1) time. Finally, the algorithm recursively calls DFS on the neighbor w.

If the neighbor w is visited and the edge (u,w) is in seen_edges, the algorithm updates the minimum weight edge if necessary. This step takes O(1) time.

Since the algorithm visits each vertex and edge at most once, the time complexity of the algorithm is O(V + E).

Reference Links:

1.) https://www.geeksforgeeks.org/find-minimum-weight-cycle-undirected-graph/
2.) https://sureshvcetit.files.wordpress.com/2018/04/solu9.pdf

3.) https://www.quora.com/Let-G-V-E-be-a-connected-undirected-graph-give-an-O-V-+-E-algorithm-to-compute-a-path-in-G-that-traverses-each-edge-in-E-exactly-once-in-each-direction

4.) https://testbook.com/question-answer/let-g-be-a-weighted-connected-undirected-graph-wit--5df4d254f60d5d4cbdd33e5c

5.) https://www.chegg.com/homework-help/questions-and-answers/question-2-given-edge-weighted-connected-undirected-graph-g-v-e-n-20-edges-design-algorith-q112034724

Ques3:

a.) Pre-Processing Stage:

There is no preprocessing stage for this problem.

b.)Precise Description for the Sub-Problems:

Let P[i][j] be the probability that a random walk starting at the source s reaches the sink tj, using only vertices {v1, v2, ..., vi} as intermediate vertices.

Thus, we have k subproblems, one for each sink ti, where i = 1, 2, ..., k. For each subproblem, we need to compute the probability that the random walk reaches the corresponding sink ti, using only a subset of the intermediate vertices {v1, v2, ..., vi}.

We can define a base case as P[1][j] = Pr(s → v1 → tj), the probability of reaching tj from s using only the edge s → v1 and the edge v1 → tj.

The main problem we want to solve is P[k][j], the probability of reaching tj from s using all the intermediate vertices {v1, v2, ..., vk}.

c.)Recurrence that relates to the Subproblems:

we can compute P(i, v) recursively as follows:

P(i, v) = 1 if v = ti

P(i, v) = 0 if v ≠ ti and v is a sink (i.e., there are no edges leaving v)

$$P(i, v) = \sum P(i, w) * p(v \rightarrow w) \text{ for all edges } (v \rightarrow w) \text{ in } E$$

P[i][j] = sum(P[i-1][u] * Pr(ui → tj)) for all edges (ui → tj) in the graph, where u ranges over all vertices that have an edge directed to tj.

In other words, to compute the probability of reaching sink tj using intermediate vertices {v1, v2, ..., vi}, we sum over all vertices u that have an edge directed to tj. For each such vertex u, we compute the probability of reaching u using only intermediate vertices {v1, v2, ..., i-1}, and then multiply it by the probability of reaching tj from u using the edge ui → tj. We sum over all such products to obtain the probability of reaching tj using intermediate vertices {v1, v2, ..., i}.

This recurrence is correct because the probability of reaching sink tj using intermediate vertices {v1, v2, ..., i} depends only on the probabilities of reaching each vertex ui that has an edge directed to tj, using only intermediate vertices {v1, v2, ..., i-1}. By computing these probabilities recursively and summing over all possible ui, we obtain the probability of reaching tj using all the intermediate vertices {v1, v2, ..., i}.

d.)Subproblem solving the original Problem:

The subproblem that solves the original problem is the probability of reaching each sink vertex tj using all the intermediate vertices {v1, v2, ..., k}. We can compute this probability by using the recurrence relation P[i][j] = sum(P[i-1][u] * Pr(ui → tj)) for all edges (ui → tj) in the graph, where u ranges over all vertices that have an edge directed to tj, and i ranges from 1 to k.

The probability of reaching a sink vertex tj using all the intermediate vertices {v1, v2, ..., k} can be obtained by computing P[k][j] for each sink vertex tj. This is because P[k][j] represents the probability of reaching sink tj using all the intermediate vertices {v1, v2, ..., k}. By computing P[k][j] for all sinks tj, we obtain the probability of reaching any sink using all the intermediate vertices {v1, v2, ..., k}.

Therefore, the subproblem of computing the probability of reaching each sink vertex tj using all the intermediate vertices {v1, v2, ..., k} is the key subproblem that solves the original problem.

e.)Description of The Dynamic Programming algorithm:

The algorithm uses dynamic programming to compute the probability of reaching each sink vertex ti from the source vertex s in a directed acyclic graph with weighted edges. It initializes an array P(i, v) for all i in t and v in V, where P(i, v) represents the probability of reaching sink i from vertex v.

The algorithm then proceeds to fill in the values of the array P using a topological order of the vertices. For each vertex v, if v is the source vertex s, then P(i, v) is set to 1 for all i in t. Otherwise, P(i, v) is set to 0 for all i in t.

For each outgoing edge (v, w) from vertex v, the algorithm updates the probability of reaching each sink vertex i in t from vertex w. The new probability is computed by multiplying the probability of reaching vertex v from the source s (P(i, v)) with the probability of taking the edge (v, w) (p(v -> w)).

Finally, the algorithm outputs the probability of reaching each sink vertex ti from the source s by outputting the value of P(i, ti) for all i in t.

The time complexity of this algorithm is O(|V|*|E|*k), where |V| is the number of vertices, |E| is the number of edges, and k is the number of sink vertices. The space complexity is also O(|V|*k) for the array P.

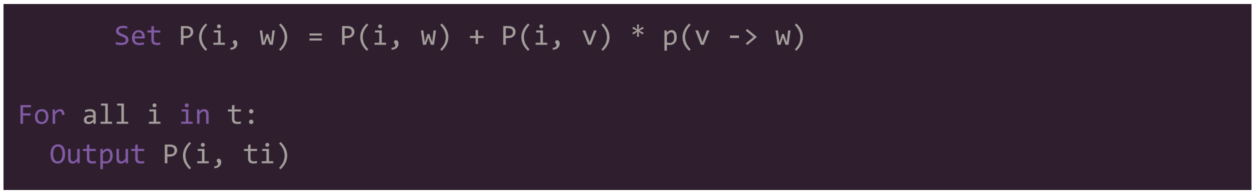
```
Inputs: G = (V, E), where V is the set of vertices and E is the set of
edges
      s, the source vertex of G
      t = {t1, t2, ..., tk}, the set of sink vertices of G
      p(e), the probability of using edge e in the random walk

Outputs: P(ti), the probability of reaching sink ti from s for all ti in t

Initialize an array P(i, v) for all i in t and v in V, where P(i, v)
represents the probability of reaching sink i from vertex v

For all v in V:
  If v is the source vertex s:
     Set P(i, v) = 1 for all i in t
  Else:
     Set P(i, v) = 0 for all i in t

For all v in V in topological order:
  For all edges (v, w) in E:
    For all i in t:
```



```
      Set P(i, w) = P(i, w) + P(i, v) * p(v -> w)

For all i in t:
  Output P(i, ti)
```

f.) Justification for the running time of the Algorithm:

The time complexity of the dynamic programming algorithm described above is O(|V|*|E|*k), where |V| is the number of vertices in the graph, |E| is the number of edges in the graph, and k is the number of sinks.

The initialization of the P array takes O(k*|V|) time, since there are k*t vertices to initialize to 0 or 1.

The loop over all vertices in topological order takes O(|V|*|E|) time, since each edge is processed once and each vertex is processed once.

For each edge (v, w), the algorithm updates the probability of reaching each sink from w based on the probabilities of reaching each sink from v, which takes O(k) time. Therefore, the total time complexity of this loop is O(|V|*|E|*k).

Finally, the algorithm outputs the probabilities of reaching each sink, which takes O(k) time.

Therefore, the total time complexity of the algorithm is O(k*|V| + |V|*|E|k + k) = O(|V||E|*k). This is a polynomial time complexity, which means that the algorithm is efficient for most practical instances of the problem.

Reference Links:

1.) https://web.stanford.edu/class/archive/cs/cs161/cs161.1176/maxflow_problems.pdf
2.) https://courses.csail.mit.edu/6.006/oldquizzes/solutions/quiz2-s2011-sol.pdf
3.) https://www.math.cmu.edu/~af1p/Teaching/GT/CH10.pdf