Dynamic Programming of the Knapsack Problem

Paul Franklin, Yangqi Su, Yaying Shi

February 6, 2020

1 Introduction

Implementations were created for bottom-up and top-down solutions to the Knapsack problem. In the bottom-up solution, optimal solutions are stored in a matrix, while in the top-down solution, the storage is located in a dictionary.

2 Bottom-Up Solution of Knapsack

$$Knapsack(Size, Val, Capa)$$

where

$Size$	is the set of the sizes of the items ordered by the distinct item indices under the binary relation $<$,
Val	is the set of the values of the items ordered by the distinct item indices under the binary relation $<$, and
$Capa$	is the total capacity of the knapsack.

In the bottom-up Knapsack solution, we assume that $|Size| = Capa$. It follows that $|Val| = Capa$ because $|Val| = |Size|$. Alternatively, we assume that $|Val| = Capa$ and it follows that $|Size| = Capa$ because $|Val| = |Size|$. The algorithm initializes dynamic storage as a matrix where all elements in the first row or the first column have a value of 0 and its dimensions are the cardinality of items, $|Size|$, + 1 and the capacity, $Capa$, + 1. For the submatrix from the 2nd row and 2nd column up to and including the last row and last column, a recursive algorithm computes from a decision function with three cases, where the recursion is over the elements of each column for each row. If the optimal subproblem leading to and including the previous item, or an initialization value, had greater value, then the same solution is chosen at the instance of recursion. If the remaining capacity does not support the size of the item under consideration, then the same decision is made. Otherwise, there is sufficient capacity and the optimal subproblem leading to and including the previous item did not have greater value than the combination of the values of this item and its optimal subproblem under addition. The algorithm continues until the last element of the matrix; all distinct optimal solutions have an equivalent value.

3 Top-Down Solution of Knapsack

$$Knapsack(Size, Val, Capa)$$

where

$Size$	is the set of the sizes of the items ordered by the distinct item indices under the binary relation $<$,
Val	is the set of the values of the items ordered by the distinct item indices under the binary relation $<$, and
$Capa$	is the total capacity of the knapsack.

The top-down *Knapsack* solution is responsible for initializing the *Topdown* subroutine as described in the “Top-Down Subroutine” subsection.

3.1 Top-Down Subroutine

$$Topdown(hashtable, Size, Val, capa, n)$$

where

$hashtable$	is the storage of optimal solutions to subproblems that have already been computed,
$Size$	is the set of the sizes of the items ordered by the distinct item indices under the binary relation $<$,
Val	is the set of the values of the items ordered by the distinct item indices under the binary relation $<$,
$Capa$	is the total capacity of the knapsack,
$capa$	is the remaining capacity of the knapsack, and
n	is the remaining number of items in the knapsack.

The *Topdown* subroutine has an initialization vector of the total capacity $Capa$ and the remaining number of items n in the bag from which items are removed. It is a recursive piecewise function with base cases of $capa^* <= 0$, $n^* = 0$, the $n-1^{th}$ element of $S > capa^*$, and the solution to the input $I^* = (capa^*, n^*)$ has already been computed and is stored as an element of a dictionary *hashtable*. Recursively, a decision function chooses between the maximum net value obtained from whether or not the last item is taken, effectively searching the entire solution space and returning the net value of any optimal solution. In the first two base cases, it is not possible to take any items, so the value of 0 is returned. In the third base case, the item under consideration does not satisfy the capacity constraint and the subroutine continues from the next possible item. Lastly, the solution retrieves solutions that have already been computed in the fourth base case, rather than recomputing them.