CS1510 Dynamic Programming Problems 13, 16 & 17

Rebecca Negley, Sean Myers

October 3, 2011

13. Our goal is now to consider the Knapsack problem, and develop a method for computing the actual items to be taken in $O(L)$ space and $O(nL)$ time.

    (a) Consider the following problem. The input is the same as for the knapsack problem, a collection of n items $I_1, ..., I_n$ with weights $w_1, ..., w_n$ and values $v_1, ..., v_n$ and a weight limit L. The output is in two parts. First you want to compute the maximum value of a subset S of the n items. that has weight at most L, as well as the weight of this subset. Let us call this value and weight v_a and w_a. Secondly for this subset S you want to compute the weight and value of the items in $\{I_1, ..., I_{n/2}\}$ that are in S. Let us call this value and weight v_b and w_b. So your output will be two weights and two values. Give an algorithm for this problem that uses space $O(L)$ and time $O(nL)$.

    Solution: Consider the algorithm used in number 12 where we construct a size $2 \times L$ matrix. Here, we will modify that algorithm to construct a $2 \times L \times 3$ matrix. The third matrix dimension corresponds to 3 values: the current value, the value that the current set had at $I_{n/2}$, and the weight that the current set had at $I_{n/2}$. Then, the space is $O(L)$. Before item $n/2$, set the $I_{n/2}$ weight and value to zero. We loop through all n inputs and rotate between the two rows of the matrix rather than traversing down an $n \times L$ matrix. After the $n/2$ iteration, we copy the $n/2$ value and weight from the location we use to get our current value. Without these modifications, our algorithm is essentially the one described in class. At each iteration i, the max value at weight j is the maximum of the maximum value at weight j in the $i-1$ iteration and the maximum value at weight $j - w_i$ in the $i-1$ iteration + v_i, corresponding to whether we add the ith item or not. Then, the algorithm would look like:

```
knap(I_1,,...,I_n, L)
    ks[2][L]
    for i from 1 to n/2 - 1
        for j from 1 to L
            if w_i < j
              ks[i%2][j][0] = max{ks[(i-1)%2][j][0],ks[(i-1)%2][j - w_i][0] + v_i}
            else
              ks[i%2][j][0] =ks[(i-1)%2][j][0]
            ks[i%2][j][1] = 0
            ks[i%2][j][2] = 0
    for j from 1 to L
        if w_{n/2} < j
            ks[0][j][0] = max{ks[1][j][0],ks[1][j - w_i][0] + v_i}
        else
            ks[0][j][0] =k[1][j][0]
        ks[0][j][1] =ks[0][j][0]
        ks[0][j][1] = j
    for i from n/2 + 1 to n
```

for j from 1 to L
  if $w_i < j$
    ks[i%2][j][0] = max{ks[$(i-1)$%2][j][0],ks[$(i-1)$%2][$j-w_i$][0] + v_i}
  else
    ks[i%2][j][0] =ks[$(i-1)$%2][j][0]
  if $w_i < j$ AND ks[$(i-1)$%2][$j-w_i$][0] + v_i >ks[i%2][j][0]
    ks[i%2][j][1] =ks[$(i-1)$%2][$j-w_i$][1]
    ks[i%2][j][2] =ks[$(i-1)$%2][$j-w_i$][2]
  else
    ks[i%2][j][1] =ks[$(i-1)$%2][j][1]
    ks[i%2][j][2] =ks[$(i-1)$%2][j][2]
$\{v_a, w_a\} = \{\max_j \text{ks}[L\%2][j][0], j\}$
$\{v_b, w_b\} = \{\text{ks}[L\%2][w_a][1],\text{ks}[L\%2][w_a][2]\}$
return $\{v_a, w_a, v_b, w_b\}$

(b) Explain how to use the algorithm from the previous subproblem to get a divide and conquer algorithm for finding the items in the Knapsack problem a and uses space $O(L)$ and time $O(nL)$.

Solution: Observe that the algorithm above gives us the maximum value and and its corresponding weight and the value and weight that the optimal set had at step $n/2$. We know that the items in $\{I_1, \ldots, I_{n/2}\}$ that are in the optimal solution have total weight w_b and total value v_b. Then, we must have that the items in $\{I_{n/2+1}, \ldots, I_n\}$ that are in the optimal solution have weight $w_a - w_b$ and value $v_a - v_b$. We can then break this problem down into two subproblems: knap$(I_1, \ldots, I_{n/2}, w_b)$ and knap$(I_{n/2+1}, \ldots, I_n, w_a - w_b)$. Consider a divide a concquer algorithm where we follow through the algorithm above and then call knap$(I_1, \ldots, I_{n/2}, w_b)$ and knap$(I_{n/2+1}, \ldots, I_n, w_a - w_b)$ recursively to construct our optimal subset. Add the following base case to knap$(I_j, \ldots, I_k$,val): if $j = k$ and $val > 0$, return I_j. Otherwise, return the empty set. The runtime of the algorithm in part (a) $n * L$. Then, runtime(knap$(I_1, \ldots, I_n, L)$) = runtime(knap$(I_1, \ldots, I_{n/2}, L_1)$)+runtime(knap$(I_{n/2+1}, \ldots, I_n, L_2)$)+ n*L, where$L_1 + L_2 \leq L$. Observe $\frac{n}{2} * L1 + \frac{n}{2} * L2 \leq \frac{n}{2} L$, so runtime= $O(n * L)$.

16. Give an algorithm for the following problem whose running time is polynomial in $n + W$:
Input: positive integers $w_1, ..., w_n$, $v_1, ..., v_n$, and W
Output: The maximum possible value of $\sum_{i=1}^{n} x_i * v_i$. subject to $\sum_{i=1}^{n} x_i * w_i. \leq W$ and each x_i is a nonnegative integer.

Solution: This problem will create a branching factor of W at each node in the tree, because we know that x_i has to be less than W. To solve this problem, we need to make sure that at each level, even if the branching factor is W, it only creates W branches at each sublevel. We need some pruning rules to do this:

(a) If we have chosen $x_1 ... x_{i-1}$, prune any branches where $x_i * w_i + \sum_{j=0}^{i-1} w_j * x_i > W$. In other words, prune any branches where adding the next weight will make the overall weight larger than W.
(b) If two branches b_1 and b_2 have weights where $w_{b1} = w_{b2}$, and both b_1 and b_2 are on the same level L, prune the tree with lesser sum of values.

The second rule makes sense because if you have a correct solution with the lower summed value: $v_{b1} + v_{x1} + ... + v_{xn}$, if it is the same weight then it can be replaced and have a higher value, making the above solution an incorrect solution, since it is not the max value.

With these two pruning rules, let's now create the algorithm. First, we make a table maxSum[n, W], and initialize all the values to 0. Each index will hold the max possible value for that level and that corresponding weight.

With this initialization, we know that row 0 is correct because nothing has been put in the set. So our first for loop will go from 1 to n. This will correspond to the depth of the tree we are at. Call this loop lvl for level

At each level, we must go through every weight, so we need another for loop going from 0 to W. call it curWeight. We also need to try all possible values of x, that equal up to W in each index, so we need yet another for loop of x going from 0 to the current iteration of curWeight, above.

First we must check if this solution is feasible, so we must check the constraints of the problem. If x*weight[lvl] > curWeight, then we continue because it violates pruning rule 1. Otherwise, take the max of either the current value at position maxSum[lvl,curWeight] or the value[lvl]*x + maxSum[lvl-1, curWeight-weight[lvl]*x]. In the second case, it is equivalent to saying: grab me the optimal value of the current weight minus the weight of the current thing we are trying to add. If the current value is greater than this, do not add it, otherwise add it.

In pseudo code, it will look like this:

```
for(i = 0 to n):
   for(j = 0 to W):
      maxSum[i][j] = 0; //Initalization

for(lvl = 0 to n):
   for(curWeight = 0 to W):
      for(x = 0 to curWeight):
         if(x*weight[lvl] > curWeight): continue;
         maxSum[lvl,curWeight] = max(maxSum[lvl, curWeight], value[lvl]*x + maxSum[lvl, curWeight-weight[lvl]*x])
```

At most x will be W, so let's say that for every inner loop, x is W iterations. Then the final runtime will be $O(n * w^2)$, which is a polynomial time of n+W. More specifically, $(n + W)^3$.

17. Give an algorithm for the following problem whose running time is polynomial in $n + L$ where
$L = max(\sum_{i=1}^{n} v_i^3, \prod_{i=1}^{n} v_i)$
Input: positive integers $v_1, ..., v_n$
Output: A subset S of the integers such that $\sum_{v_i \in S} v_i^3 = \prod_{v_i \in S} v_i$

Solution: First, it is recognizeable that this will be a binary tree, where each level is a branch whether you take the coin or not (and combinations of previous coin taking). The leaves will produce the solution if there is one. There are going to be 2^n leaves without pruning, which is not acceptable. So let's prune!

The only pruning rule that is applicable is if there are two nodes on the same level whose product and sum of cubes are equal, then prune one of them.

If we set up an array of boolean values like subAlg[lvl, S, M] where S is the summation of all the values, and M is the multiplicative product of all the values. First we set all values in the array to 0, except for 0,0,0 which we set to 1.

First we run a for loop on the levels, from 0 to n. In each , level, we transfer the table from the current level to the lower level. To do so, we must have for loops running through all the S and M. To compute the lower level, you set the top level 1's to the positions of subAlg[lvl-1, S, M] or the sum and multiplication if you don't accept the value of the lower level, value[lvl-1]. If you accept it,

you change the value of both $S + value[lvl - 1]^3$ and $M * value[lvl - 1]$ to 1 as well, or subAlg[lvl-1, $S + value[lvl - 1]^3$,M*value[lvl-1]].

In code it looks like this:

```
initialize subAlg[n,S,M] array. to all 0's
subAlg[0,0,0] = 1

for(lvl = 0 to n):
    for(i= 0 to S):
        for(j = 0 to M):
            subAlg[lvl+1,i,j] = subAlg[lvl,i,j]
            if(i+value[lvl]^3 > S || j*value[lvl] > M ): continue;
            subAlg[lvl+1,i+value[lvl]^3, j*value[lvl]] = subAlg[lvl,i,j]
```

The final algorithm will run in n*L*L, which is polynomial $n + L$. Finding the subset, is looking at subAlg[n][p][p], where p goes from 1 to min(S,M). If one of them is 1, then there is a subset, and then backtrack to find the values that were included.