Homework 7

Robbie McKinstry, Jack McQuown, Cyrus Ramavarapu

19 September 2016

Problem 8:

Given an arbitrary tree $\mathcal{T}$ with both positive and negative edge weights, finding the maximally weighted simple path, $\mathcal{P}$ can be found recursively by considering both the best linear path and the best kinked path available at a given vertex v.

Although this seems to suggest that naively recursing to the immediate subproblem, in this case a sub tree $\mathcal{T}'$ of $\mathcal{T}$, and extracting the two required values is sufficient, such thinking actually proves to be problematic. This is because although a path may not be the immediate best choice, its inclusion into the solution may prove beneficial at a later stage because it leads to an overall greater edge weight.

Accounting for the possibility that a path may not be immediately optimal, gives rise to the following recursive algorithm that considers both the linear and kinked paths at a given vertex.

```
Function: $MWSP$
Input: $Node\ \mathcal{N}$
if $\mathcal{N}$ has no children then
    $return\ 0$
else
    $linear\ 1,\ linear\ 2 = max\_2(\forall\ children\ i:\ MWSP(i) + w_{parent,i})$
    $best\ linear = max(linear\ 1, linear\ 2)$
    $L.append(best\ linear)$
    $kinked\ path = linear\ 1 + linear\ 2$
    $K.append(kinked\ path)$
    Return: $L,\ K$
```

The maximum element from $L,\ K$ can be taken. If this maximum value is less than 0, the maximum weighted simple path can be taken to be 0 since not not moving from any vertex will be optimal.

Although the recursive algorithm produces the maximum weight of a simple path on a tree, an interative process can be developed that will run in linear time by starting with the leaves and moving up the tree while keeping track of the kinked and linear paths at every vertex.

```
Function: $IT\ MWSP$
Input: $Tree\ \mathcal{T}$
Globals: $A\ [\ ][\ ]$
$j = 0$
foreach $Leaf\ Node\ v$ do
    $A\ [v][0] = 0$
    $A\ [v][1] = 0$
    $j{+}{+}$

/* Assume the nodes are numbered from the leaves upward */
for $j\ to\ n$ do
    $linear\ 1,\ linear\ 2 = max\_2(\forall\ children\ i: A[j][i] + w_{j,i})$
    $A\ [j][0] = max(linear\ 1,\ linear\ 2)$
    $A\ [j][1] = linear\ 1 + linear\ 2$
```

As with the recursive algorithm, the maximum value between the two rows of A will be the answer, unless this value is negative. In this case the answer will be 0 for the reason mentioned above.

Problem 9:

To find the minimum AVL tree, first note certain properties of the tree:

1. In the general case, any key could serve as the root of the tree. This is obvious, because K_1 could be used in the case where the tree has only 2 keys (this is also true for K_n). Internal nodes can of course be used to create an AVL tree (obvious from looking at most examples of AVL trees).

2. Because the problem of minimizing the expected depth of the keys is reducible to weighing each key and minimizing the weights, the whole problem is reducible to finding the best OBST that is also an AVL tree. (*Introduction to Algorithms* by CLRS explains this in more detail). This means we can apply our same weight calculation tricks from the OBST problem to this problem as well.

Our approach is to find all AVL trees, and then finally take the minimum-valued AVL tree from that set. It operates in cubic space complexity and polynomial time complexity.

We summarize our approach as follows, before giving a specification of the algorithm in more detail:

Because each key can serve as the root of the AVL tree, we iterate over all of the keys, selecting each to serve as that root. Call the selected key K_i. This creates two partitions of the remaining keys in the set: $K[1 \ldots i]$ and $K[i + 1 \ldots n]$. Recursively calculate all possible AVL trees for each of those sets, storing only the most optimal AVL tree with height h for all h (i.e. do not store duplicate trees. You need not store two trees with the same height but different weights for a given set of keys, instead only the most efficient of those two trees).

Now, we need to recombine the left and right subtrees with our root node such that the whole subtree is still an AVL tree. Currently, we have stored an array (indexed by height) of weights of the possible AVL trees created on the left, and the same on the right. The recombination is the Cartesian product of the right and left sets, then filtered by removing any ordered pairs that differ in height by more than 1. This can be done in linear time by looping over the left array for $j = 1$ to i, and looking in the right array at indices $j-1, j, j+1$, and selecting those non-empty array elements and summing their weights. Finally, you can return the array of weights indexed by height to the caller, and when the call stack is empty, simply take the tree with the minimum weight. Note the degenerate case for this algorithm is when the input is of length 1; simply populate the array with the single weight.

Now, we formalize our iterative specification of the algorithm below.

Algorithm 1: Optimal AVL Tree

```
Data: Array $A[n][n][n]$ where $n$ is the size of the input
Input: $K_1 \dots K_n$ and $W_1 \dots W_n$
Output: The minimum of $A[n][n]$
for $i \leftarrow 1$ to $n$ do
    $A[i][i] = [W_i]$
begin
    for $start \leftarrow 1$ to $n$ do
        for $end \leftarrow start$ to $1$ do
            for $i \leftarrow start$ to $end$ do
                Let $left = A[start][i]$ Let $right = A[i+1][end]$
                /* Recombine the left and right subtrees,
                   which are AVL trees with the selected root.
                   */
                foreach $height, weight \leftarrow left$ do
                    /* The combined subtree can only be an AVL
                       tree if the heights differ by 1 or less.
                       */
                    if $right[height]! = empty$ then
                        /* If the subtree has been cached by a
                           different loop iteration,           */
                        /* then take whichever tree is least
                           weighted                            */
                        $A[start][end][height+1] = min(left[height] +$
                        $right[height], A[start][end][height+1])$
                    if $right[height+1]! = empty$ then
                        $A[start][end][height+2] = min(left[height] +$
                        $right[height + 1], A[start][end][height+2])$
                    if $right[height-1]! = empty$ then
                        $A[start][end][height] = min(left[height] +$
                        $right[height-1], A[start][end][height])$
```