

Towards a Memory-Efficient Knapsack DP Algorithm

Sanjay Rajopadhye

The 0/1 knapsack problem (0/1KP) is a classic problem that arises in computer science. The Wikipedia entry http://en.wikipedia.org/wiki/Knapsack_Problem has a lot of useful general information. Despite its simplicity (simple formulation and clear algorithms) it is known to be NP-complete, and so the best known algorithms to solve it are have exponential worst case complexity. Nevertheless, common instances of the problem are not difficult to solve.

These notes describe the first steps towards the development of a memory efficient dynamic programming (DP) algorithm for the 0/1KP, originally due to McConnell. We first recap the basic DP solution to 0/1KP.

Problem statement: Given n objects, each one with a weight w_i and profit p_i , a knapsack of capacity c , choose an subset of the n objects so as to maximize the profit without exceeding the capacity, i.e., maximize $\sum_{j=1}^n p_j x_j$ subject to the constraints $\sum_{j=1}^n w_j x_j \leq c$. and $1 \geq x_j \geq 0$.

Since the x_j s are binary numbers, there are 2^n candidate solutions. Because the problem is known to be NP-complete, one solution strategy is to exhaustively evaluate each of the candidate solutions. This leads to “search based” algorithms, and when coupled with techniques for efficiently pruning the search space, yields a family of algorithms called “branch-and-bound.”

DP is an alternative strategy. It is a classic algorithmic technique that can be used to solve many optimization problems other than the knapsack. It exploits recursive structure and relies on decomposing a problem instance into many instances of smaller, simpler subproblems. It usually works in two phases. In the first phase, we first determine the *value* (or *profit*) of an

optimal solution to the problem, and typically to all subproblems, and store these values in a table. In the second, so called “backtracking phase,” the actual solution that achieves the optimal profit is chosen by traversing this table of optimal profits.

Knapsack subproblems One way of thinking of a “smaller instance of a knapsack problem” is to have a different capacity: for any integer in the range $1 \dots c$, we have an instance of a similar but smaller knapsack problem. Another way is to consider subsets of the objects. For example if we restrict ourselves to just the first i objects, we also have a smaller instance of the problem. Following this idea, we see that we can formulate nc problem instances, one for each pair of integers $\langle i, k \rangle$ in the set $\{\langle i, k \rangle | 1 \leq i \leq n; 0 \leq k \leq c\}$.

Let us define a function $F(i, k)$ that gives the maximum profit of subproblem $\langle i, k \rangle$, i.e., the maximum profit that can be obtained with capacity k , while choosing from only the first i objects.

Exploiting recursive structure of subproblems If we investigate the optimal solution to this subproblem, $\langle i, k \rangle$, there are only two choices vis-à-vis the i -th object: either it is chosen in some optimal solution to the subproblem, or it is not. Let us look at the two cases one by one.

Case I: If the i -th object does not contribute to any optimal solution to the problem $\langle i, k \rangle$, then $F(i, k)$ is the same as $F(i - 1, k)$, the best solution that can be achieved by simply considering the first $i - 1$ objects at the same capacity.

Case II: On the other hand, if the i -th object *does* contribute to some solution of subproblem $\langle i, k \rangle$, it “takes up” a capacity of w_i and contributes p_i to the total profit. Let us keep aside this object and consider how the remaining capacity $k - w_i$ is filled. By the principle of optimality (subproblems of the optimal solution must also be solved optimally), the profit achieved by this must be $F(i - 1, k - w_i)$. Hence, for Case II, $F(i, k) = p_i + F(i - 1, k - w_i)$.

However, we do not know, *a priori*, whether or not the i -th object contributes to the solution of subproblem $\langle i, k \rangle$, and we must account for both cases, i.e., $F(i, k)$ is just the larger of the profit from each of the two. Putting this together, and adding “boundary conditions” (where one

or the other subproblems is missing, or has a trivial solution), we get the following recurrence equation.

$$F(i, k) = \begin{cases} k = 0 & 0 \\ i = 0 & 0 \\ 0 < k < w_i; i > 0 & F(i - 1, k) \\ k \geq w_i; i > 0 & \max \left(\begin{array}{c} F(i - 1, k) \\ p_i + F(i - 1, k - w_i) \end{array} \right) \end{cases} \quad (1)$$

Evaluation The above recurrence can be evaluated by allocating an $n \times (c + 1)$ table—an array with n rows and $c + 1$ columns—to store all the values and filling it up in an “appropriate” order. The order is flexible, as long as we respect the implicit dependences in the computation. Viewing the table as an array (origin at top-left, rows indexed by i , and columns by k) we see that in order to determine $F(i, k)$, we need two entries in the table, both from the previous row: one from directly above and one that is w_i to the left. Hence we may fill the table up row-by-row or column by column. Also, note that since only the previous row is needed to compute any given row, if we need to compute a single $F(i, k)$ or a row of values, we only need $\Theta(c)$ memory—we only maintain the previous row in order to compute the current one, and at the end of each row, we swap the previous and current rows.

Complete solution However, simply computing the value of $F(i, k)$ is not enough. We need to find the set of objects that make up the optimal solution. We can do this by keeping track of the “winner” whenever we compared to profits (i.e., the max term in the equation above). In other words, we define a Boolean function $B(i, k)$ to mean the following: $B(i, k) = 0$ if the i -th object does not contribute to any optimal solution of subproblem $\langle i, k \rangle$, and 1 otherwise (i.e., if there is some solution of subproblem $\langle i, k \rangle$ containing the i -th object).

Recurrence for $B(i, k)$

$$B(i, k) = \begin{cases} k = 0 & 0 \\ i = 0 & 0 \\ 0 < k < w_i; i > 0 & 0 \\ k \geq w_i; i > 0 & \text{if } F(i - 1, k) > p_i + F(i - 1, k - w_i) \text{ then } 0 \text{ else } 1 \end{cases} \quad (2)$$

Once the table of $B(i, k)$ is available, the actual solutions can be easily computed by calling the function

$\text{PrintSolution}(n - 1, c)$, where

$$\text{PrintSolution}(i, k) = \begin{cases} \text{if } i = 0, k \geq w_i & \text{print "select } i\text{"}; \text{return} \\ \text{if } i = 0, k < w_i & \text{print "unselect } i\text{"}; \text{return} \\ \text{else if } B(i, k) & \text{print "select } i\text{"}; \text{PrintSolution}(i - 1, k - w_i) \\ \text{else} & \text{print "unselect } i\text{"}; \text{PrintSolution}(i - 1, k) \end{cases} \quad (3)$$

It is also possible to not compute $B(i, k)$ explicitly, but save the elements of the table $F(i, k)$, and redo the comparison between the elements during backtracking.

Memory Efficient DP

The problem with the above algorithm is that in order to compute the final solution, we need to save at least $\Theta(nc)$ values (the B table). Therefore, we will derive a memory efficient algorithm that uses only $O(c)$ memory.

In this section, we will get you to think about the main ideas needed to develop the memory efficient algorithm. We will not give away the “spoiler” but leave it to Part II and the next lecture. We encourage you to think about the two main questions that we ask here. The main points we cover are: (i) a naive strategy that recomputes (parts of) the table solves the problem

in $O(c)$ memory but adds time complexity, (ii) the outline of a divide-and-conquer strategy that could do better, and (iii) the development of the parameters needed in order for it to succeed. We end with a question that you should try to answer before the next lecture.

Suppose that only the last two rows of the $F(n, k)$ table for $0 \leq k \leq c$ were saved. Then we would only have information to determine whether the *last* (i.e., the n -th) object contributed to an optimal solution. In addition, we would also be able to decide the *residual* capacity, c_r that the remaining objects took up in an optimal solution. If the n -th object does not contribute to any optimal solution then $c_r = c$, otherwise it is c . Now, a naive strategy is that after determining whether or not the n -th object contributed to an optimal solution, we recurse on the subproblem $\langle n - 1, c_r \rangle$, i.e., re-compute the table up to the $n - 1$ -th row with a capacity of c_r . This will solve the problem from the last object down to the first. Since in the worst case, c_r can be as large as c , the time complexity for this is,

$$\begin{aligned}
 T(n, c) &= nc + (n - 1)c + \dots 1 \cdot c \\
 &= c \sum_{i=1}^n i \approx \frac{1}{2}n^2c \\
 &= O(n^2c)
 \end{aligned}
 \tag{4}$$

This is not acceptable. We seek a much better algorithm, hopefully one that is only a constant factor slower than the full-table version. Divide-and-conquer is a classic algorithmic technique, and we want to apply it here. For this, we will divide the objects into two disjoint sets, “solve” the problem on each half and somehow “combine” the solutions to solve the original problem. The key trick is to decide what it means to “solve” and to “combine.” For our problem, “solve” means determining the last-row of the table in a memory-efficient manner,¹ so the

¹Note that this is not really solving the sub-problem, since we do not still know what choice of objects to pick. In the rest of this discussion, “solve” in quotes means determining the last-row of a subproblem in a memory-efficient manner, while *solv*, without the quotes is actually obtaining the solution to the knapsack problem.

heart of the algorithm is in the “combine” step.

Because we are going to break up the set of objects, let us modify our notion of “knapsack subproblems.” Instead of thinking of the $\langle i, k \rangle$ -th subproblem, let us think of the $\langle i, j, k \rangle$ -th subproblem, where we select from among the i through j objects and optimally try to fill a knapsack of capacity k (so our old $\langle i, k \rangle$ -th subproblem is just the $\langle 1, i, k \rangle$ -th subproblem in the new notation). Note that the tables for two subproblems $\langle i, k \rangle$ and $\langle i', j, k \rangle$ are, in general, completely different, even though they both end at the j -th row.

We have the outline of our solution

- first “solve,” i.e., build the last row of the table for, the subproblems $\langle 1, \frac{n}{2}, c \rangle$ and $\langle \frac{n}{2} + 1, n, c \rangle$. This will need nc time/work.
- Next we somehow “combine” this information in such a way that we have two recursive sub-problems to solve.

Even before we get to the details of the “combine” step—the pieces and how to put them together—let us do an analytical exercise. What should be the size of the subproblems that we will set ourselves to “solve” recursively? In other words, when would the divide-and-conquer strategy lead to a better algorithm.

For example, if each of the two subproblems had a size $\frac{n}{2}c$, would this provide any improvement? To analyze this, let us compute the resulting complexity. We know that with a divide-and-conquer strategy that breaks the number of objects into two equal halves at each step, the number of steps before we get to a singleton object is $\lceil \lg n \rceil$. Since the total work at each decomposition adds up to nc , the total complexity will be $O(nc \lg n)$. Although this is a considerable improvement over the $O(n^2c)$ of the naive memory-efficient algorithm, it is still worse than the full-table version, so we are not yet done.

So the question you need to solve is: what should be the complexity of the two subproblems so that overall, we retain $O(nc)$ complexity. The answer to this will not give us the whole algorithm, but it will guide us to the strategy to follow. Please try to answer this before the next lecture.