CS1510 Dynamic Programming Problems 8 & 9

Rebecca Negley, Sean Myers

September 26, 2011

8. The input to this problem is a sequence S of integers (not necessarily positive). The problem is to find the consecutive subsequence of S with maximum sum. "Consecutive" means that you are not allowed to skip numbers. For example if the input was

   12,-14,1, 23, -6, 22, -34, 13

   the output would be 1,23,-6,22. Give a linear algorithm for this problem.

   Solution: First, let's create a recursive algorithm to this problem and then reconstruct our dynamic algorithm around that. A naive recursive algorithm would start at the end integer. It would decide whether or not to add itself to the subproblem + the integer or not to if the integer was negative. This solution is horrible because we do not have enough information being passed up from the recursive algorithm to know if we can even do that. For example, the smaller subproblem might not be in the max sum of consecutive integers, so it would be impossible for the current element to be in it either (since that would contradict the definition of consecutive numbers). So we need to add that information into the algorithm. In order to do that, we need two pieces from the recursive calls returning: The max subproblem and a set of consecutive integers that is currently being built. Let's see what this would look like:

   If there is only 1 element, then the sum of both the current consecutive set and the max set is going to be the integer of that element. In pseudocode: if(n==1) return [n,n] the first n being the max set, and the second being the current set being worked on. That is the base case, now we need the recursive element. If the current set being worked on added with the current integer is greater than the max set, change the max set to the current set being built, or in pseudocode(Kind of a mix of python and how matlab deals with returning matrices): [maxSet, currentSet] = maxSeq(n-1); if(sequence[n] + currentSet > maxSet): return [currentSet + sequence[n], currentSet + sequence[n]];

   Now if the two previous sets are equal, the above condition will also apply to them, so we need not worry about that condition. Another condition is if the currentSet is + the sequence is less than just the sequence, then we change currentSet to just the sequence. For example, in the example illustrated above, the currentSet and maxSet returned from maxSeq(2) would be [12, -2] since the maxSet would be highest at 12 and the currentSet is 12 + -14, but maxSeq(3), currentSet will be 1 since currentSet is -2 and sequence is 1, the addition of them both is -1, but just the sequence alone is 1. In pseudo code that is:

```
if(sequence[n]+ currentSet < sequence[n]):
    if(sequence[n] > maxSet)
      return [sequence[n], sequence[n];
    else
      return [maxSet, sequence[n]]
```

I added the extra possibility that the sequence was also larger than the current maxSet. For example, a set: 9, -14, 19. on call maxSeq(3), the maxSet is greater than both the currentSet and the maxSet.

The last case we need to worry about is if currentSet is not greater than the maxSet, but is still greater than the sequence[n]. If this occurs, just add sequence[n] to the currentSet but do not change maxSet, so:

else: return[maxSet, sequence[n]+currentSet]

The final recursive algorithm looks like this:

```
maxSeq(n):
    if(n==1): return [sequence[n], sequence[n]
    [maxSet, currentSet] = maxSeq(n-1);
    else if(sequence[n]+ currentSet > maxSet): return [currentSet+sequence[n], currentSet+sequence[n]]
    else if(sequence[n]+ currentSet < sequence[n]):
        if(sequence[n] > maxSet)
          return [sequence[n], sequence[n]]
        else
          return [maxSet, sequence[n]]
    else: return [maxSet, sequence[n]+currentSet]
```

This algorithm, through the mechanical means we have seen in class then turns into this:

```
maxSeq(n):
    maxSet[n];
    currentSet[n];
    maxSet[1] = sequence[1] //Assuming array starts at 1.
    currentSet[1] = sequence[1]
    for(i = 2 to n):
        if(sequence[i] + currentSet[i-1] < sequence[i]):
            currentSet[i]= currentSet[i-1] + sequence[i]
            maxSeq[i] = currentSet[i-1] + sequence[i]
        else if(sequence[i]+ currentSet[i-1] < sequence[i]):
            if(sequence[i] > maxSet[i-1)
              currentSet[i] = sequence[i]
              maxSet[i] = sequence[i]
            else
              maxSet[i] = maxSet[i-1]
              currentSet[i] = sequence[i]
        else:
            maxSet[i] = maxSet[i-1]
            currentSet[i] = sequence[i]+ currentSet[i-1]
```

9. The input to this problem is a tree T with integer weights on the edges. The weights may be negative, zero, or positive. Give a linear time algorithm to find the shortest simple path in T. The length of a path is the sum of the weights of the edges in the path. A path is simple if no vertex is repeated. Note that the endpoints of the path are unconstrained.

   Solution: Consider the following SSP recursive algorithm. Examine the root of the tree. If the tree is empty (null), then return 0. Otherwise, we have three options. The shortest simple path could be within

one of the subtrees and not connect to the root, in one of the subtrees and connect to the root, or extend into both subtrees (hence, connecting to the root from both sides). Note that if the shortest simple path is empty, SSP will return 0 for each subtree. Also, if a subtree connects to the root, it must use the shortest simple path that has an endpoint at its root. This allows us to connect from the root of the subtree to the root of T. Let $SSP(T)$ be the shortest simple path in T and $SSP2(T)$ be the shortest simple path in T ending at the root. Then, we return $min\{SSP$(left subtree), SSP(right subtree), $SSP2$(left subtree) + $weight$(edge connecting root to left subtree), $SSP2$(right subtree) + $weight$(edge connecting root to right subtree), $SSP2$(left subtree) + $weight$(edge connecting root to left subtree) + $weight$(edge connecting root to right subtree) + $SSP2$(right subtree)$\}$. Our recursive algorithm would look like:

SSP(T)
    if T= $NULL$
        return 0
    else
        return $min\{SSP(leftT), SSP(rightT), SSP2(leftT) + weight$(edge connecting $leftT$ to root), $SSP2(rightT) + weight$(edge connecting $rightT$ to root), $SSP2(leftT)+$ $weight$(edge connecting $leftT$ to root) + $SSP2(rightT) + weight$(edge connecting $rightT$ to root)$\}$ //$leftT$ =left subtree, $rightT$ =right subtree

SSP2(T)
    if $T = NULL$
        return 0
    else
        return $min\{SSP2(leftT) + weight$(edge connecting $leftT$ to root), $SSP2(rightT)$ $+weight$(edge connecting $rightT$ to root)$\}$

Observe that we call $SSP2(leftT)$ and $SSP2(rightT)$ twice in every non-base case call to $SSP(T)$. To make this more efficient, we will rewrite the solution as a dynamic programming algorithm. Label the nodes in the tree $v_1,\ldots,v_n$ from the deepest nodes to the root. Create two arrays, SSP and $SSP2$ of size $(n + 1)$. The ith index corresponds to the subtree with v_i as its root. $i = 0$ indicates a null tree. Initialize the arrays with the base cases from above, so $SSP[0] = 0$ and $SSP2[0] = 0$. Then, run through a loop to fill in the rest of the arrays using the non-base cases above. The dynamic programming algorithm will then look like:

ShortestPath($v_1,\ldots,v_n$)
    $SSP[n + 1]$. $SSP2[n + 1]$
    $SSP[0] = 0$
    $SSP2[0] = 0$
    for i from 1 to n
        $SSP[i] = min\{SSP[j],\ SSP[k]\ , SSP2[j] + w_{i,j},\ SSP2[k] + w_{i,k},\ SSP2[j] + w_{i,j} + SSP2[k]$ $+w_{i,k}\}$
            //v_j is the left child of v_i, v_k is the right child of v_i
            //by definition, $j, k < i$
            //$w_{p,q}$ is the weight of the edge connecting v_p and v_q
            //if $p = 0$, $w_{i,p} = 0$
        $SSP2[i] = min\{SSP2[j] + w_{i,j},\ SSP2[k] + w_{i,k}\}$
    return $SSP[n]$

Then, we have the value of the shortest simple path. We can find the path by working backwards through the array and determining which of the subtrees were used.