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 last integer n. It would decide whether or not to add n to the consecutive subsequence with maximum sum of the first $n-1$ integers. However, if it had only the maximum consecutive subsequence, it would not have enough information to conclude the new maximum consecutive subsequence. If the consecutive subsequence returned from the first $n-1$ integers stops before $n-1$st integer, a separate subsequence (that ends at the end of the sequence and is not maximal for the subproblem with the first $n-1$ integers) may become the maximum when the integer at position n is added to it. Therefore, we need two pieces of information from the recursive call at position k: the maximum subsequence of the first $k-1$ integers and the set of consecutive integers with maximum sum that ends at $k-1$ or is empty. We will call this second sequence the max current subsequence.

   If $k = 1$ and the integer at the first position is non-negative, then the maximum consecutive subsequence and the max current subsequence are both going to include only the integer at the first position. Otherwise, if $k = 1$ and the first integer is negative, the max current subsequence and the max consecutive subsequence will be empty. Then, at step k, if the integer at position k + the sum of the max current subsequence for $k-1$ is greater than the sum of the maximum consecutive subsequence for the first $k-1$ integers, return the max current subsequence for $k-1$ + integer at k. Otherwise, return the max consecutive subsequence for the first $k-1$ integers.

   We also must update the max current subsequence. If the sum of the max current subsequence for the first $k-1$ integers + the integer at k is non-negative, return the max current subsequence + integer at k. Otherwise, return an empty set. In pseudo-code, this recursive algorithm looks like:

```
maxSeq(n):
    if n=1
        if sequence[n]$\geq$0
            return sequence[n]
        else
            return $\emptyset$
    else
        if maxCur(n-1)+sequence[n] > maxSeq(n-1)
```

```
            return maxCur(n-1)+sequence[n]
        else
            return maxSeq(n-1)

maxCur(n):
    if n=1
        if sequence[n]$\geq$0
            return sequence[n]
        else
            return $\emptyset$
    else
        if(maxCur(n-1)+sequence[n]$\geq$0
            return maxCur(n-1)+sequence[n]
        else
            return $\emptyset$
```

This algorithm, though, makes several identical recursive calls. To improve upon its efficiency, we turn it into a dynamic programming algorithm. We create two arrays of size n and use the base cases above as initialization. We then iterate through increasing indices to fill the arrays from the bottom up:

```
maxSeq(n):
    maxSeq[n]; maxCur[n];
    if(sequence[1]$\geq$0): //Assuming array starts at 1
        maxSeq[1] = sequence[1]
        maxCur[1] = sequence[1]
    else:
        maxSeq[1] = 0
        maxCur[1] = 0
    for(i=2 to n):
        if(maxCur[i-1]+sequence[i]>maxSeq[i-1]):
            maxSeq[i] = maxCur[i-1]+sequence[i]
            maxCur[i] = maxCur[i-1]+sequence[i]
        else:
            maxSeq[i] = maxSeq[i-1]
            if(maxCur[i-1]+sequence[i]$\geq$0):
                maxCur[i] = maxCur[i-1]+sequence[i]
            else:
                maxCur[i] = 0
    return maxSeq[n]
```

We then have the maximum sum from maxSeq[n] and can trace back through the arrays to find the consecutive sequence it describes.

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 arrays and determining which of the subtrees were used.