CS1510 Dynamic Programming Problems 10 & 11

Rebecca Negley, Sean Myers

September 28, 2011

10. The input for this problem consists of n keys $K_1, ..., K_n$ with $K_1 < K_2 < ..., K_n$ and associated probabilities $p_1, ..., p_n$. The problem is to find the AVL tree for these keys that minimizes the expected depth of a key. An AVL tree is a binary search tree with the property that every node has balance factor -1, 0, or 1. Give a polynomial time algorithm for this problem.

    Solution: This problem is equivalent to the BST problem we did in class, except with a new caveat of the balance factor. In the BST, we checked every node from an arbitrary left endpoint, to a right endpoint, and called that node i. i will have a left subtree of left to $i-1$, and the right subtree will have a subtree of $i+1$ to right. In a BST, the left and right subtrees were indepenedent and so naive recursion was fine. All you had to do was calculate the minimum weight of each subtree, and send that up. At the root, you would have to add the sum of all the weights of both subtrees, and the weight of itself, because the weight is $Depth * P_n$ and when you add a new root, you add a layer of depth, so you need to add all the nodes from left to right again. The formula would look something like this:

    $$BST(left, i-1) + BST(i+1, right) + \sum_{i=left}^{right} D * p_i$$

    The problem is with AVL, the balance factor makes it so that the left and right subtrees are no longer independent. This makes it so that naive recursion fails for AVL. This means that in each recursive call, we need more than just the optimal weight coming back up the call chain. We also need the height. But if you have only the height of the minimum weight of a subtree, the recursive algorithm is still insufficient. It will only find the local optimal solution, and send that up. What needs to be sent up is a collection C of feasible weight/height combinations so that the root can determine all possible venues. An example to prove this:

    Suppose in a left subtree, you find two feasible weights: 15 and 16, with heights 7 and 6, respectively. In the right subtree you find one feasible weight of 10 with a height of 5. If we were only to send the local optimum of both, the root would only receive a [15,7] and [10,5] pairing. Of course, it couldn't be an AVL because the height is of difference 2. If a collection is sent up, then it would be able to pair [15,6] and [10,5], which could be the optimal solution.

    So now that we know that a collection must be sent up, we can now create our recursive algorithm:

    The base case is if the right and left are equal, which only occurs at a leaf node. Only send up one value, which is the probability at that node, and a height of 1 in a collection.

    Also, at each root, we must add all possible sets from the left and right subtrees. It can only be a feasible solution if the left and right nodes are between -1 and 1. so:

    ```
    minAVL(left,right):
    if(left = right):
      return set([probability[left], 1])
    ```

```
else:
  Set allPossibilities
  for(j = left to right):
      allLeft = minAVL(left, j-1) //This is all feasible solutions from the left subtree
      allRight = minAVL(j+1, right)//same for the right.
      for(allLeft as [leftW, lHeight]): //creates two variables for every item in the set
         for(allRight as [rightW, rHeight]):
            if(lHeight − rHeight ≤ 1 AND lHeight − rHeight ≥ −1):
              minWeight = leftW + rightW + sum of keys left to right
              height = max(lHeight, rHeight)+ 1
              allPosibilites.add([minWeight, height]);
  return allPossibilities
```

The thing returned from the entire function will be a set of all possible weights and heights. If a person wanted to know the minimum tree weight, they would only have to do a min of the entire set.

So now that we have all this, it is just a simple mechanical process transfering it to a dynamic algorithm.

First we need to create a 2d-array called AVL[n][n]. This array is going to hold a set for each index in the array.

Then, we need to initialize the array, going in a diagonal fashion, and setting the base case up:

for(i = 0 to n): AVL[i][i] = Set([probability[i], 1])

For this array-based solution to work, we must build the array from the bottom up and from the left to the right (the left being where the diagonal base case is, not 0). Then to get the answer, look at the minimum in the set at the top right index of the array.

The final solution looks like this:

```
minAVL(n):
      for(i = 0 to n):
          AVL[i][i] = Set([probability[i], 1])
      for(i = n to 0):
          for(j = i+1 to n):
              left = i, right = j
              for(k = left to right):
                  allLeft = AVL[left, k-1]
                  allRight = AVL[k+1, right]
                  for(allLeft as [leftW, lHeight]):
                     for(allRight as [rightW, rHeight]):
                        if(lHeight − rHeight ≤ 1 AND lHeight − rHeight ≥ −1):
                        minWeight = leftW + rightW + sum of keys left to right
                        height = max(lHeight, rHeight)+ 1
                        AVL[left,right].add([minWeight, height]);
      return min(AVL[0][n])
```

11. The input consists of n intervals over the real line. The output should be a collection C of non-overlapping intervals such the sum of the lengths of the intervals in C is maximized. Give a polynomial time algorithm for this problem.

    Solution: We first examine the naive recursion approach. For any set of n intervals over the real

line, order the intervals I_1, I_2, ..., I_n in order of increasing right endpoint. (That is, I_n is the interval that ends last.) Then, we can consider I_n. If we know the collection C_{n-1} of non-overlapping intervals with maximum length over I_1, ...,I_{n-1}, we could then (naively) do the following: if I_n does not overlap any intervals in C_{n-1}, add I_n to C_{n-1} to create C_n. If I_n does overlap some interval in C_{n-1}, $C_n = C_{n-1}$. However, if there is an overlap, it could be optimal to include I_n instead of any previous intervals that overlap it. Therefore, we need to know C_{n-1} and $C2_n$ where $C2_k$ is the collection of non-overlapping intervals I_j with maximum length such that $j < k$ and I_j does not overlap I_k. Let p be the maximum index such that $p < k$ and I_p does not overlap I_k. Then, $C2_k$ as defined above is just the maximum non-overlapping collection of the first p intervals (which must all end before I_k starts by the way we constructed our indices). Hence, $C2_k = C_p$ by the way we constructed our indices. The recursion approach would then look like:

```
MNOL(n)
    if(n=1)
        return {I_1}
    else
        return max(MNOL(n-1),MNOL(p)∪{I_n}) //p as defined in paragraph above
```

We can turn this algorithm into a much more efficient dynamic programming algorithm. Simply create a size $n+1$ array and work from $k = 1$ to n filling in the array from the non-base case in the recursive code. We can easily find p in $O(n)$ time, so the entire algorithm is $O(n^2)$:

```
MNOL(n)
    Mnol[0]=∅
    Mnol[1] = {I_1}
    for i from 2 to n
        p=i-1
        while p ≠ 0
            if I_p∩I_i = ∅
                break
            p − −
        Mnol[i]=max(Mnol[i-1],Mnol[p]∪{I_i}) //p is max int s.t. p < k and I_p ∩ I_k = ∅
    return Mnol[n]
```

*Note: The "max" function above returns the set with the max total length of intervals.