1. Solution: -

i) Run DFS on the graph G, making sure every vertex is visited at least once and make note of the

finish time of every node

ii) Reverse every edge in the graph and set r(v) of every vertex as negative infinity.

iii) Start running DFS on the reversed Graph according to the decreasing order of the finish times of

vertex

iv) Now for every depth-first forest create a set of nodes present in the forest (connected

components) and keep note of maximum value of *l(u)* of each set (consider this as r(v) of that set).

(step iv will return us acyclic graph of connected components with the maximum *l(u)* on each)

v) Run DFS from every source node on graph returned by step 4. For every node

If r(v) of parent is greater than r(v) of child then set child r(v) to parent`s r(v)

vi) Reverse all edges again so that original graph is retrieved at the end,

Correctness: -

**Lemma1:**

r(v) for a strongly connected component is the vertex with of maximum l(v) among all vertices

present in that component.

**Proof:** Proof by contradiction.

r(v) is the vertex that has largest influence among all vertices reachable from v in graph G. …***Given***

Let us assume that there exists a vertex in a strongly connected component whose r(v) doesn`t

have the maximum influence in the connected components.

For Graph G(V, E)

{ v V for which r(v) = v` and v`` V such that l(v`) < l(v``) }

Now according to the definition of connected components, it is a set of vertices in which every vertex is reachable from every other vertex in that set. Thus, v`` will be reachable from every other vertex in V and vice-a-versa. So, v`` is also reachable from the point v and as per the given definition of r(v) it should reflect the maximum influence vertex which is v``.

This contradicts our assumption is incorrect and hence lemma is proved.

Lemma1

r(v) for a strongly connected component is the vertex with of maximum l(v) among all vertices

present in that component.

Proof:

Will prove this by contradiction.

Let us assume that for all vertices in a strongly connected components

2. Solution: -

Logic –

* Change vertex structure to save details of vertices travelled so far in it (route). Basically every time you visit a vertex you save the path taken to reach to that vertex. The path is nothing but list of vertices travelled to reach to the current vertex.
* A vertex is considered as a loop vertex if it can reach to its parent vertex.

X Y

For an edge e = connecting vertex {x, y}, vertex x is a loop vertex if route of x contains y

* Maintain a list to save all the loop vertex(loopVertices).

In above example loopVertices will have ‘x’ in it.

* Return True, if route of a destination vertex has any vertex that is present in loopVertices.

Procedure –

|  |
| --- |
| **DetermineCertainCycle (G, SourceVertex v, DestinationVertex u):**  Stack <Node> s = new Stack () list <**Node**> loopVertices = **new** ArrayList<loopVertices> () list <**Node**> visitedVertices = **new** ArrayList<loopVertices> () s.push(v)  /\* This is sort of DFS from the source vertex v \*/ **while**(! s.isEmpty()) {  **Node *n*** = s.pop()  **forEach**(**ChildNode** vn : children of **n**) {  **pathTill** = **n**.**route**  **pathTill**.add(**n**)  vn.**route**.add(**pathTill**)  **if** (! visitedVertices **contains** vn) {  s.push(vn)  }  **else if** (visitedVertices **contains** vn **&& n**.**route contains** vn) {  loopVertices.add(**n**)  if (vn **== v**) {  **return** True  }  }  visitedVertices.add(**n**)  } }  **forEach**(**Node intermediateV** : **u**.**route**) {  **if** (loopVertices **contains intermediateV**) {  **return** True  } }  **return** False |

Analysis –

* First while loop is nothing but DFS from source vertex v, thus the running time of first loop will be

*(m + n)* where m and n represents the number of edges and vertices respectively.

* The forEach loop will go through each vertex in the route of the destination node and check whether it presents in the list of loop vertices or not. We can do the search operation (contains) in constant time by making use of some data structure like Hash Set instead of list. In worst-case destination node will have all remaining nodes in route thus, the running time of the second loop will be

*(n - 1)* where n represents the number of vertices in the graph.

* Therefore, ultimately the total running time complexity of the algorithm will be  ***(m + n)***

Correctness –

**Lemma1:**

If in a graph G, if there exists a back edge between child vertex to one of its parent vertex then the graph contains cycle.

**Proof:** Proof by induction.

* Let us assume that the lemma is true for child depth ‘m’,

that is lemma holds True if there is a back edge after m vertices where 0 < m < n

* Base Case –

Let i be an integer value such that 0 < i < m

When m = 0

When there are no child nodes at that time there won`t be any back edge so graph will be acyclic.

When m =1

Vi U

* Induction Step –

Suppose the lemma works when i = m; now suppose we remove the back edge from the Vi and point it to a new node Vi+1.

Vi Vi-1 V2 V1 U

If suppose Vi+1 has a back edge that goes to U then again, we end up having a cyclic graph containing U, V1, V2, …, Vi, Vi+1 vertices.

* Thus, from the induction proof our lemma is correct.

3. Solution: -

Logic –

i) For every process calculate ti/wti and save it in an array list

ii) Sort the list in increasing order ti/wti

iii) Schedule the process according to the sorted list (process with minimum ti/wti is executed first)

iv) Calculate the total weighted finish time for the schedule formed in step 3

6(a). Solution: -

Logic –

i) For each edge in the graph, multiply the weight by -1

(make the weights negative such a way that maximum wt vertex becomes minimum & vice-versa).

ii) Apply Kruskal`s algorithm and create a minimum spanning tree

iii) Now calculate the total weight of the spanning tree formed in step 2

iv) Return the value formed in step 3 by multiplying it by -1

6(b). Solution: -

Logic –

i) Find maximal spanning tree just like the one done in question 6(a) of this assignment and create

a list of edges that are present in the mst

ii) Remove each v belonging to the given graph that is present in mst found in step 1

iii) Return list of edges remained in step 2