Yash Avinash Patole

Course : CS600

Homework 6

CWID: 10460520

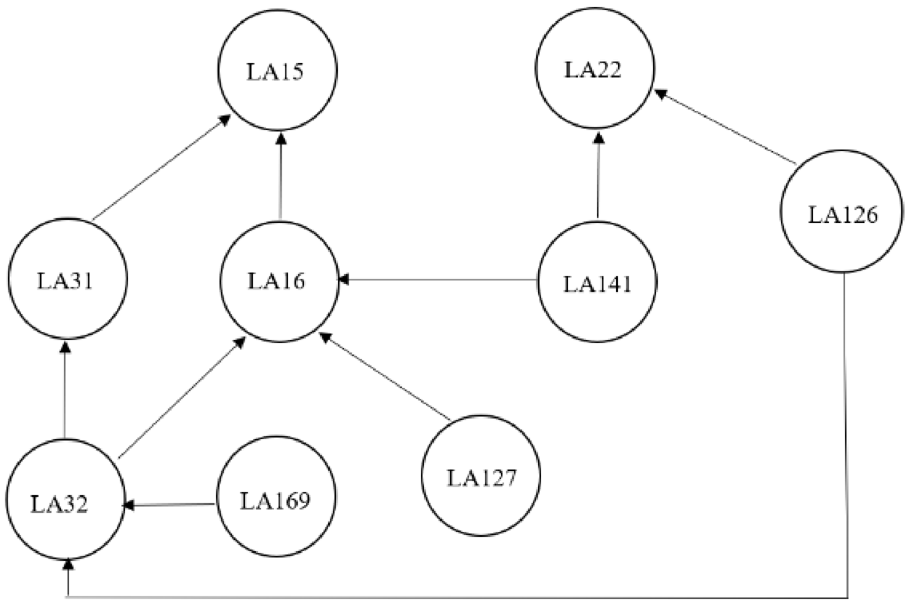
Chapter 13

Exercise 13.7.4

Answer:

LA15, LA22, LA16, LA31, LA32, LA126, LA127, LA141, LA169

Reference for the course perquisites for each language to follow the sequence.



Exercise 13.7.19

Suppose G is a graph with n vertices and m edges. Describe a way to represent G using O(n + m) space so as to support in O(log n) time an operation that can test, for any two vertices v and w, whether v and w are adjacent.

Answer:

Given: We have the two vertices v and w of Graph G.

Step 1: Locate the connected component in the graph.

Step 2: Assign a number to each vertex, as well as its start and end times.

Step 3: When the parser has covered both vertices v and w, stop parsing.

Step 4: The compiler will take log n time until step 3. Where n is the number of graph vertices.

Step 5: When the compiler reaches the second vertex after step 4, it verifies the previous one. If the preceding vertex is the v of the w vertex. It indicates that two vertices are close together.

Step 6: The compiler will take only O (1) time in step 5.

As a result, the total time required by the compile is = log(n) + O. (1) = time log(n)

It will take O (n + m) space complexity to store the n vertices and m edges.

Exercise 13.7.37

Answer:

a) Proposition: Leaf nodes cannot be the \centre of a tree- Why? It's obvious that the peripheral nodes will be at the longest distance from the other end of the path. So, we can safely remove all leaves from the tree T, name the modified tree as T.
Claim: T, is again a tree.
Proof: Because we only removed the leaf nodes, it is still connected.
It's also acyclic because we only removed connections and did not add any new ones.

Rep the previous step. T, remove the leaf nodes from this tree. Continue these steps until we have one or two nodes left. In the case of two nodes, both are candidates to become the tree's center.

Algorithm:
Insert all leaf nodes into FIFO queue Q.
This can be done by traversing(ustng DFS or BFS) from the root node, and insert all leaf nodes in Q.
while Q has more than 2 nodes
node n 🡐 Q firstnode // remove first node from Q
pnumChlldrens = pnumChiIdrens
if p 🡐 numChlIdrens=O
insert p into Queue
Return
// parent of node n
I/ decrement the number of childrens of parent
if parent has O childrens, then it becomes leaf
// Hence, insert into Q
Number of nodes In Q(I or 2) islare the centres of the Tree T.
Time complexity of the algorithm is O(n). Traversing the tree is done in O(n). The "while loop" processes
each node only one time. Hence, the overall complexity is O(n).
b) No, the icentre need not be unique, There could be at most two distinct centres In a free tree. Consider the longest path P of tree T. The median of this path P is the centre of the Tree.
If length (P) is odd, then there is only one \centre of the Tree
else if length(P) is even, two mid nodes are the \centre of Tree.
Please let me know if the explanations are not clear to the understanding.

Exercise 14.7.11

Answer:

Both algorithms are guaranteed to produce the same shortest path weight, but if there are multiple shortest paths, Dijkstra's will choose the greedy path, and Bellman-Ford will choose the shortest path based on the order of relaxations, and the two shortest path trees may differ. No. If there is a zero-weight cycle, relaxing an edge may cause parent pointers to be messed up, making it impossible to recreate a path back to the source node. The simplest example is a vertex v with a zero-weight edge pointing back to itself. If we loosen that edge, v's parent pointer will return to itself. When we try to recreate a path from one vertex to another, to the source, if we go through v, we will be stuck there. The time complexity of this solution wil be O(nlogn).

Exercise 14.7.17

Answer:

Set the distances between all other nodes to infinity.

Set the status of all other nodes to unvisited.

Make a set of all unvisited nodes (excluding the start node).

```
Therefore,
Current node = starting node
Cd = 0
boolean movingLeft = true;

while (current node != destination node and its distance < infinity)
{
  for (each unvisited neighbor of current node)
  {
    Set neighbor.distance to the lesser of its current value and (Cd + the length of
the edge between the current and neighbor).
  }
  Mark current node as visited.
  current = node from unvisited set with smallest distance
}

Minimum distance to end moving left = Cd movingLeft = false
Set the status of all nodes to unvisited and their distances to 0.
Set the current node to start.
Set the current to visited and the distance to zero.

By moving to the right, we will now find the shortest path to the end.
while (current node != destination node and its distance < infinity)
{
  for (each unvisited neighbor of current node)
  {
    Set neighbor.distance to the smaller of its current value and (Cd + length of
edge between current and neighbor).
  }
  Mark current node as visited.
  current = node from unvisited set with smallest distance
}
```

Minimum distance to end moving right = Cd
To find the overall minimum distance to the end, compare distance to end moving left with distance to end moving right. There is no way to reach the end if both are infinity.

Exercise 14.7.20

Answer:

1. In the graph, each flight represents an edge. The flight time is proportional to the edge weight.

2. In the graph, each airport creates a node.

3. The constraint is that the difference between the next flight's departure time and the previous flight's arrival time must be greater than c. (a).

4. The above condition reduces the number of edges we must traverse through the graph to find the best solution.

6. The traversal concludes when we find a set of paths from the source airport to the destination airport and report the cost of each path. Cycles must be detected and eliminated by keeping track of whether or not a node has already been visited.

7. Its execution time will be O(n+m), where n is the number of nodes and m is the number of flights. We must also keep an eye out for the shortest route.