Data Structures and Algorithms Homework 4

Due Wednesday Sept 25; Joseph Sepich (jps6444)

1 Problem 1 Pre and Post Processing

Collaborators: Wikipedia Minimum Spanning Tree

1.1 Part 1a

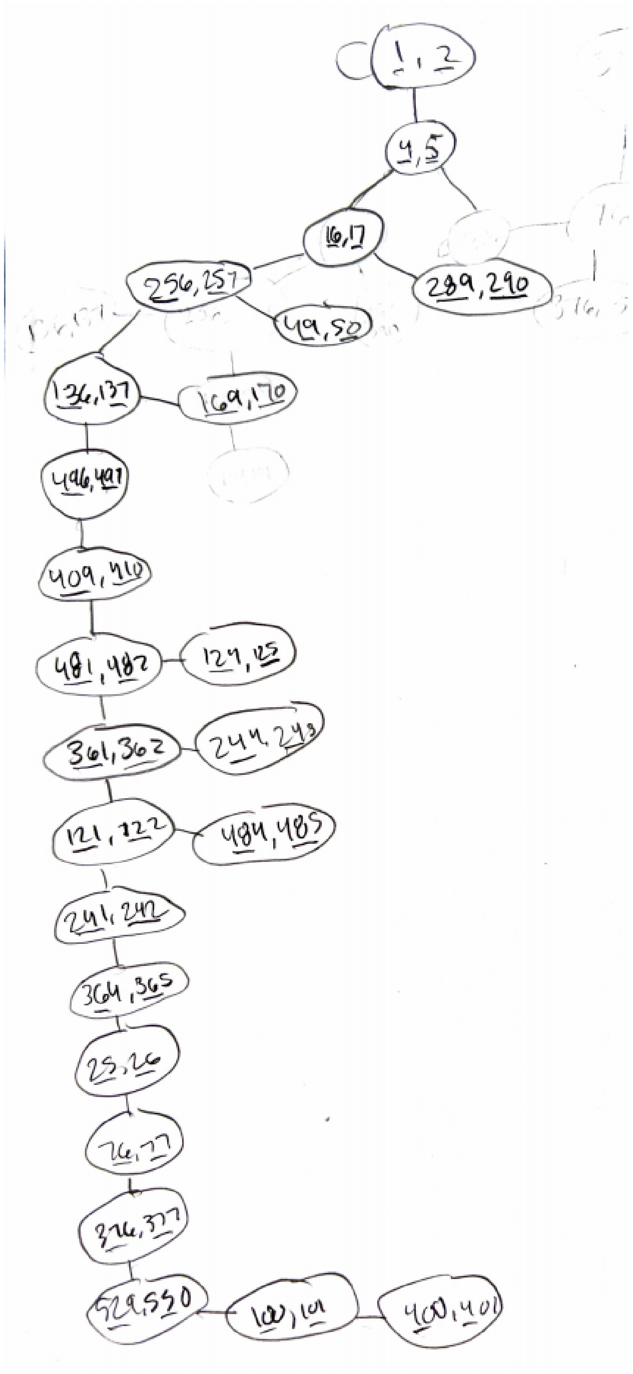
The property that post(u) < post(v) makes v an ancestor of v false. If u and v are both connected to a and you start exploring from a with all the edges ({a, u}, {a, v}), then u and v siblings in the tree and v is not an ancestor of u. In this case we would have pre(u)<post(u)<pre(v)<post(v), so post(u)<post(v).

1.2 Part 1b

We can piggyback off of part a for this problem. If our question in a tree is “is u an ancestor of v”, then we can run DFS on the tree and record all the pre and post numbers. If the post number of v is less than the post number of u, then we know that u and v could be siblings, but if pre(u)<pre(v)<post(v)<post(u) then we know u must be an ancestor of v. If v starts and ends while u is running, then v must be lower in the recursive stack and therefore a descendant of u. We know they are not siblings because then you would have pre(u)<post(u)<pre(v)<post(v). This means we can run DFS to solve the problem, which is a linear time algorithm.

2 Problem 2 Funny Money

To find a combination to obtain a 10$ bill, we can create a DFS tree. Since there are only two options, connected to each node, it the tree is a binary tree. Leaves occur when both choice nodes already appeared as an ancestor to that node. We want a node that gives us [10, 11], since that means we currently can get a 10 dollar bill. An example of the tree can be found below.



According to this DFS tree I made of the possible options, there is no way to obtain a $10 bill through the printing machine starting with a 1 dollar bill.

3 Problem 3 Topological Ordering

3.1 Part 3a

- A - [1, 14]
- B - [15, 16]
- C - [2, 13]
- D - [3, 10]
- E - [11, 12]
- F - [4, 9]
- G - [5, 6]
- H - [7, 8]

3.2 Part 3b

The sources of the DAG are A and B. The sinks of the DAG are G and H.

3.3 Part 3c

1. B
2. A
3. C
4. E
5. D
6. F
7. H
8. G

3.4 Part 3d

In the ordering of this DAG, it is always possible to swap two consecutive nodes that are not connected by edge. An example of this would be A and B, D and E, or G and H. This means that there are 8 possible ways to order the topological sort or 2ˆ3.

4 Problem 4 One-Way Streets

4.1 Part 4a

The roads in the city of Computopia are in the form of a directed graph with the intersections as nodes and roads as edges. The mayor has claimed that any intersection can be reached by starting at any other intersection. This implies that the entire city is one strongly connected component where any U and V have a path from U to V and V to U. We can use our algorithm for finding a strongly connected sink component. To do this we reverse all the edges and run DFS. The vertex with the highest post number is in a source strongly connected component on the reversed graph, but in a sink strongly connected component on the regular graph. If we run DFS from this point and obtain every node in the graph, then we know that the graph has to be a singly strongly connected component, and the mayor's claim holds. Since this algorithm just requires a run of DFS twice to determine the truth value of the original statement, the running time must be 2*O(|v| + |E|) = O(|V| + |E|), which is linear time.

4.2 Part 4b

The mayor's new claim revolves around the town-hall. She says that if you start at the node that is the town-hall, then no matter when you go you can always end at the town hall. This implies that the town-hall is contained in the sink strongly connected component of the graph of roads and intersections. To determine this you can run DFS on the reverse graph and start at the town-hall. If the town-hall has the largest post-number, then it must be in a source strongly connected component of the reversed graph; however there could be multiple source components, but this is okay. In order to figure out if the town-hall is in a source component. Then if the town-hall pre and post numbers are not contained within any previously found source component intervals, then it cannot be a child, so you can remove the source component you found and run the algorithm again. If the town-hall is never in the source component, then the mayor is wrong, however if the town-hall is in a source component, on this reverse graph, then it must be in a sink strongly connected component on the non-reversed graph and would evalute to true.

This algorithm for the more specific claim, would have the same growth rate as the previous (linear), but would have a greater coefficient, because it would invovle running DFS more times in the worst case scenario, since there is more to validate.

5 Problem 5 City Hopping

5.1 Part 5a

Is there a feasible rouute from s to t?

You can easily model a DFS algorithm to understand if s to t is a plausible route. Run a standard DFS algortihm, but do not add the next city node as a child, unless the length value of the edge is less than or equal to your max range. Then you can check to seed if the pre and post number of s encompasses the pre and post number of t.

```
clock = 1

# G is a graph
# L is max range
# s is source city
# t is target city
findRoute(G, L, s, t) {
  visited = new boolean[|V|] # array for storing pre and post numbers
  pre = new int[|V|]
  post = new int[|V|]
  pre[s] = clock
  clock++
  foreach {s, v} that is an edge {
    if (lengths[{s, v}] <= L && !visited[v]) { ## CHECK FOR MILEAGE
      Explore(G, v, visited)
    }
  }
  post[s] = clock

  if (pre[s] < pre[t] && post[s] > post[t]) { ## CHECK FOR CHILD
    return true
  }
  return false
}

Explore(G, s, visited) {
  visited[s] = true
  pre[s] = clock
  clock++
  foreach {s, v} that is an edge{
    if (lengths[{s, v}] <= L && !visited[v]) { ## CHECK FOR MILEAGE
      Explore(G, v, visited)
    }
  post[s] = clock
  clock++
  }
}
```

Since this algorithm is merely DFS with a different coefficient, since it has a different number of steps in each function, it is by definition O(|V| + |E|). The worst case is all vertices are visited and each edge is traversed,

5.2 Part 5b

Finding a minimum tank capacity path can be modeled as a shortest path algorithm, because we want the shortest distance between each city. To do this we can construct a Minimum Spanning Tree and then find the path from s to t.

```
FindMin(G, s, t, L) {
  sort(G.E, L) # sort edges in ascending order: O(ElogE) time

  edgeCount = 0
  edgeIndex = 0
  verticesConnected = new boolean[|V|] ; true if given vertex is connected (also mst vertices)
  mstEdges = list<edges>()

  while (edgeCount < |V| - 1) {
    edge = G.E[0] # smallest edge first

    x = G.E[0][0] # first vertex in edge
    y = G.E[0][1] # second vertex in edge

    if (verticesConnected[x] != true && verticesConnected[y] != true) {
      verticesConnected[x] = true
      verticesConnceted[y] = true
      mstEdges.add({x,y})
      edgeCount++
    }

  }

  ; we now have MST

  preNums = DFS(MST, s, L) ; run DFS from s

  foreach V in verticesConnected {
    if (pre(v) > pre(t)) {
      verticeConnected.Remove(V)
      mstEdges.Remove(v)
    }
  }

  return L[mstEdges[length(mstEdges)]] # mstEdges is an ascending list, last item will be max
}
```

The longest part of this algorithm would be the sorting of the edges, so it would grow at O(ElogE).