CS 6363.003 Spring 2021 Official Solutions

Homework 3 Problem 1

Describe and analyze a dynamic programming algorithm to compute, given a list of integers separated by $+$ and $-$ signs, the maximum possible value the expression can take by adding parentheses.

Solution: We'll assume the input is given as two arrays $A[1\,..\,n]$ and $O[1\,..\,n-1]$ where $A[i]$ is the ith integer of the input sequence and $O[i]$ is the operator immediately following the ith integer.

Observe how different placements of the parentheses tell us what order to evaluate the individual operators. If we know the final operator evaluation for the optimal placement of parentheses, then all that remains is to figure out what the best expressions are for all the integers to the left and right of that operator. If the operator is a $+$, then we would want to maximize both the left and right expressions. Otherwise, we would want to maximize the left expression, but minimize the right expression so we subtract off as little as possible (or even subtract off as low a negative number as possible!). So it appears we want to tell the Recursion Fairy whether we want a maximum or minimum value expression along with the subsequence of integers and their operators that should be optimized over. In forming these subsequences, we always remove some integers from the left or right sides of a contiguous subsequence, making the recursive subsequences contiguous as well. It will suffice to only tell the Recursion Fairy where our contiguous subsequences begin and end (just like with optimal binary search trees).

Let $MaxExpr(i,j)$ for any $1 \le i \le j \le n$ denote the maximum possible value of any expression obtainable by placing parentheses in various positions around integers i through j and the operators separating them. Define $MinExpr(i,j)$ similarly. If $i = j$, then the only value possible for either $MaxExpr(i,j)$ or $MinExpr(i,j)$ is $A[i]$ itself. Otherwise, let the kth integer be the rightmost one immediately before the last operation performed using the optimal placement of the parentheses. We have $i \le k < j$. Per the above discussion, $MaxExpr(i,j) = MaxExpr(i,k) + MaxExpr(k+1,j)$ if the best last operation is a $+$, and $MaxExpr(i,j) = MaxExpr(i,k) - MinExpr(k+1,j)$ otherwise. Similarly, $MinExpr(i,j) = MinExpr(i,k) + MinExpr(k+1,j)$ if the best last operation is a $+$, and $MinExpr(i,j) = MinExpr(i,k) - MaxExpr(k+1,j)$ otherwise. Of course, we don't know k in advance, so we'll have to maximize or minimize over all possible values to obtain $MaxExpr(i,j)$ and $MinExpr(i,j)$, respectively. We conclude with the following recursive definitions:

$$MaxExpr(i,j) = \begin{cases} A[i] & \text{if } i = j \\ \max_{i \le k < j} \begin{cases} MaxExpr(i,k) + MaxExpr(k+1,j) & \text{if } O[k] = \text{‘+’} \\ MaxExpr(i,k) - MinExpr(k+1,j) & \text{otherwise} \end{cases} & \text{otherwise} \end{cases}$$

$$MinExpr(i,j) = \begin{cases} A[i] & \text{if } i = j \\ \min_{i \le k < j} \begin{cases} MinExpr(i,k) + MinExpr(k+1,j) & \text{if } O[k] = \text{‘+’} \\ MinExpr(i,k) - MaxExpr(k+1,j) & \text{otherwise} \end{cases} & \text{otherwise} \end{cases}$$

By definition, we want to compute $MaxExpr(1,n)$. The possible values of i and j as subproblem parameters suggest we should store subproblem solutions in two 2D arrays $MaxExpr[1\,..\,n, 1\,..\,n]$

and $MinExpr[1 .. n, 1 .. n]$. Every subproblem depends upon other subproblems for which either the first parameter is higher or the second parameter is lower, so we'll fill the arrays in decreasing order of first index and increasing order of second index. For each i, j, we can fill either $MaxExpr[i, j]$ or $MinExpr[i, j]$ first. It takes $O(n)$ time to solve each of the $O(n^2)$ subproblems, ***so the total running time will be $O(n^3)$***. Here's a pseudocode implementation.

```
MAXVALUEEXPRESSION($A[1 .. n], O[1 .. n-1]$):
  for $i \leftarrow n$ to 1
      $MaxExpr[i,i] \leftarrow A[i]$
      $MinExpr[i,i] \leftarrow A[i]$
      for $j \leftarrow i+1$ to $n$
          $MaxExpr[i,j] \leftarrow -\infty$
          $MinExpr[i,j] \leftarrow \infty$
          for $k \leftarrow i$ to $j-1$
              if $O[k] =$ '+'
                  $maxForOp \leftarrow MaxExpr[i,k] + MaxExpr[k+1,j]$
                  $minForOp \leftarrow MinExpr[i,k] + MinExpr[k+1,j]$
              else
                  $maxForOp \leftarrow MaxExpr[i,k] - MinExpr[k+1,j]$
                  $minForOp \leftarrow MinExpr[i,k] - MaxExpr[k+1,j]$
              if $MaxExpr[i,j] < maxForOp$
                  $MaxExpr[i,j] \leftarrow maxForOp$
              if $MinExpr[i,j] > minForOp$
                  $MinExpr[i,j] \leftarrow minForOp$
  return $MaxExpr[1,n]$
```

■

Rubric: 10 points total: 5 points total for the recurrence; −1 for no justification, −2 for missing the base cases; 3 points for filling the table (0 if the recurrence is very wrong); 2 points for the running time analysis.

+3 extra credit points for a correct *and justified* $O(n^2)$ time algorithm that does not depend on outside sources. (A likely solution came up during office hours. It's fine if they acknowledge others but appear to have come up with the main ideas themselves.)

CS 6363.003 Spring 2021 Official Solutions
Homework 3 Problem 2

Let T be a rooted binary tree with n vertices, and let $k \leq n$ be a positive integer. We would like to mark k vertices in T so that every vertex has a nearby marked ancestor. More formally, we define the ***clustering cost*** of any subset K of vertices as

$$cost(K) = \max_{v} cost(v, K),$$

where the maximum is taken over all vertices v in the tree, and $cost(v, K)$ is the distance from v to its nearest ancestor in K.

(a) Describe a dynamic programming algorithm to compute, given the tree T and an integer r, the size of the smallest subset of vertices whose clustering cost is at most r. For full credit, your algorithm should run in $O(nr)$ time.

Solution: Similar to the maximum independent set problem, we'd like to decide whether a vertex v belongs to an optimal subset K by recursively learning the consequences of our choices on the subtrees of v's children. Suppose we've guessed which strict ancestors of v belong to K, and then we guess $v \notin K$. We want to know the smallest subsets we can get away with for the subtrees of v's children. For each child w of v, we'll have $cost(w, K) = cost(v, K) + 1$. We cannot allow those costs to go above r, so it seems we must let the Recursion Fairy know the distance between v and its nearest ancestor in K.

For any vertex v in T, let V_v denote the vertices in v's subtree. Let $SCS(v, d)$ denote the size of the smallest possible subset $K_v = K \cap V_v$ such that the clustering cost of K is at most r and the distance between v and its nearest *strict* ancestor in K is at least d.

Suppose a smallest such subset K_v contains v. Each child w of v lies distance exactly 1 from its nearest strict ancestor. Therefore, the portion of K_v in w's subtree would have size $SCS(w, 1)$. Let $w \downarrow v$ mean "w is a child of v". We have $SCS(v, d) = 1 + \sum_{w \downarrow v} SCS(w, 1)$. However, if $v \notin K_v$, then each child w is one edge further away from their nearest strict ancestor in K than v is, so the portion of K_v in w's subtree would have size $SCS(w, d+1)$. We have $SCS(v, d) = \sum_{w \downarrow v} SCS(w, d+1)$.

Let a be the root of T. If $v = a$, then any subset $K = K_v$ of finite clustering cost must include v. Similarly, if $v \neq a$ but $d = r + 1$, then any subset K_v as described above must include v; otherwise, $cost(v, K) \geq r + 1$. If $v \neq a$ and $d \leq r$, then we v may or may not be in K_v depending on which one leads to a total size for K_v as described above. We have the following recursive definition:

$$SCS(v, d) = \begin{cases} 1 + \sum_{w \downarrow v} SCS(w, 1) & \text{if } v = a \text{ or } d = r + 1 \\ \min\{1 + \sum_{w \downarrow v} SCS(w, 1), \sum_{w \downarrow v} SCS(w, d+1)\} & \text{otherwise} \end{cases}$$

The exact value of d does not matter if we're trying to use our function to find the smallest subset K, so we'll work to compute $SCS(a, 1)$. The first parameter to our function is a vertex of T, but for the second parameter, we have $1 \leq d \leq r + 1$. Therefore, we'll store subproblem solutions in a collection of 1D arrays $v.SCS[1 .. r + 1]$, one for each vertex v of T. Every subproblem depends on those for the first parameter's children, so we'll solve subproblems in postorder. The order we consider settings of the second parameter does not matter. There are $O(nr)$ subproblems, and the solution to each is used at most once, ***so the total running time will be $O(nr)$***. Here's a pseudocode implementation.

```
SMALLESTSUBSET($T, r$):
    $a \leftarrow$ root of $T$
    for each vertex $v$ in postorder
        $keepv \leftarrow 1$
        for each child $w$ of $v$
            $keepv \leftarrow keepv + SCS[w, 1]$
        for each $d \leftarrow 1$ to $r + 1$
            if $v = a$ or $d = r + 1$
                $SCS[v, d] \leftarrow keepv$
            else
                $skipv \leftarrow 0$
                for each child $w$ of $v$
                    $skipv \leftarrow skipv + SCS[w, d + 1]$
                $SCS[v, d] \leftarrow \min\{keepv, skipv\}$
    return $SCS[a, 1]$
```

■

Rubric: 7 points total: 4 points total for the recurrence; −1 for no justification, −1.5 for missing the base cases; 2 points for filling the table (0 if the recurrence is very wrong); 1 points for the running time analysis.

+3 extra credit points for a correct *and justified* $O(n)$ time algorithm that does not depend on outside sources. (A likely solution came up during office hours. It's fine if they acknowledge others but appear to have come up with the main ideas themselves. Greedy algorithms require a very convincing proof of correctness.)

(b) Describe an algorithm to compute, given the tree T and an integer k, the minimum clustering cost of any subset of k vertices in T. For full credit, your algorithm should run in $O(n^2 \log n)$ time.

Solution: Let r^* be the minimum clustering cost. Let r be any integer such that $1 \leq r \leq n$. Adding more vertices to a subset can only decrease its clustering cost. Therefore, if $r \geq r^*$ then the smallest subset of clustering cost at most r has size at most k. Conversely, if $r < r^*$, then the smallest subset of clustering cost at most r has size strictly greater than k (otherwise, we could reduce r^*). Therefore, we'll do a binary search over choices of r to find r^*.

```
MINCLUSTERINGCOST($T, k$):
    $left \leftarrow 1$
    $right \leftarrow n$
    while $left < right$
        $r \leftarrow \lfloor (left + right)/2 \rfloor$
        if SMALLESTSUBSET($T, r$) $\leq k$
            $right \leftarrow r$
        else
            $left \leftarrow r + 1$
    return $left$
```

The size of the search space $right - left + 1$ decreases by approximately $1/2$ with every iteration of the while loop, so there are $O(\log n)$ iterations. Each iteration does an $O(nr)$ time call to SmallestSubset(T, r) and $r \leq n$, so each iteration takes $O(n^2)$ time. ***The total running time is $O(n^2 \log n)$.*** ■

Rubric: 3 points total: 2 points for the algorithm; 0.5 points for justification, 0.5 points for running time analysis. Simply explaining how comparisons guide a binary search is sufficient for full credit.

+2 extra credit points for a correct *and justified* algorithm whose worst-case running time is an $o(\log n)$ factor higher than the running time of the algorithm given in part (a) (that is a "little-oh"). (I don't know if the likely $O(n)$ time algorithm for part (a) can be extended to an $o(n \log n)$ time algorithm or if an $o(n \log n)$ time algorithm even exists for this problem.)

Remark SmallestSubset along with the comparison of its return result to k is an example of a ***decision procedure*** for an optimization problem. One uses a decision procedure to determine how the optimal value for a decision problem compares to a given guess value. They're usually used as subroutines in binary searches and other optimization algorithms based on similar search strategies. An efficient optimization algorithm immediately leads to an efficient decision procedure, so finding one is often the same difficulty as finding the other.

Also, this problem is another excellent example of Kyle writing more than necessary for full credit in order to explain his thought process.

CS 6363.003 Spring 2021 Official Solutions

Homework 3 Problem 3

Describe and analyze an algorithm to compute an optimal *ternary* prefix-free code for a given array of frequencies $f[1 \,..\, n]$.

Solution: We'd naturally like to use the same strategy we used for constructing optimal binary code trees. However, it is not always possible to construct a *full* ternary code tree. Fortunately, we can observe that a full ternary code tree does exist if and only if n is odd. For $n = 1$, the empty tree serves as an example of a ternary code tree where all (zero) internal nodes have exactly three children. For $n = 2$, there are not enough leaves to create an internal node with exactly three leaves, and so no full ternary code tree exists. Suppose we try to build a full ternary code tree for $n \geq 3$. A full ternary tree with n leaves can be built by adding three children to some leaf of a full ternary tree with $n-2$ leaves. Inductively, the full ternary with $n-2$ leaves exists if and only if $n-2$ is odd if and only if n is odd.

For simplicity, from here on we'll assume n is odd. If not, we can add one more entry of 0 to the end of f, representing a new character that never actually appears in the message we'd like to encode.

We may now observe that there exists an optimal ternary code tree that is in fact full. Let T be any optimal ternary code tree. If any internal node v of T has one child, we can delete v and attach the child directly to v's parent, reducing the depth of all descendant leaves of v. If any internal node u has two children, a similar parity argument to that given above implies there must be another internal node v with two children. Without loss of generality, let the depth of u be at least the depth of v. We can take attach either child of u as a child of v instead without increasing the depth of any of u's descendant leaves, and then perform the same one child trick we just described for the other child of u. There are now two fewer internal nodes with two children, so inductively, we know repeating this exchange leads to an optimal ternary code tree that is full.

Now that we know to look for a full tree, we can design our greedy algorithm. As in binary code trees, we claim the optimal ternary code tree contains the three least frequent characters as singling leaves of maximum depth. Let T be a full optimal code tree and suppose otherwise. Let x, y, and z be the three least frequent characters with $f[x] \leq f[y] \leq f[z]$, and let a, b, and c be three sibling leaves of maximum depth with $f[a] \leq f[b] \leq f[c]$. Suppose $x \neq a$. As before, we can swap the positions of x and a. Because the original depth of a was at least the original depth of x, and $f[x] \leq f[a]$, the swap will not increase the cost of the code tree. We can do similar swaps between y and b and z and c to prove our claim.

Finally, we describe the algorithm. We replace x, y, and z as defined above with a new character X where $f[X] = f[x] + f[y] + f[z]$ and then recursively compute an optimal code tree T' for the now shorter alphabet. We replace the leaf X in T' with an internal node that has leaves x, y, and z as its children. As before, the previous claim implies there is an optimal code tree T constructed by adding the three leaves to some code tree T' for the frequency array containing X. The cost of T is exactly $f[X]$ greater than the cost of T', so we are correct to optimize T by recursively optimizing T' as we do.

For the sake of completeness, here's some pseudocode based on the strategy described above. Similar to Erickson Figure 4.4, we construct three arrays listing the Left, Middle, and Right children of each node along with their Parents. We use a standard binary heap for a priority queue supporting INSERT and EXTRACTMIN operations.

```
BuildTernaryHuffman($f[1\,..\,n]$):
    for $i \leftarrow 1$ to $n$
        $L[i] \leftarrow 0$; $M[i] \leftarrow 0$; $R[i] \leftarrow 0$
        Insert($i, f[i]$)
    for $i \leftarrow n+1$ to $\frac{3n-1}{2}$
        $x \leftarrow$ ExtractMin()
        $y \leftarrow$ ExtractMin()
        $z \leftarrow$ ExtractMin()
        $f[i] \leftarrow f[x] + f[y] + f[z]$
        Insert($i, f[i]$)
        $L[i] \leftarrow x$; $P[x] \leftarrow i$
        $M[i] \leftarrow y$; $P[y] \leftarrow i$
        $R[i] \leftarrow z$; $P[z] \leftarrow i$
    $P[\frac{3n-1}{2}] \leftarrow 0$
```

There are $O(n)$ binary heap operations, ***so the algorithm takes $O(n \log n)$ time***.

■

Rubric: 10 points total: 5 points for the algorithm; 3 points for justification, 2 points for running time analysis. Pseudocode is not necessary, but they do need to carefully explain how to handle even n and why it's still correct to find a full tree with the least frequent characters as siblings.

CS 6363.003 Spring 2021 Official Solutions

Homework 3 Problem 4

There are n galaxies connected by m intergalactic teleport-ways. Each teleport-way joins two galaxies and can be traversed in both directions. Also, each teleport-way e has an associated cost $c(e)$ of dollars, where $c(e)$ is a positive integer. A teleport-way can be used multiple times, but the toll must be paid every time it is used.

Describe and analyze an algorithm to compute the smallest number of times Judy needs to teleport to travel from galaxy s to galaxy t so that the total cost is a multiple of five dollars.

Solution: Per the hint, we'll build a *directed* graph $G = (V, E)$ to model the scenario. At any point during Judy's trip, she'll want to know 1) where she is, and 2) how much small change she has. The second point can be modeled as the total amount of dollars she has spent so far, ($\bmod 5$). Therefore, for each galaxy u and each integer $x \in \{0, \dots, 4\}$, we add ordered pair (u, x) to V. For each teleport-way e between two galaxies u and v and each integer $x \in \{0, \dots, 4\}$ we add *two* directed edges $(u, x) \to (v, x + c(e) (\bmod 5))$ and $(v, x) \to (u, x + c(e) (\bmod 5))$.

Now we observe the following: Graph G contains a directed walk between vertices (u, x) and (w, z) of length ℓ if and only if Judy can travel from galaxy u to galaxy w using exactly ℓ teleports of total cost $z - x (\bmod 5)$. Suppose such a walk W exists. If $\ell = 0$, then $w = u$ and $z - x = 0 (\bmod 5)$. Indeed, there is a corresponding trip where she doesn't teleport at all. If $\ell \geq 1$, let $(v, y) \to (w, z)$ be the last edge of W. Inductively, Judy can travel from galaxy u to galaxy v using $\ell - 1$ teleports of total cost $y - x (\bmod 5)$. The existence of edge $(v, y) \to (w, z)$ implies there is a teleport-way e between v and w of cost $c(e) = z - y (\bmod 5)$. By traveling from v to w using this teleport-way, Judy's trip now ends at w after using a total of ℓ teleports costing $v - x + z - v = z - x (\bmod 5)$.

Conversely, suppose there is a trip from galaxy u to galaxy w using exactly ℓ teleports of total cost $z - x (\bmod 5)$ for some integers $x, z \in \{0, \dots, 4\}$. If $\ell = 0$, then the trip started and ended at u and cost $0 (\bmod 5)$ dollars, implying $z = x (\bmod 5)$. The trivial walk from (u, x) to (u, x) also has 0 edges. Now, suppose $\ell \geq 1$. Let e be the last teleport-way used by the trip, and let v be the other galaxy the teleport-way e shares with w. The trip up to but excluding e goes from u to v and uses $\ell - 1$ teleports of total cost $z - c(e) (\bmod 5)$. Inductively, we know there is a walk W' in G from (u, x) to $(v, z - c(e) (\bmod 5)$. The existence of teleport-way e implies G contains an edge $(v, z - c(e) (\bmod 5)) \to (w, z (\bmod 5))$. Appending that edge to W' gives the desired walk of ℓ edges.

The above observation implies our original problem is equivalent to finding the length of the shortest walk from $(s, 0)$ to $(t, 0)$. We do so using a breadth-first search in G starting at $(s, 0)$. Graph G contains $|V| = 5n$ vertices and $|E| = 10m$ edges. Building G and running the BFS takes $O(V + E)$ time, ***so the running time of this algorithm in terms of the input size is $O(n + m)$.*** ■

Rubric: 10 points total: 5 points for the algorithm; 3 points for justification, 2 points for running time analysis. Any justification that explains at least one of a) what each vertex and edge represents or b) the direct relationship between walks and trips is sufficient for full credit.