Variant of the knapsack problem

How would you approach the knapsack problem in a dynamic programming situation if you now have to limit the number of item in the knapsack by a constant p ? This is the same problem (max weight of W, every item have a value v and weight w) but you can only add p item(s) to the knapsack and obviously need to optimize the value of the knapsack.

Do we need a 3rd dimension or we could find an other approach without it. I tried to simply add the number of item in the knapsack in the cell and taking the max value at the end with the number of item <= p but it is not the BEST solution.

algorithms optimization dynamic-programming knapsack-problems

asked Nov 30 '13 at 19:46
user11536
58 ● 1 ● 4

This is a nice homework exercise. What have you tried? Are you comfortable doing dynamic programming? (If not, maybe try some exercises to practice with it.) Have you studied the standard dynamic programming algorithm for the knapsack problem? Look for a way to modify that standard approach. Your main task is to design what the set of subproblems should be. In the standard approach, a subproblem is characterized by one parameter (a bound on the weight of the items). You might consider using two parameters (so a larger set of subproblems). Try various possibilities -- what do you get? – D.W. ♦ Dec 2 '13 at 4:04

1 Answer

Very nice question!

You are twice right:

1. Propagating the number of items in the knapsack does not lead to optimal solutions.
2. *One* solution consists of adding a third dimension. This is rather simple but it is necessary to take some facts into account when doing so. Note however that it is not the only alternative

In the following, I am assuming that you are familiar with the solution based in dynamic programming. In particular, **I will not discuss how to traverse the table backwards to determine the solution**.

Let us first focus on the typical case: *the number of items is unrestricted*. In this case, you just build a table T where $T_{i,j}$ contains the optimal value when the overall capacity of the knapsack equals i and only the first j items are considered. From here:

$$T_{i,j} = \max\{T_{i,j-1}, T_{i-w_j,j-1} + v_j\}$$

where w_j and v_j stand for the weight and value of the j-th item respectively. If C is the overall capacity of your knapsack and there are in total N items the optimal solution is given by $T_{C,N}$. This algorithm is known to run in pseudo-polynomial time and one of its beauties is that it only considers those combinations that fit the maximum capacity.

However, this is not enough when adding your constraint: a maximum number of items p. The reason is that the previous recurrence formula does not take into account different combinations of items:

1. First, if $T_{i,j-1} < (T_{i-w_j,j-1} + v_j)$ then $T_{i,j} = (T_{i-w_j,j-1} + v_j)$ so that the j-th item is added to the knapsack in spite of the maximum number of items considered, p ---so that you might be violating your constraint. Well, you might be tempted here to apply the preceding formula keeping track of the number of items inserted at each step and do not add others if the number of items currently in the knapsack exceeds p but,
2. Second, if $T_{i,j-1} > (T_{i-w_j,j-1} + v_j)$ then $T_{i,j} = T_{i,j-1}$ so that this item is not added but that might be a big mistake in case the optimal solution $T_{i,j-1}$ already consists of the maximum number of items to insert into the knapsack. The reason is that we are not properly comparing: on one hand, to preserve the optimal solution consisting of p items selected among the previous $(j-1)$; on the other hand, to insert the j-th item and, additionally consider the best subset with $(p-1)$ items among the previous $(j-1)$.

So that a first solution consists of adding a third dimension. For your case, let $T_{i,j,k}$ be the optimal solution when the capacity of the knapsack is i, only the first j items are considered and it is not allowed to put more than k items in the knapsack. Now,

- If you are computing $T_{i,j,k}$ for a number of items strictly less or equal than the number of items that can be inserted ($j \leq k$) then proceed as usual but using the same value of k: $T_{i,j,k} = \max\{T_{i,j-1,k}, T_{i-w_j,j-1,k} + v_j\}$
- Now, if you have to compute $T_{i,j,k}$ for a number of items strictly larger than the number of items that can be inserted ($j > k$) then: $T_{i,j,k} = \max\{T_{i,j-1,k}, T_{i-w_j,j-1,k-1} + v_j\}$

The first expression should be clear. The second works since the $(k-1)$-th layer of the table T keeps track of the best combination of $(k-1)$ items among the first $(j-1)$ as required above.

An efficient implementation of this algorithm does not need to compute $T_{i,j,k}$ for all k. Note that the preceding recurrence relationships relate layer k with $(k-1)$ and thus, it is possible to alternate between two successive layers (e.g., if you are interested in the optimal solution with $k=4$ you just use two consecutive layers: 0 and 1, 1 and 2, 2 and 3, 3 and 4 and you're done). In other words, this algorithm takes twice the memory required by the traditional approach based on dynamic programming and thus, it can be still run in pseudo-polynomial time.

Be aware, however, that this is not the only solution! And there is another you might find more elegant. In the preceding formulae, we retrieved the optimal solution which consisted of no more than $(k-1)$ items among the first $(j-1)$ as $T_{i,j-1,k-1}$. However, it should be clear that this is precisely equal to $\max_{p=0,j-1}\{T_{i,p}\}$ just by using the original table!! ie., the optimal solution with no more than k items can be also retrieved by considering the optimal solutions with 1 item, 2 items, 3 items, ... $(j-1)$ items ... To make this formulation work you should also keep track of the number of items considered in every partial solution so that you will need two integers per cell. **This memory occupation results in precisely the same memory requirements of the algorithm shown above (using a third dimension in the form of layers k)**.

Hope this helps,

answered Dec 2 '13 at 8:43

Carlos Linares López
2,694 ● 8 ● 23

Very great response, thank you. I've been able to get through it before your post by also implementing a 3rd dimension. – user11536 Dec 3 '13 at 22:53

Ouch, thanks a lot for closing the question and glad to hear that you liked the response. In order to clarify my ideas I also tried an implementation of this algorithm in Python. If you are interested to have a look at it, let me know and I will happily post it (or send it to you). Cheers, – Carlos Linares López Dec 4 '13 at 7:37

Amazing explanation of multidimensional knapsack problem.However I was wondering if we had similar case but with exactly k elements,we will only look at the values returned by the kth column of the 3rd dimension.If there are no values there then return 0.I am not sure if I am right because I am still new to dynamic programming. – StevelrwinJun 26 '14 at 20:17

@CarlosLinaresLópez great answer. Could you please share the python script too? Maybe post it on gist.github.com? – Saad Malik Aug 23 '15 at 1:39

1 Hi @Carlos! I posted a follow-up question to using your alternate formula here: Finding the n-best items in a 0/1 Knapsack. Anyway I hope you're enjoying your vacation! – Saad Malik Aug 23 '15 at 23:24