DR. RAJIV GANDHI ALGORITHM ASSIGNMENT

HW2 -

1.Given a directed graph G with n vertices represented using an n x n adjacency matrix , give an algorithm that determines whether there is a node in G whose indegree is n-1 and outdegree is 0.

Ans :

To determine whether there is a node in a directed graph G represented by an n x n adjacency matrix where the indegree is n-1 and the outdegree is 0, we can follow these steps:

1) Iterate over each row and column of the adjacency matrix to check for nodes that meet the specified criteria.

2) For each node i, check if the sum of the elements in the i-th row (indegree) is equal to n-1 and the sum of the elements in the i-th column (outdegree) is 0.

3) If such a node is found, return that node as the answer.

4) If no such node is found after checking all nodes, then there is no node in the graph that satisfies the given conditions.

Here is an algorithm to determine the node with indegree n-1 and outdegree 0:

1) Initialize a variable found_node to -1 (indicating no node found).

2) Iterate over each node i from 0 to n-1:

a. Initialize indegree_sum = 0 and outdegree_sum = 0.

b. Calculate the sum of the i-th row to get indegree_sum.

c. Calculate the sum of the i-th column to get outdegree_sum.

d. Check if indegree_sum is equal to n-1 and outdegree_sum is 0.

e. i if true, set found_node = i and break out of the loop.

3) If found_node is still -1 after checking all nodes, then no such node exists in the graph.

This algorithm efficiently scans through the adjacency matrix of the directed graph to identify a node with indegree n-1 and outdegree 0. If such a node is found, it is returned as the solution; otherwise, the algorithm concludes that no such node exists in the graph.

5.Give an efficient algorithm that takes as input a directed acyclic graph G = (V,E), and two vertices s, t E V, and outputs the number of different directed paths from s to t in G.

Ans :

To find the number of different directed paths from vertex s to vertex t in a directed acyclic graph G = (V,E), we can use a dynamic programming approach.

Here is an efficient algorithm:

1. Initialize an array count[V] where count[i] will store the number of different directed paths from s to i.

2. Initialize count[s] = 1 as there is only one way to reach s from s (which is directly to itself).

3. For each vertex i in topological order of the graph (starting from s), do the following:

a. For each outgoing edge (i, j) from vertex i, update count[j] by adding count[i] to it. This means that the number of paths to vertex j is incremented by the number of paths to vertex i.

4. After processing all vertices in topological order, the value of count[t] will represent the number of different directed paths from s to t in the graph.

Let's consider an example to illustrate this algorithm:

Given graph G:

V = {s, v1, v2, v3, t}

E = {(s, v1), (s, v2), (v1, v3), (v2, v3), (v3, t)}

1. Initialize count[] = {0, 0, 0, 0, 0}

2. Set count[s] = 1

3. Topological order: s, v1, v2, v3, t

a. For vertex s: No outgoing edges to update.

b. For vertex v1: Update count[v3] = count[v3] + count[v1] = 0 + 1 = 1

c. For vertex v2: Update count[v3] = count[v3] + count[v2] = 1 + 1 = 2

d. For vertex v3: Update count[t] = count[t] + count[v3] = 0 + 2 = 2

4. The number of different directed paths from s to t is count[t] = 2.

Therefore, the algorithm efficiently calculates the number of different directed paths from s to t in a directed acyclic graph.

6.Consider a weighted, directed acyclic graph G = (V, E) in which the edges that leave the source vertex s may have negative weights and all other edge weights are non-negative. Does Dijkstra's algorithm, started at S correctly compute the shortest paths from s to every other vertex in the graph? Prove your answer.

Ans :

No, Dijkstra's algorithm may not correctly compute the shortest paths from the source vertex s to every other vertex in a weighted, directed acyclic graph (DAG) G = (V, E) if the edges leaving s have negative weights.

Dijkstra's algorithm relies on the assumption that all edge weights are non-negative. This assumption ensures that once a vertex is visited and its shortest path is determined, it will not be revisited and the shortest path to it remains valid.

However, in the case where edges leaving the source vertex s have negative weights, Dijkstra's algorithm may incorrectly determine the shortest paths due to the possibility of a negative cycle. Even in a DAG, if there exists a negative cycle reachable from s , Dijkstra's algorithm may get stuck in a loop, continually revisiting vertices and updating their distances, leading to incorrect shortest path calculations.

To prove this, let's consider a simple example:

s

/ \

v1 v2

\ /

v3

Let's assume the weight of edge (s, v1) is -3, and the weights of all other edges are non -negative.

Now, Dijkstra's algorithm may explore the following sequence:

1. Start at s.

2. Visit v1 with distance 0.

3. Visit v3 from v1 with distance -3.

4. Return to v1 from v3 with distance -3.

5. Continue looping between v1 and v3 .

As there's a negative cycle (going from v1 to v3 and back to v1) , Dijkstra's algorithm will not correctly terminate and will not provide the correct shortest paths.

Therefore, Dijkstra's algorithm, when started at s does not correctly compute the shortest paths from s to every other vertex in the graph if there are negative-weight edges leaving s and forming a negative cycle.

HW3 –

5.Professor Stewart is consulting for the president of t corporation that is planning a company party. The party has a hierarchical structure; that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a convivality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend.

Professor Stewart is given the tree that describes the structure of the corporation. Each node of the tree holds in addition to the pointers, the name of an employee and that employee's convivality ranking.

(a) Describe an algorithm to make up a guest list that maximizes the sum of the convivality ratings of the guests. Analyze the running time of your algorithm.

(b) How can the professor ensure that the president gets invited to his own party?

Ans:

The problem exhibits optimal substructure in the following way:

If the root r is included in an optimal solution, then we must solve the optimal subproblems rooted at the grandchildren of r. If r is not included, then we must solve the optimal subproblems on trees rooted at the children of r.

The dynamic programming algorithm to solve this problem works as follows:

We make a table C indexed by vertices which tells us the optimal conviviality ranking of a guest list obtained from the subtree with root at that vertex. We also make a table G such that $G[i]$ tells us the guest list we would use when vertex i is at the root.
Let T be the tree of guests. To solve the problem, we need to examine the guest list stored at $G[T.root]$.

First solve the problem at each leaf L.

If the conviviality ranking at L is positive, $G[L]=\{L\}$ and $C[L]=L.convivC[L]=L.conviv$.

Otherwise $G[L]=\varnothing$ and $C[L]=0$.

Iteratively solve the subproblems located at parents of nodes at which the subproblem has been solved.

In general for a node x,

$C[x]=\max($ y is a child of $x\sum C[y]$, $x.conviv +$ y is a grandchild of $x\sum C[y]$ $)$.

The runtime is $O(n)$ since each node appears in at most two of the sums (because each node has at most 1 parent and 1 grandparent) and each node is solved once.

6.Suppose we are given a set S = { s1 , s2 , ... , sn } of positive integers such that Σ^n i=1 Si =T .

Give an O(nT) time algorithm that determines whether there exists a subset U C S such that Σ si E U Si = Σ si E S\U Si .

Ans:

To determine whether there exists a subset U⊆S such that the sum of elements in U is equal to the sum of elements in S\U, we can use a dynamic programming approach.

Here is a step-by-step algorithm:

1. Create a 2D array dp of size (n+1)×(T+1), where dp[i][j] will represent whether there exists a subset of s1,s2,...,si with a sum of j.

2. Initialize dp[0][0]=True, meaning that it is always possible to achieve a sum of 0 with an empty subset.

3. For each i from 1 to n, iterate over all possible sums j from 0 to T.

4. Update dp[i][j] as follows: - dp[i][j]=dp[i−1][j] if si>j, meaning we can't include si in the subset summing to j. - dp[i][j]=dp[i−1][j]ordp[i−1][j−si] if si≤j, meaning we can either exclude or include si to achieve a sum of j.

5. After completing the iteration, check if dp[n][T] is True. If True, then there exists a subset U such that the sum of elements in U is equal to the sum of elements in S\U; otherwise, such a subset does not exist.

The time complexity of this algorithm is O(nT) since we are iterating through all elements of S and considering all possible sums up to T in the dynamic programming approach.