2nd Lt David Crow

ENG/20M

Analysis of Algorithms Homework 2

**Chapter 3, Problem 5**

A binary tree is a rooted tree in which each node has at most two children. Show by induction that in any binary tree the number of nodes with two children is exactly one less than the number of leaves.

**Solution**

*Overall, this problem took me 15 minutes. I’d rate it a 3 on the difficulty scale; I needed to review proof by induction, but the problem was otherwise relatively simple.*

To establish a proof by induction, we need to show a basis step, an inductive hypothesis, and an inductive step.

Proof:

Basis step:

A tree with one root node has either zero, one, or two children.

* If it has zero children, there are zero nodes with two children and there is one leaf node.
* If it has one child, there are zero nodes with two children and there is one leaf node.
* If it has two children, there is one node with two children and there are two leaf nodes.

In each case, our theorem holds.

Inductive hypothesis:

In a binary tree with unique nodes, the number of nodes with exactly two children is and the number of leaf nodes is .

Inductive step:

When adding a node to a binary tree with unique nodes, we only have two options:

1. Add to a node with zero children.

By the inductive hypothesis, before adding , we have nodes with exactly two children and leaf nodes; additionally, is a leaf.

After we add , is not a leaf and is a leaf. In other words, we have not changed the number of leaf nodes.

Furthermore, because originally had zero children and it now has one child, we have not changed the number of nodes with two children.

Thus, we still have nodes with exactly two children and leaf nodes in our tree with nodes.

1. Add to a node with one child.

By the inductive hypothesis, before adding , we have nodes with exactly two children and leaf nodes; additionally, is a node with exactly one child.

After we add , is a node with exactly two children and itself is a new leaf node. In other words, we have increased the number of nodes with exactly two children by one, and we have increased the number of leaf nodes by one.

We now have nodes with exactly two children and leaf nodes in our tree with nodes. Thus, our theorem holds.

There are no other cases in which we can add a new node to the tree.

By the principle of mathematical induction, the theorem is proved.

**Chapter 4, Problem 4**

Some of your friends have gotten into the burgeoning field of *time-series data mining*, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges – what’s being bought – are one source of data with a natural ordering in time. Given a long sequence of such events, your friends want an efficient way to detect certain “patterns” in them – for example, they may want to know if the four events

occur in this sequence , in order but not necessarily consecutively.

They begin with a collection of possible *events* (e.g., the possible transactions) and a sequence of of these events. A given event may occur multiple times in (e.g., may be bought many times in a single sequence ). We will say that a sequence is a *subsequence* of if there is a way to delete certain of the events from so that the remaining events, in order, are equal to the sequence . So, for example, the sequence of four events above is a subsequence of the sequence

Their goal is to be able to dream up short sequences and quickly detect whether they are subsequences of . So this is the problem they pose to you: Give an algorithm that takes two sequences of events – of length and of length *n*, each possibly containing an event more than once – and decides in time whether is a subsequence of .

**Solution**

*Overall, this problem took me 30 minutes. I’d rate it a 4 on the difficulty scale; a solution was readily apparent to me, and I only needed to ensure it could run in time. However, the average case analysis and proof of optimality required a bit of effort.*

Let’s jump right in. Here’s my algorithm:

1 If m > n, return is not a subsequence of (assuming we know m and n)

2 Else

3 Let

4 For to do

5 If equals

6 If equals

7 Return is a subsequence of

8 Endif

9 Else

10 Let

11 Let

12 Endelse

13 Endif

14 Else

15 If equals

16 Return is not a subsequence of

17 Endif

18 Else

19 Let

20 Endelse

21 Endelse

22 Endfor

23 Endelse

As we can see, we continue incrementing and until one of the following conditions is met:

* equals ,
* or equals .

At either of those points, the algorithm terminates. If we terminate when equals , we’ve shown that is a subsequence of ; otherwise, we’ve shown that is not a subsequence of .

No matter what, we will not look at more than every element in . Either is not a subsequence, in which case we look at elements of before terminating, or is a subsequence, in which case we look at all elements of , and is necessarily less than (by the definition of a subsequence). Thus, our worst-case runtime is .

In the best (nontrivial) case, either is a subsequence of and composes the first elements of , or is not a subsequence and . In both cases, we only inspect the entirety of the shorter sequence, and our algorithm will thus run in .

The average case performance of this algorithm is very nuanced. For this algorithm, it heavily depends on whether , on the number of unique events, and on the frequency of each event (among other factors). Without knowing more about the problem, we can only constrain our cases to and .

Let’s prove that this algorithm is optimal. That is, let’s prove that it always identifies whether or not is a subsequence of .

To do this, we’ll assume the algorithm doesn’t correctly identify subsequences. This means one of two things:

1. is not a subsequence, but the algorithm says it is; or
2. is a subsequence, but the algorithm says it is not.

For (1) to occur, the algorithm must indicate that some exists at . However, because the algorithm will only indicate that some element exists at if is equal to for some , we know that the algorithm cannot incorrectly identify a non-subsequence.

For (2) to occur, there must exist some event such that the algorithm identifies some (where ) and, later, some (where ) without identifying between the two. However, because the algorithm iterates over the events in one-by-one, we know that cannot be identified before .

Because we’ve reached a contradiction in both cases, we’ve shown that our algorithm is optimal.

**Chapter 4, Problem 19**

A group of network designers at the communications company CluNet find themselves facing the following problem. They have a connected graph , in which the nodes represent sites that want to communicate. Each edge is a communication link, with a given available bandwidth .

For each pair of nodes , they want to select a single path on which this pair will communicate. The *bottleneck rate*  of this path is the minimum bandwidth of any edge it contains; that is . The *best achievable bottleneck rate* for the pair is simply the maximum, over all paths in , of the value .

It’s getting to be very complicated to keep track of a path for each pair of nodes, and so one of the network designers makes a bold suggestion: Maybe one can find a spanning tree of so that for *every* pair of nodes , the unique path in the tree actually attains the best achievable bottleneck rate for. (In other words, even if you could choose any path in the whole graph, you couldn’t do better than the path in .)

This idea is roundly heckled in the offices of CluNet for a few days, and there’s a natural reason for the skepticism: each pair of nodes might want a very different-looking path to maximize its bottleneck rate; why should there be a single tree that simultaneously makes everybody happy? But after some failed attempts to rule out the idea, people begin to suspect it could be possible.

Show that such a tree exists, and give an efficient algorithm to find one. That is, give an algorithm constructing a spanning tree in which, for each , the bottleneck rate of the path in is equal to the best achievable bottleneck rate for the pair .**Solution**

*Overall, this problem took me about 45 minutes. I’d rate it a 5 on the difficulty scale; a possible solution was almost immediately obvious to me, but the proof and analysis was a little more involved than I predicted.*

1 Let represent the set of visited nodes;

initialize with one arbitrary node

2 Let represent the set of all nodes

3 Let represent the desired set of edges;

initialize with the empty set

4 While does not equal

5 Add to the edge from some to some

with the highest ; break ties arbitrarily

6 Add to

7 Endwhile

8 Return the set of edges

Given a graph of nodes, we only add edges to nodes that are not in the explored set ; thus, we add edges to our graph. We know that our graph is connected (that’s stated in the problem), and so we know that we will *always* add exactly edge for each node .

Line 5 in the above algorithm can run in if we implement our set of edges as a priority queue. That is, extracting the edge with the maximum and inserting new edges can run in , so adding the edge with the highest to (as in line 5) takes .

Thus, for each of edges, we run code that requires ). This indicates our running time is .

In the best case, we will never have to decide between multiple edges which has the highest bandwidth-rate. In other words, we will only ever have to pick from one edge at each node, and so this algorithm can run in .

This is unlikely, though. Our best case can only be achieved if nodes have connected edges each and the remaining nodes have edge each. In other words, we’d need a direct path from some to some , and none of the nodes on the path would have any other edges.

Usually, we can expect that a network graph, such as the one employed by CluNet, will have multiple edges leaving multiple nodes. We can’t assume that our graph is as sparse as just explained; still, though, we shouldn’t assume that the graph is fully connected, either. The average case lies somewhere in between, and thus runs somewhere between and . It’s hard to state an exact runtime without knowing just how connected really is, but we can certainly assume that, at some point, our algorithm must determine which edge (of a set of edges) has the best bandwidth-rate. Because this is extremely likely, we *can* say the average case is likely closer to than .

Because and and is connected, we’ve created a tree and, because we’ve connected every node, our tree is spanning.

We just need to show that we’ve created the best achievable bottleneck for each path from to in (we’ll call this optimal). Let’s do that.

Assume we haven’t created an optimal minimum spanning tree. That is, assume there exists some path such that has a worse bottleneck rate than some other path .

This means that there exists an edge and an edge such that . Because our algorithm only adds edges that link visited nodes to unvisited nodes, this also means that our tree does not include . Our algorithm thus selected instead of , but we know that our algorithm selects edges in order of greatest bandwidth rate. In other words, the algorithm would’ve selected before selecting ; thus, we have a contradiction and we know that cannot be worse than any other path .

We have proved the optimality of our algorithm.