CS 317, Assignment 6

Answers

1. An AVL tree is a binary search tree where the following property holds for every node v in the tree: the difference between the height of the left child of v and height of the right child of v is at most 1.

   As in class, you are given a sorted array $A_{1...n}$ of search keys (or strings) along with an array $p_{1...n}$ with the respective search probabilities.

   (a) Design a backtracking algorithm that returns all the possible AVL trees for the given array of keys. Establish the correctness and analyze the running times of your algorithm. Also comment on the optimality of your algorithm.

   ANSWER:

   Let me start with a backtracking solution for the simplest variant of the problem namely, returning an AVL tree (no matter which one) and doing so in the most straightforward though not necessarily most efficient way.

   The choice to be made at each step in the backtracking algorithm is obvious: we pick one of the keys in the current range to be the root of the tree. We return the tree thus constructed, together with its height (so that we can check whether we have an AVL tree). For the time being we do not worry about the average search time so we do not even need to pass probabilities to our algorithm.

```
algorithm SomeAVL($A_{i...j}$):
    if $i > j$ then return (Null, 0)
    else if $i = j$ then return (Node($A_1$), 1)
    else
        for $k = i$ to $j$ do
            $(l, hl) \leftarrow$ SomeAVL($A_{i...k-1}$)
            $(r, hr) \leftarrow$ SomeAVL($A_{k+1...j}$)
            if $|hl - hr| \leq 1$ then return (Node($A_{bk}, bl, br$), $1 + \max\{hl, hr\}$)
```

   The call that provides the overall solution is SomeAVL($A_{1...n}$). Correctness is established by a pretty straightforward structural induction: For the base cases, a null node is the only possible AVL tree for an empty range and similarly the only possible AVL tree for a single key is a node containing that key and having no children.

   In the recursive step the returned tree contains A_k as root. In addition, A is sorted and so the key A_k is larger than all the keys contained in the left child ($A_{i...k-1}$) and smaller than all the keys contained in the right child ($A_{k+1...j}$). The two children are obtained by recursive calls and so they are binary search trees by induction hypothesis. It follows that the whole tree is a binary search tree, as desired. The tree is also an AVL tree since any difference between the heights of the left and right children that is larger than 1 is filtered out by the test $|hl - hr| \leq 1$.

Now back to the problem at hand, we need to generate all the possible AVL trees. The most straightforward solution that comes to mind is to return sets of trees instead of single trees. We will also need to pass along the height of every tree included in the result, as before (to keep checking for the AVL property).

```
algorithm AllAVL($A_{i...j}$):
    if $i > j$ then return $\{(\text{Null}, 0)\}$
    else if $i = j$ then return $\{(\text{Node}(A_1), 1)\}$
    else
        $T \leftarrow \emptyset$
        for $k = i$ to $j$ do
            $L \leftarrow$ AllAVL($A_{i...k-1}$)
            $R \leftarrow$ AllAVL($A_{k+1...j}$)
            foreach $(l, hl) \in L$ do
                foreach $(r, hr) \in R$ do
                    if $|hl - hr| \leq 1$ then $T \leftarrow T \cup \{(\text{Node}(A_{bk}, bl, br), 1 + \max\{hl, hr\})\}$
            return $T$
```

The argument that all the trees returned are AVL trees is a relative simple extension of the single-tree algorithm above, so I am going to leave it as an exercise. Additional;y we need to make sure that the algorithm returns *all* the possible AVL trees for the given set of keys. To see that, we note that there is a single such a tree for each base case (which is correctly returned). Assuming now by induction hypothesis that L and R contains all the possible AVL trees for the keys $A_{i...k-1}$ and $A_{k+1...j}$ respectively, we note that we try all the keys as possible overall root (so that no possible root is overlooked), and we keep all the trees thus obtained that observe the AVL property.

What is more interesting is the determination of the running time, which turns out to be an exercise in determining the properties of the state space of partial AVL trees as explored by SomeAVL. Obviously the number of leaves in such a state space tree is the number of possible AVL trees for the given set of keys, so we have to first find that out.

We have as many leaves in this tree as there are AVL trees for the given sequence of keys. We can ignore the actual values and consider the number $s(h)$ of possible trees of height h with empty nodes observing the AVL property. Since the keys are all fixed each such a tree can be populated with the keys in a single way, and so the number of possible AVL trees for a given set of n keys is going to be given by $s(\log n)$.

We have $s(0) = s(1) = 1$ (there is a single such a tree with no nodes or one node), and $s(h) = s(h-1)^2 + 2s(h-1)s(h-2)$. This follows the definition of AVL trees: the height of the two children of a node of height h can either have both the same height $h - 1$, or their height can differ by at most one (so that the height of one is $h - 1$ and the height of the other is $h - 2$).

This is not a linear recurrence and so we have no idea how to solve this (nobody does). Asymptotically however we can consider that $s(h-1) \approx s(h-2)$ and so the rate of growth should be similar to the rate of growth of $s(h) = 3s(h-1)^2$. This we *can* solve using a range transformation: We apply log on both sides and so we have $\log s(h) = 3 + 2\log s(h-1)$, or $b_h - 2b_{h-1} = 3$, with $b_h = \log s(h)$. For the uniform part we have $r_1 = 2$ and for the nonuniform part $b = 1$ and $d = 0$, so that $r_2 = 1$. Therefore $b_h = c_1 2^h + c_2 1^h = \Theta(2^h)$. This in turn means that $s(h) = s^{b_h} = \Theta(2^{2^h})$ (a double exponential). Given now that $h = \Theta(\log n)$ we shave off one of the exponents, so that $s(n) = \Theta(2^n)$. This is how many AVL trees exist for a given set of n keys. Note in passing that this number is considerably less than the number of possible binary search trees, as it should (since AVL trees are more restricted).

For those who are still with me after all of this, $s(n)$ is the lower bound for the problem (being the size of the output). As it turns out, it is not difficult to see that out algorithm only spend a constant time for every node generated, and so its running time is also $s(n) = \Theta(2^n)$.

(b) Modify the algorithm you developed in Question 1a so that it finds *one* AVL tree for the given array of keys as efficiently as possible. Establish the correctness and analyze the running times of your algorithm.

ANSWER:

The simple answer to this question is already provided by the algorithm SOMEAVL above. The problem is that the running time of this algorithm is $\Theta(n^n/2^{n^2})$ and that is pretty horrible. The interesting question is, can be do better? In other words, can be come up with a stronger condition (the function we called PROMISING in class)?

In exploring such a possibility we first note that we are still not required to take into account the average search time. Any AVL search tree will do. We also notice that an AVL tree is "almost balanced". It would then appear that the middle of the array is always a good choice for the root, in the sense that the subsequent recursive calls will be able to construct appropriate AVL subtrees. Strangely enough, this is actually not always true! For an even n we have to round the middle, and if we round it in the same way at every step we end up with very unbalanced trees. This is dramatically obvious for $j - i + 1 = 2^k$ for some $k > 0$. However, this kid of a situation can be avoided by rounding the middle differently between recursive calls. One way or another, we are certain that the right root for an AVL tree is either $A_{\lfloor (j-i+1)/2 \rfloor}$ or $A_{\lceil (j-i+1)/2 \rceil}$. We end up with the following new and improved algorithm:

```
algorithm BETTERSOMEAVL($A_{i...j}$):
    if $i > j$ then return (NULL, 0)
    else if $i = j$ then return (NODE($A_1$), 1)
    else
        $M = \{\lfloor (j-i+1)/2 \rfloor, \lceil (j-i+1)/2 \rceil\}$
        foreach $k \in M$ do
            $(l, hl) \leftarrow$ BETTERSOMEAVL($A_{i...k-1}$)
            $(r, hr) \leftarrow$ BETTERSOMEAVL($A_{k+1...j}$)
            if $|hl - hr| \leq 1$ then return (NODE($A_{bk}, bl, br$), $1 + \max\{hl, hr\}$)
```

Note that the set of root candidates M contains either one or two value (when the number of keys is odd or even, respectively). The worst case analysis will have to assume two values in M and so the running time becomes $t(n) = 4t(n/2) + 1$ that is, $\Theta(n^2)$ (try it out for yourself, it is a pretty straightforward linear recurrence). This is one of the very few backtracking algorithms that is restricted enough so that it has a polynomial running time.

Can we do even better? We might for indeed the obvious lower bound for the problem is $\Omega(n)$ (all the keys must be processed), and I am also encouraged by the new polynomial running time.

It turns out that doing better is not even that hard. All we need is decide which if the two candidates for the root is the right choice. For this purpose we want to construct an AVL tree of height $h = \lfloor \log(j - i + 1 + 1) \rfloor$. Suppose we decide to construct a left child of height $h - 1$, so that it contains at most $2^{h-1} - 1$ nodes. To do that, we choose $\lfloor 2^{h-1} - 1 \rfloor$ nodes for the left child that is, $A_{\lfloor 2^{h-1}-1 \rfloor + 1}$ as root. The recursive calls will then do the rest. The algorithm is just like BetterSomeAVL, with the above calculations to establish the single right value for k (so that M is always a singleton). We end up with a running time given by $T(n) = 2T(n/2) + 1$ that is, linear. One cannot ask for more.

(c) Design now a backtracking algorithm that returns the optimal AVL tree for the given array of keys as quickly as possible. Establish the correctness and analyze the running times of your algorithm.

Answer:

We start with the obvious modification of SomeAVL. We now have to take the average search time into account, so instead of returning the first AVL tree, we try them all and return the one that minimizes the average search time.

```
algorithm MinAVL($A_{i...j}$, $p_{i...j}$)):
    if $i > j$ then return (Null, 0, 0)
    else if $i = j$ then return (Node($A_1$), $p_1$, 1)
    else
        $b \leftarrow \infty$
        for $k = i$ to $j$ do
            $(l, cl, hl) \leftarrow$ MinAVL($A_{i...k-1}$, $p_{i...k-1}$)
            $(r, cr, hr) \leftarrow$ MinAVL($A_{k+1...j}$, $p_{k+1...j}$)
            $c \leftarrow cl + cr + \sum_{m=i}^{j} p_m$
            if $c < b \wedge |hl - hr| \leq 1$ then
                $b \leftarrow c$
                $bk \leftarrow k$
                $bl \leftarrow l$
                $br \leftarrow r$
                $bh \leftarrow 1 + \max\{hl, hr\}$
        return (Node($A_{bk}$, $bl$, $br$), $b$, $bh$)
```

This is a bare-bones solution that is perfectly suitable for conversion into a dynamic programming algorithm.

Now on to the races to do better than this. The first standard technique is to keep the average search time of the best solution obtained so far and stop the backtracking as soon as the average search time of the current solution exceeds that.

The second technique is to prune the width of the search tree. We can do that by reconsidering the argument of choosing a possible root based on the number of nodes that can go into the children in an AVL tree. We provided an argument that choosing $A_{\lfloor 2^{h-1}-1 \rfloor+1}$ will always make an AVL tree possible, but now we are not happy with any such a tree so we must explore the other possible candidates. However, if we choose the root more than one index to the left of $A_{\lfloor 2^{h-1}-1 \rfloor+1}$ then the height of the left child will go down by one while the height of the right child will grow by one. The result cannot be an AVL tree anymore. A mirror situation happens if we choose an index more than one away from $A_{\lfloor 2^{h-1}-1 \rfloor+1}$ to the right. It follows that the only three candidates for the root are $A_{\lfloor 2^{h-1}-1 \rfloor+1}$, $A_{\lfloor 2^{h-1}-1 \rfloor}$, and $A_{\lfloor 2^{h-1}-1 \rfloor+2}$.

These two changes are both pretty straightforward to express in pseudocode so I will leave that as an exercise. I cannot think of any other way to reduce the running time and still have a backtracking algorithm (dynamic programming is further optimization, we already know that, but we cannot go there given the way the question was asked).

(d) Design a nondeterministic algorithm that receives the two arrays $A_{1\ldots n}$ and $p_{1\ldots n}$ as above and also a positive real number T. The algorithm returns True if and only if there exists an AVL tree for $A_{1\ldots n}$ whose average search time is less than or equal to T. Establish the correctness of your algorithm. Give an argument that the running time of your algorithm is polynomial.

Note in passing that there is usually no need to go into a more precise run time analysis for nondeterministic algorithms, since their purpose is to make a point (such as establish membership in $\mathcal{NP}$), not to be efficient.

Answer:

I am going to use the number guessing algorithm that we also used in class:

```
algorithm GuessNumber($n$):
    $k \leftarrow 0$
    for $i = 1$ to $\log n$ do
        $k \leftarrow 2 \times k + $ Guess
    return $k$
```

Recall that this algorithm guesses nondeterministically a number between 0 and n. It obviously runs in $\Theta(\log n)$ time.

Now for the algorithm at hand, I am going to proceed in the most straightforward way I can think of: First, I am going to nondeterministically guess an AVL tree. This is largely the same as MinAVL except that now I do not need to try all alternatives, I can just nondeterministically guess the right one.

```
algorithm GuessTree($A_{i...j}, p_{i...j}$):
    if $i > j$ then return (Null, 0, 0)
    else if $i = j$ then return (Node($A_1$), $p_1$, 1)
    else
        $k \leftarrow i +$ GuessNumber($j - i + 1$)
        $(l, cl, hl) \leftarrow$ GuessTree($A_{i...k-1}, p_{i...k-1}$)
        $(r, cr, hr) \leftarrow$ GuessTree($A_{k+1...j}, p_{k+1...j}$)
        $c \leftarrow cl + cr + \sum_{m=i}^{j} p_m$
        if $|hl - hr| \leq 1$ then return (Node($A_{bk}, bl, br$), $c$, $1 + \max\{hl, hr\}$)
```

Now all I have to do is to make sure that the average search time is below the given T:

```
algorithm GoodAVL($A_{i...j}, p_{i...j}, T$):
    $(t, c, h) \leftarrow$ GuessTree($A_{i...j}, p_{i...j}$)
    return $c \leq T$
```

Correctness is I hope obvious, the algorithm being a relatively minor variation on the overall theme. Yes, I am aware that I should have done that in a more structured manner (first guess a binary tree and then verify its properties), but that would have resulted in an algorithm that has to repeatedly traverse the tree (for a polynomial running time overall, but still. . .) and so I believe that this solution is more elegant even if it is possibly more difficult to understand. I encourage you to also try out such a standard solution.

For the running time we note that GuessNumber runs in $\log n$ time, and thus the running time of GuessTree (as well as GoodAVL) is given by $t(n) = 2t(n/2) + \log n$. With the usual change of variable $n = 2^k$ we end up with $b_k = 2b_{k-1} + k$, where $b_k = t(2^k)$. The characteristic equation of the homogeneous part is $r - 2 = 0$ and so $r_1 = 2$. For the non-homogeneous part we have $(r-1)^2 = 0$ and so $r_2 = r_3 = 1$. Therefore $b_k = c_1 2^k + c_2 1^k + c_3 k 1^k = c_1 2^k + c_3 k + c_2 = \Theta(2^k + k) = \Theta(2^k)$. Given that $k = \log n$ this in turn means that $t(n) = \Theta(2^{\log n}) = \Theta(n)$, clearly polynomial.

Now of course there is no real need to go through the whole linear recurrence thing. Indeed, it is enough to note that $t(n) = 2t(n/2) + \log n$ grows slower than $t(n) = 2t(n/2) + n$, whose solution is $\Theta(n \log n)$ (being the recurrence for MergeSort). This means that our running time is in $O(n \log n)$ (though obviously not in $\Theta(n \log n)$), which is enough to establish the desired polynomial running time. Still, this looks like a good opportunity to remind you of how cool recurrence relations are so I could not resist.

---

2. We talked in class about the complexity classes $\mathcal{P}$ and $\mathcal{NP}$ together with the concept of polynomial reductions being used to define the hardest problems in $\mathcal{NP}$ (that is, the $\mathcal{NP}$-complete problems). The concept of complete problems is not restricted to these classes, but can be defined for any pair of classes X and Y such that $X \subseteq Y$ as follows:

   - A problem π is Y-hard iff for all $\pi' \in Y$ it holds that $\pi' \trianglelefteq \pi$.
   - A problem π is Y-complete iff π is Y-hard and $\pi \in Y$.

   You will notice that the only missing piece is the definition of the $\trianglelefteq$ reduction. Define a suitable reduction to be used in the definition above. Explain carefully how your reduction is suitable for the purpose of supporting an eventual proof that either $X \subsetneq Y$ or $X = Y$ (as the case might turn out to be).

Answer:

The reduction has to come from the "inner" class X, for otherwise all the problems in Y become Y-complete and so the notion of completeness becomes utterly useless. That is, $\pi' \trianglelefteq \pi$ iff there exists an algorithm that reduces π' to π and does not require more resources than the ones required to solve the problems in X. Note that we can restrict the algorithm as much as we like within X. The more we restrict it the fewer complete problems for Y we will have, possibly up to the point of not having any such a problem at all. Generally we want as many complete problems as we can get, so further restrictions must be introduced only when strictly necessary.

Make sure you review the submission guidelines posted on the course's Web site before submitting.