How to Design Dynamic Programming Algorithms Sans Recursion

Kirk Pruhs
Computer Science Department
University of Pittsburgh
Pittsburgh, PA 15260
kirk@cs.pitt.edu

Abstract: We describe a method, which we call the Pruning Method, for designing dynamic programming algorithms that does not require the algorithm designer to be comfortable with recursion.

1 Introduction

In teaching algorithms courses, dynamic programming is the topic that maximizes the ratio of my students' perceived difficulty of the topic to my perceived difficulty of the topic. Most of the standard textbooks (e.g. [1, 2, 3, 5]) on algorithms offer the following strategy for designing a dynamic programming algorithm for an optimization problem $\mathcal{P}$:

1. Find a recursive algorithm/formula/property that computes/defines/characterizes the optimal solution to an instance of $\mathcal{P}$.

2. Then determine how to compute an optimal solution in a bottom-up iterative manner.

My students experience great difficulty with devising a recursive algorithm when the inductive hypothesis has to be strengthened.

As an example, consider the following Longest Increasing Subsequence (LIS) Problem:

INPUT: A sequence $X = x_1, \ldots, x_n$ of integers
OUTPUT: A longest increasing subsequence of X

So if $X = 12, 13, 23, 24, 16, 17, 18, 14, 15, 19$, the output would be $12, 13, 16, 17, 18, 19$.

Let $LIS(k)$ be a longest increasing subsequence of $x_1, \ldots, x_k$. The most obvious way to design a recursive algorithm for the LIS problem would be to inductively compute $LIS(k)$ from $LIS(k-1)$ and x_k. However, it is not to difficult to see that knowing $LIS(k-1)$ and x_k is insufficient information to compute $LIS(k)$. The standard fix is to strengthen the inductive hypothesis, that is, assume that the recursive call returns more information than $LIS(k-1)$. Eventually, one reaches the conclusion that one may need to return up to k subsequences of $x_1, \ldots, x_{k-1}$ in order to compute the longest increasing subsequence of $x_1, \ldots, x_k$ from this inductive information and x_k. Essentially the idea is that shorter subsequences that end in smaller numbers might be preferable, to longer subsequences that end in larger numbers, since they are easier to extend. As shown in [3], it is sufficient to remember the subsequence of each length that ends in the smallest last number. Let $LIS(k, \ell)$ be the smallest last number of a subsequence of length ℓ of $x_1, \ldots, x_k$. One can then compute $LIS(k, \ell)$ recursively in the following manner:

If $LIS(k-1, \ell-1) < x_k$
then $LIS(k, \ell) = \min(LIS(k-1, \ell), x_k)$
else $LIS(k, \ell) = LIS(k-1, \ell)$

This leads to the following code:

```
For k = 1 to n do
    For ℓ = 1 to n do
        If LIS(k − 1, ℓ − 1) < x_k
        then LIS(k, ℓ) = min(LIS(k − 1, ℓ), x_k)
        else LIS(k, ℓ) = LIS(k − 1, ℓ)
```

As is standard, we will omit the code for initializing the data structures, and for determining the actual longest increasing subsequence from final filled-in data structure.

Many of my students are still quite uncomfortable with recursion. For those that are comfortable with with recursion, the most difficult task seems to be identifying the additional information that needs to be added to the next attempt at an inductive hypothesis when the previous inductive hypothesis fails. For example, in the LIS problem it is a big leap from the initial naive inductive hypothesis to realizing that one should remember a linear number of subsequences.

We give a method, which we call the Pruning Method, for designing dynamic programming algorithms that does not require the algorithm designer to be comfortable with recursion. The Pruning Method is most obviously applicable if the underlying structure of the feasible solutions are subsets, or subsequences, including paths in a graph. Note that this includes well over half of the problems in the standard introductory algorithms texts. We illustrate this method using the longest increasing subsequence problem, and the standard single source shortest path problem. We offer the following general guidelines for designing a dynamic programming algorithm using the Pruning Method:

1. State the problem as a optimization problem so that the feasible solutions are subsets, subsequences, or paths.
2. Consider the standard enumeration tree where the collection of all feasible solutions are the leaves of this tree.
3. Determine how to prune redundant/unnecessary nodes from this tree.
4. Develop an iterative algorithm that generates the information in this tree level by level from the root to the leaves.

2 The LIS Problem

We now apply the Pruning Method to the LIS problem. In this problem the feasible solutions are subsequences of the input sequence $x_1, \ldots, x_n$. The root of the enumeration tree is labeled with the empty sequence. A node at depth $k-1$ with label S has two children at depth k, one labeled S and one labeled Sx_k. Thus the 2^k nodes in the kth level (nodes at depth k) are labeled with the 2^k subsequences of $x_1, \ldots, x_k$, and the leaves contain the 2^n feasible solutions. Figure 1 shows the enumeration tree for $n = 3$. Let $|S|$ denote the length of a sequence S, and let $S(j)$ denote the jth element of S. One obvious pruning rule is:

1. If any node α is labeled with a subsequence S_α that is not increasing, then the subtree rooted at α can be pruned from the tree since the label of every node in the subtree rooted at α is a nonincreasing sequence.

A less obvious pruning rule is:

2. If two nodes α and β at the same depth are labeled with subsequences S_α and S_β such that
   (a) $|S_\alpha| = |S_\beta| = \ell$, and
   (b) $S_\alpha(\ell) \geq S_\beta(\ell)$, then

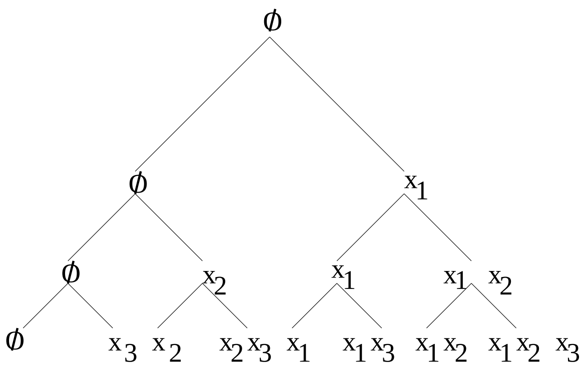

Figure 1: Tree of all Subsequences of x_1, x_2, x_3

   (c) we can prune the subtree rooted at α.

We claim that pruning the subtree rooted at α leaves an optimal solution in the remaining tree. To see this consider an optimal sequence of the form $S_\alpha T$ that is a label for a leaf in the subtree rooted at α. Then the subsequence $S_\beta T$ is also optimal, and is a label for a leaf in the subtree rooted at β. Thus at each level we need only remember one increasing subsequence of each possible length, namely the one that ends in the smallest last number. Since the possible lengths of a subsequence lie in the range from 0 to n, and there are $n+1$ levels in the tree, these pruning rules leave a pruned tree with $O(n^2)$ nodes. To obtain $O(n^2)$ time code, we let $LIS(k, \ell)$ be the smallest last number of an increasing subsequence of $x_1, \ldots, x_k$ of length ℓ. Updating LIS level by level leads to the following code, which is essentially the same as we obtained when using recursion to design an algorithm:

For $k = 1$ to n do
    For $\ell = 1$ to n do
        $LIS(k, \ell) = \min(LIS(k-1, \ell), LIS(k, \ell))$
        If $LIS(k-1, s-1) < x_k$ then
            $LIS(k, \ell) = \min(x_k, LIS(k, \ell))$

Note that we write this code in the way that seems most intuitive given our development of the algorithm. Further reflection can often lead to cleaner or more efficient code.

One can see that an alternative second pruning rule could have been:

2. If two nodes α and β at the same depth are labeled with subsequences S_α and S_β such that
   (a) S_α and S_β end in the same number x_j, and
   (b) $|S_\alpha| \leq |S_\beta|$, then

(c) we can prune the subtree rooted at α.

Once again it is easy to see that pruning the subtree rooted at α leaves an optimal solution in the remaining tree. Thus at each level we need only remember a longest increasing subsequence ending at each x_j, $1 \leq j \leq n$. This leads us to define an array $LIS[k, j]$, $1 \leq k \leq n$ and $1 \leq j \leq n$, to be the length of the longest increasing subsequence of $x_1, \ldots, x_k$ that ends in x_j. Updating LIS level by level leads to the following code:

```
For k = 1 to n do
  For j = 1 to k − 1 do
    LIS(k, j) = max(LIS(k − 1, j), LIS(k, j))
    If x_j < x_k then
    LIS(k, k) = max(LIS(k, k), LIS(k − 1, j) + 1)
```

Note that when designing the algorithm for the LIS problem using recursion, one starts with a minimal amount of information and adds information as needed. Using the Pruning Method, we start with all the possible information that we could possibly need, i.e. all the feasible solutions, and discard information that is unnecessary.

3 Shortest Path Problem

The standard single source shortest path problem can be stated at follows:

INPUT: A directed edge-weighted graph G with designated vertex s.
OUTPUT: For each vertex v, the shortest path from s to v.

We assume that G is not allowed to have cycles with negative aggregate weight so that shortest paths are well defined. Here we let n denote the number of vertices in G, $|P|$ denote the length of a path P, and $d(u, v)$ denote the weight of the edge (u, v). In this problem the feasible solutions are directed paths starting from s containing at most $n-1$ edges. We can generate these feasible solutions as the leaves of an enumeration tree by having the kth level contain nodes labeled with all paths starting from s with at most k edges. Assume that we have a node α labeled with a path P_α, that ends at vertex v, in the kth level of the tree. Further assume that in G the directed edges leaving v are to the vertices $w_1, \ldots, w_j$. Then the $j+1$ children of α in the enumeration tree will be labeled $P_\alpha, P_\alpha w_1, P_\alpha w_2, \ldots, P_\alpha w_j$. The first three levels of the enumeration tree for the graph in figure 2 is shown in figure 3.

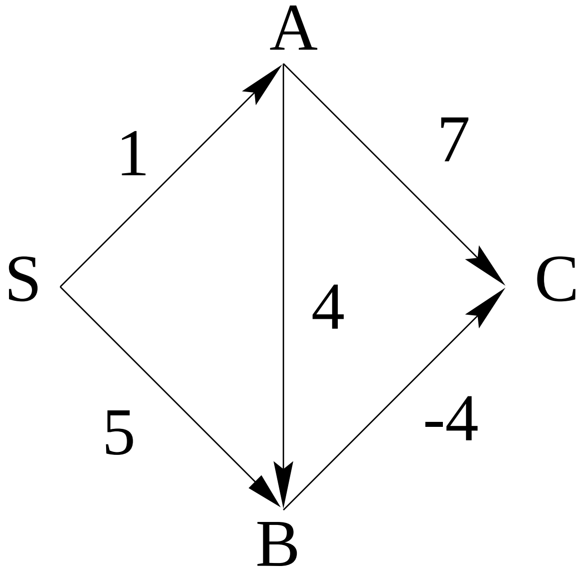

Figure 2: The Graph G

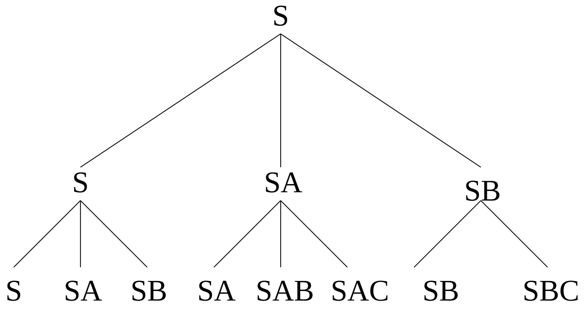

Figure 3: The First Three Levels of the Enumeration Tree

The natural pruning rule is to remember only the shortest path to a particular vertex. This can be stated more formally as:

1. If there are two nodes α and β, labeled with paths P_α and P_β, on the same level of the enumeration tree such that
   (a) P_α and P_β end at the same vertex v, and
   (b) $|P_\alpha| \geq |P_\beta|$,
   (c) then you can prune the subtree rooted at α.

We claim that pruning the subtree rooted at α leaves an optimal solution in the remaining tree. To see this consider an optimal path of the form $P_\alpha T$ that is a label for a leaf in the subtree rooted at α. Then the path $P_\beta T$ is also optimal, and is a label for a leaf in the subtree rooted at β. Since there are n possible last vertices and the tree is of height n, this pruning rule leaves a tree with $O(n^2)$ nodes. To obtain an algorithm we let $D[k, v]$ be the shortest path from s to v with k or less edges. Updating D level by level we get the following code:

For $k = 1$ to $n-1$ do
  For $v = 1$ to n do
    $D[k, v] = D[k-1, v]$
    For each edge $e = (v, w)$ do
      $D[w] = \min(D[w], D[v] + d(v, w))$

Note that this is the Bellman-Ford algorithm, and runs in time $\Theta(nm)$, where m is the number of edges.

4 Conclusions

I generally teach both the standard recursive method, and the Pruning Method, for designing dynamic programming algorithms. At least some students find the Pruning Method easier to master. One reason is that it seems to be easier to identify unnecessary information then it is to identify the new information that is required to make the inductive argument work. Another reason is that if in a new problem it is the case that the collection of feasible solutions has a known form (say subsets of a fixed set for example), then a student can determine how to construct the enumeration tree for that type of feasible solution by look-up. Thus, the first creative step required of the student is to determine how to prune the tree. Another more subtle reason is that when using the Pruning Method the designer needs to answer the question, "Which entries in row k of the array are effected by a particular entry in row $k-1$?". This question is often conceptually easier than the equivalent question that arises when developing the algorithm using recursion, which is, "Which entries in row $k-1$ of the array does one need to know to compute a particular entry in row k?"

Still, many students fail to master either method. In my opinion, the most common error that students make, when using either method, is to immediately attempt to develop the iterative table/array-based code (the last step), without first doing the preliminary steps where the intuition is developed. The following variant of the standard Subset Sum Problem is a good exercise that forces a student to think about the preliminary steps since the data structure used in the final algorithm can not be an array:

Assume that you are given a collection $B_1, \ldots, B_n$ of boxes. You are told that the weight in kilograms of each box is an integer between 1 and some constant L, inclusive. However, you do not know the specific weight of any box, and you do not know the specific value of L. You are also given a pan balance. A pan balance functions in the following manner. You can give the pan balance any two disjoint subcollections, say S_1 and S_2, of the boxes. Let $|S_1|$ and $|S_2|$ be the cumulative weight of the boxes in S_1 and S_2, respectively. The pan balance then determines whether $|S_1| < |S_2|$, $|S_1| = |S_2|$, or $|S_1| > |S_2|$. You have nothing else at your disposal other than these n boxes and the pan balance. The problem is to determine if one can partition the boxes into two disjoint subcollections of equal weight. Give an algorithm for this problem that makes at most $O(n^2\ L)$ uses of the pan balance.

Recursion and strengthening the inductive hypothesis are important concepts useful for purposes other than designing dynamic programming algorithms. Hence, one disadvantage of teaching the Pruning Method is that it can rob students of an opportunity to improve their recursive thinking skills. For more information on developing algorithms using recursion, I highly recommend [4], and more generally [3].

Acknowledgments: I would like to thank Marty Wolf, and Udi Manber for helpful comments.

References

[1] Gilles Brassard and Paul Bratley, Fundamentals of Algorithmics, Prentice Hall, 1996.

[2] Thomas Cormen, Charles Leiserson, and Ronald Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.

[3] Udi Manber, *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, 1989.

[4] Udi Manber, "Using induction to design algorithms,", *Communications of the ACM*, **31**, 1300 – 1313, November 1988.

[5] Richard Neapolitan and Kumarss Naimipour, *Foundations of Algorithms*, D. C. Heath, 1996.