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)

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,

2. Solution: -

Logic –

* Change node structure to save details of vertices travelled so far in it (route). Every time you visit a vertex you save the path taken to reach to that vertex.
* Maintain a list (loopVertices) to save vertices that leads to a loop.
* Return True if destination vertex`s route has a vertex that is present in loopVertices.

|  |
| --- |
| Stack <Node> s = newStack() list <**Node**> loopVertices = **new** ArrayList<loopVertices> () list <**Node**> visitedVertices = **new** ArrayList<loopVertices> () s.push(v)  **while**(! s.isEmpty() && *all* vertices not **in** visitedVertices) {  **Node *n*** = s.pop()  **forEach**(**Node** vn : childre of **n**) {  **pathTill** = **n**.**route**()  **pathTill**.add(**n**)  vn.**route**.add(**pathTill**)  **if** (! visitedVertices **contains** vn) {  s.push(vn)  }  **else if** (**n**.**route contains** vn) {  loopVertices.add(**n**)  }  visitedVertices.add(**n**)  } }  **forEach**(**Node** vn : ***n***.**route**) {  **if** (loopVertices **contains** vn) {  **return** True  } }  **return** False |

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