Jason Less
404-640-158
CS180

CS180 Homework 3

Question 1

The LCA of two nodes u and v in a directed rooted tree T can be found in O(h) time by doing a little preprocessing before-hand. As the preprocessing doesn't have knowledge of the input nodes u and v, it will consist of recording certain information about all of the nodes in the tree: (1) the direct parent of each node in the tree (excluding that of the root of the tree), and (2) the depth of each of the nodes within the tree. These two pieces of information can be stored in two hash sets; for example, called parent and level, where parent[v] represents the parent of node v and depth[v] represents the depth of node v. This preprocessing will be carried out via a level-order traversal and use of a queue to process each level of the tree.

This preprocessing information is useful in order to achieve O(h) time complexity for finding the LCA. Basically, we want to start from nodes u and v and then traverse back up the tree such that both nodes are on the same level (i.e. same depth) in the tree.
After each node has traversed up the tree to the same level, check to see if they are the same node. If so, then this is the LCA (as the LCA of two nodes u and v could in fact be either u or v). If not, then move each node u and v one step up the tree using the parent hash set. At some point, both nodes will converge at the same point, which is the LCA.

Preprocessing:

- Use of two hash sets called parent (to store the direct parent of a given node) and depth (to store the depth of the node in the tree)
- Use of a queue q to perform level-order traversal on the tree to fill the two hash sets
    - The queue will consist of a tuple of tuples: <x, <y, z>> where x is the value of node v, y is the depth of node v, and z is the parent of node v
- Push the root to the queue (where y = 0, and z = -1 as the root has no parent node)
- Pop the front node off the queue and:
    - Add the corresponding parent and depth information to the two hash sets
    - Push any children to the queue with the updated parent and depth information
- Repeat the steps until processing all nodes within the tree

Time complexity of the level-order traversal is O(|V| + |E|) which is bound by the number of edges in a connected graph, so just O(|E|). The reasoning for this is that the minimum number of edges in a connected graph is |V| - 1. Thus, |E| >= |V| - 1, and as 1 is just a constant value, we can disregard it, so |V| <= |E| (i.e. the number of vertices is bound by the number of edges); and thus, O(|V|) <= O(|E|). Even if the graph is disconnected, the sub graphs can be thought of in this same way; and thus, preprocessing in general is O(|E|).

Jason Less
404-640-158
CS180

Algorithm:

- Given nodes u and v, retrieve their depths from the depth hash set (constant time)
- If node u's depth > node v's depth, then move node u one step up in the tree (i.e. u = parent[u]) (and vice versa if node v's depth > node u's depth). Keep doing this until the nodes are at the same depth
- Once the nodes are at the same depth, check if they are the same node
    - If so, we have found our LCA, so done
    - Else, move node u and node v one step up in the tree using the parent hash set (i.e. u = parent[u] and v = parent[v])
        - Again, check if they are the same node (i.e. LCA), and if not, then repeat this process until finding an LCA in the case of a connected graph, or finding no LCA which could be the case from an unconnected graph

Time complexity of this algorithm is O(h) as we start from nodes u and v and at worst have to traverse up the graph h steps (i.e. the distance between nodes u and v) until they converge at the LCA.

Jason Less
404-640-158
CS180

Question 2

The algorithm to find such a subset S can be achieved by using an in-degree array to find which nodes belong in the subset S and those that do not. Nodes in the in-degree array that have a value of 0, do not belong in the subset S as nodes in S must have at least one incoming edge from another node in S. Therefore, S can be found by using the in-degree array to remove cities from the picture that can't possibly belong to S (i.e. in-degree value of 0), and then updating the city that the outgoing edge of the offending city goes to. If the city that the offending city's outgoing edge goes to has an in-degree value of 0 as a result of disconnecting the edge, then this city too can't belong to S. By repeating this procedure, we can get the subset S.

Algorithm:

- This algorithm assumes that the input consists of an array of all of the flights within the graph
- Using the flights, compute the in-degree array for the entire graph
    - Processes all n cities so time complexity for this step is O(n)
- After constructing the in-degree array, iterate through the in-degree array to find all the cities with in-degree of 0, and push these cities to a queue
    - Iterates through all n cities in the in-degree array so time complexity for this step is O(n)
- Pop the front city u off of the queue, and update the in-degree of the city v that city u travels to (i.e. disconnect edge connecting the outgoing edge from node u to the incoming edge of node v)
    - If the city v's in-degree now has a value of 0, then push it to the queue to process
- Repeat this process until we have processed all cities in the queue
    - At worst, this step processes all cities in the graph, as each time we process a city in the queue, it results in a new city being added to the queue so time complexity is O(n)
- Finally, iterate through the in-degree array and count the number of cities that have a value > 0, which will be the size of subset S
    - Iterates through the in-degree array so time complexity is O(n)

Overall, the algorithm has a time complexity of O(n) as described above.

Jason Less
404-640-158
CS180

Question 3

The problem of finding out whether a directed graph is acyclic (i.e. a DAG) can be solved by finding if there is a topological ordering that exists within the graph (which is only possible if the graph is a DAG). Finding if a topological ordering exists can be achieved by performing a topological sort on the graph, which can be achieved in O(|V| + |E|) or O(|E|) time complexity (see the explanation in question 1 for the reasoning the algorithm is O(|E|)).

The topological sorting algorithm can be carried out by using a data structure (e.g. an array) to keep track of the incoming edges for each node in the graph and also constructing an in-degree array for each node in the graph as well. Push all nodes with in-degree of 0 to a queue for processing. If the in-degree of a given node has a value of 0, then this means that we have removed all of its dependencies and can process this node. For all nodes u with in-degree of 0, update the in-degree array (i.e. remove all incoming edges for nodes that node u is connected to) and if they have in-degree of 0, then also push them to the queue for processing. If we are able to process all the nodes (i.e. all of the dependencies to process a given node have been removed), then a topological ordering exists, and the graph is a DAG. If not, then a cycle must exist.

Algorithm:

```
# where edges is an array of tuples (a, b) where a = src node and b = dst node
bool isDAG (input: edges)
  # Build graph where graph is a 2D array that maps from a node to an array of the nodes it connects to
  # Build in-degree array
  for (each edge : edges)
    graph[edge.first].add(edge.second)
    indegree[edge.second]++

  # Keep track of number of nodes in the graph to process
  size = graph.size

  # If a node has in-degree = 0, then it has no dependencies and can be processed, push it to the queue
  for (each node : indegree)
    if (indegree of node is 0) push node to queue

  # while there are still nodes left to process in the queue
  while queue is not empty
    pop front node u from the queue
    size--

    for (each neighbor : u)
      indegree[neighbor]--
      # if indegree of the neighbor node is 0, then we can process it, so add to queue
      if (indegree[neighbor] = 0) push neighbor to queue


  # afterwards, check to see if all the nodes have been processed, if so, then no DAG
  if size = 0 return true
  else false
```

Jason Less
404-640-158
CS180

Question 4

In the previous question, I performed topological sort on a graph using Kahn's algorithm to show if a graph is a DAG or not. Topological sorting can also be executed by performing DFS via graph coloring. The idea is to perform DFS from a given node u, and then visit its neighbors, and then DFS from the neighbors and so on. In addition to DFS, make use of three colors: (1) white for uncolored, (2) gray for currently being processed, and (3) black for already processed. If a node is currently being processed, this means that not all of the descendants of this node have been visited yet. If a node is already processed, this means that all descendants of this node have been visited. If at any point in the process, a node is visited that has already been colored gray, then this means that there is a cycle in the graph. For this algorithm, a graph will be constructed (similarly to question 3) to keep track of the incoming edges for each of the nodes in the graph. In addition, an enum array called colors will be used to keep track of the coloring scheme of the nodes (where a value in colors can be either white, gray, or black). The time complexity of DFS topological sort is O(|V| + |E|) as we traverse each edge and vertex only once, and this evaluates to O(|E|) as described in the previous two questions. The reasoning that this algorithm works for detecting if a directed graph is acyclic has to do with the gray set. If for a given DFS traversal, we encounter a vertex that is colored gray, this means this vertex has some descendant nodes (i.e. neighbors or neighbors of neighbors) that are still being explored, and these such neighbors have found a path back to a vertex that is also colored gray, indicating a cycle.

Jason Less
404-640-158
CS180

Algorithm:

```
# where edges is an array of tuples (a, b) where a = src node and b = dst node
bool isDAG (input: edges)
  # Build graph where graph is a 2D array that maps from a node to an array of the nodes it connects to
  for (each edge : edges)
    graph[edge.first].add(edge.second)

  # Initially, all nodes have color white
  for (each node : graph)
    color of node is white

  # Perform DFS traversal from any vertex that hasn't already been processed (i.e. color is white)
  for (each node : graph)
    if (color is white and dfsHelper(node, color) is true) return false

  # graph is a DAG
  return true

# DFS helper function to visit the nodes in the graph and perform graph coloring to detect a cycle
bool dfsHelper(input: node, input: colors array)
  # Currently processing the given node, so color it gray
  color of node = gray

  # visit all neighbors of the current node
  for (each neighbor : node)
    # if the neighbor is also being processed (i.e. color is gray), a cycle exists, return true
    if color of neighbor = gray
      return true

    # if the neighbor hasn't been visited yet, visit it and determine if there is a cycle
    if color of neighbor = white and dfsHelper(neighbor, color)
      return true

  # after visiting all descendants of the current node, it has been completely processed
  color of node = black

  # also, no cycle found from the current DFS traversal
  return false
```