Homework 5

Adam Karl

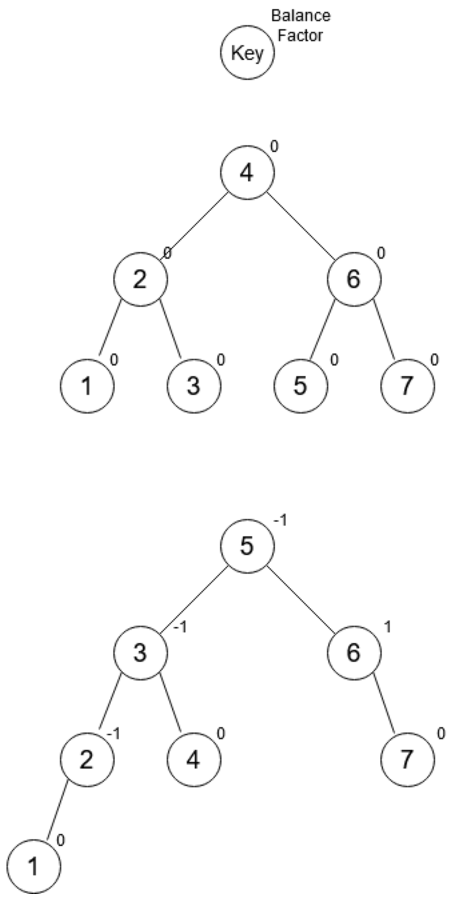
September 22, 2020

1 Problem 16 (6 points)

1.1 Motivation

It is possible for a valid AVL tree of n nodes to have multiple different heights.

- It is possible for a valid AVL tree with 7 nodes to have either height 2 or height 3. (see figure)



valid AVL trees with 7 nodes, yet different heights

- It is possible for a valid AVL tree with 14 nodes to have either height 3 or height 4.

Therefore, imagine there are 22 total nodes and we are trying to find the best solution that has node 8 as the root. There are two subtrees, one for the 7 nodes to the left of 8 and one for the 14 nodes to the right of 8.

Valid solutions

- 7 nodes => 2 height; 14 nodes => 3 height (balance factor = 1 VALID)

- 7 nodes => 3 height; 14 nodes => 3 height (balance factor = 0 VALID)
- 7 nodes => 3 height; 14 nodes => 4 height (balance factor = 1 VALID)

Invalid solutions

- 7 nodes => 2 height; 14 nodes => 4 height (balance factor = 2 INVALID)

It is NOT sufficient to simply find the optimal AVL tree for each subproblem, since it is possible that the optimal subtree for 7 nodes has a height of 2 and the optimal subtree for 14 nodes has a height of 4. In this case, even though the left subtree is a valid AVL tree and the right subtree is a valid AVL tree, combining them would give the root node an invalid balance factor of 2, thus creating an invalid overall AVL tree.

Thus, when calculating optimal subtrees/subproblems, we must calculate the optimal solutions FOR EACH POSSIBLE HEIGHT. Then, we are able to only combine 2 subtrees when their heights differ by only 0 or 1.

1.2 Array Setup and Visualization

Create an n by n 2D matrix. In A[i, j] we will be storing information about valid AVL tree(s) for nodes i through j (inclusive).

The solutions in each element in the array will be tuples of the form (tree height, expected depth of a key). This tuple, and the reason there may be several solutions per array element will make more sense as we get into the finer details of the algorithm.

Initialization: We start by filling in solutions along the i=j diagonal, then build up since each A[i,j] element depends on elements in the row to the left of A[i,j], and the elements in the same column below A[i,j].

(1) For i from 1 to n:

    (I) set A[i,j] = (0, 0) //"tree" of a single node, so: (height=0, expected depth = 0)

Initialized table:

	1	2	3	4	...	n
1	(0,0)					
2		(0,0)				
3			(0,0)			
4				(0,0)		
...						
n						(0,0)

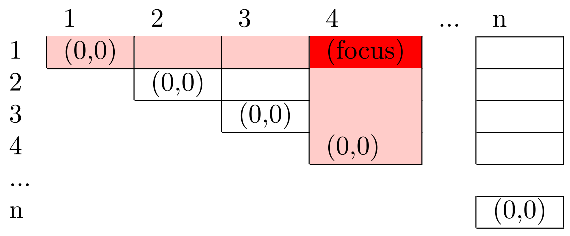
Imagine we are trying to find the solution to put in A[1,4]. The intuition is the AVL subtree that contains nodes 1 through 4, and minimizes the expected depth of a key. We must consider each possible node as the root, and the left and right subproblems choosing this root depends on:

- root = 1
  - left subtree: n/a
  - right subtree: A[2,4]
- root = 2
  - left subtree: A[1,1]
  - right subtree: A[3,4]
- root = 3
  - left subtree: A[1,2]
  - right subtree: A[4,4]
- root = 4

- left subtree: A[1,3]
- right subtree: n/a

Note that nodes 1 and 4 cannot be the root, as the resulting tree would be too lopsided to be an AVL tree (they would have balance factors of 2 and -2, respectively). Since the heights of the subproblems would have a difference greater than one, our algorithm will not find any viable solutions for these possible root nodes.

Here I have highlighted the target A[1,4] and each element in the array that A[1,4] is dependent on:



As you can see, each element in A is dependent on the solutions in the column below it, and in the row to the left of it.

We will traverse the array so that when we try to fill in solutions for an element, each of its dependents will already be completed. Left to right, bottom to top.

1.3 Iterative, Array-based, bottom-up algorithm

```
(1) for i from n-1 to 1:
    (I) for j from i+1 to n: // we will be finding solutions for A[i,j] (subtree of nodes i to j)
        (A) for pivot from i to j:
            (i) // the pivot may be any node i through j (although many outer nodes may not
                make valid AVL trees, and i/j can ONLY be the pivot for a subtree of exactly
                2 nodes)
            (ii) for each tuple in A[i, pivot-1] (if any):
                (a) // this is the "left tuple," with elements "leftHeight" and "leftExpected-
                    Depth"
                (b) for each tuple in A[pivot+1, j] (if any):
                    (1) this is the "right tuple," with elements "rightHeight" and "rightExpected-
                        Depth"
                    (2) if |rightHeight - leftHeight| $\leq$ 1:
                        (I) // ensures combining these two sub-trees creates a valid AVL tree
                        (II) testHeight = 1 + max(leftHeight, rightHeight)
                        (III) testExpectedDepth = leftExpectedDepth + SUM($p_i$ through $p_{pivot-1}$))
                              + rightExpectedDepth + SUM($p_{pivot+1}$ through $p_j$))
                        (IV) // SUM($p_i$ through $p_{pivot-1}$)) is the "cost" to the expected node depth
                             due to moving the left subtree down by 1 depth.
                        (V) // SUM($p_{pivot+1}$ through $p_j$) is the "cost" to the expected node depth
                            due to moving the right subtree down by 1 depth.
                        (VI) if A[i,j] does not contain a tuple with height = testHeight, add tuple
                             (testHeigh, testExpectedDepth) to A[i,j]
                        (VII) else compare testExpectedDepth to the expected depth of that tuple. if
                              testExpectedDepth is smaller, replace the tuple in A[i,j] with (testHeight,
                              testExpectedDepth)
```

Solution

(1) find the minimum expected depth of all tuple(s) in A[1,n]. This is the minimum expected depth for a valid AVL tree of all nodes 1 through n.

Implementation note: for this algorithm I imagine each element of A pointing to a linked list of the (height, minimumExpectedDepth) tuples. A linked list is obviously not perfect, but it's an easily implemented way to allow us to keep track of the best valid AVL solutions for each possible height of the subtree.

It may seem strange that even for say, 15 nodes we will consider every node 1 through 15 to be a possible root. However, when we choose many of these roots (say, node 2), the heights for the left subtree and right subtree will never be within 1 of each other. Therefore, they cannot combine into a valid AVL tree (the balance factor of the node would not be -1, 0, or 1) and no solutions for these root nodes will be added to A[i,j]. This check happens at the step labeled (2) in the algorithm

1.4 Potential Changes and Improvements

We could instead convert 2D array with an unknown number of tuples in each element to a 3D array with a value. We would do this by creating a third dimension for height, where A[i, j, h] would contain the optimal expected depth of a key for the optimal AVL subtree containing trees i through j (inclusive). At the end of building the array we would check the row of A[1, N, h] over all values of h for the optimal solution. The reason I dislike this solution (and the reason I didn't implement it for my solution) is that it creates a LOT of NULL or empty elements in the 3D array, which seems wasteful of space. For instance, obviously A[1, 7, 1] would be impossible, since no AVL tree with 7 elements could have a height of 1. In fact, since an AVL tree with 7 nodes can ONLY have a height of 2 or 3, ALL other values for h other than 2 or 3 would have an empty cell at A[1, 7, h]. Note that this solution IS strictly faster than my solution (since you don't have to traverse a linked list of tuples for each array element), but it would be a tradeoff with wasted space that grows exponentially as the possible AVL tree heights grow.

When n is small, the possible AVL trees only have a couple different possible heights, so traversing a linked list with 1 entry for each height is negligible. When n is large, this change would save time, but at the cost of creating a massive (and almost entirely empty) array that wastes a ton of space.

Additionally, I initially created an array where the solution for each subtree from nodes i to j also included the sum of probabilities p_i through p_j. This made the solution faster as the "cost" to moving the subtree down by 1 depth equates to the previous expected value + this sum of probabilities (no longer had to constantly do O(n) probability additions within the critical loop). However, it had the significant drawback in how easily the algorithm was understood, as now there's 3 things each subtree solution had to keep track of (height, expected depth, sum of probabilities), and the motivation for the third value can be re-calculated at any time (it's not specific to that subtree but to all subtrees of nodes i through j). However, understand-ability aside I do believe this modification makes the algorithm strictly better (it reduces the runtime by a factor of n), although even without it is undoubtedly polytime.