Approaches to Solve a Knapsack Problem

Sachin Prasanna

Department of Information Technology
National Institute of Technology Karnataka, Surathkal

***Abstract* – This paper gives an view on the various methods that can be used to solve the 0/1 Knapsack Problem.**

Key Terms – Dynamic Programming, Memoization, Tabulation, Space Optimization, Knapsack, Recursion

I. PROBLEM STATEMENT

You are given weights and values of N items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. Note that we have only one quantity of each item. In other words, given two integer arrays val[0..N-1] and wt[0..N-1] which represent values and weights associated with N items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don't pick it (0-1 property).

II. APPROACES TO SOLVE IT

1. **Simple Recursion:** A novice approach from the knapsack problem is to simply pick all items which when combined have a lesser weight than the maximum capacity of the knapsack. Finally, the maximum of this is returned as the final answer. This is implemented by the inclusion – exclusion principle, where in one traverses the entire array and either picks the element or does not pick the element.
Both types of traversals are possible, where one can traverse the array from left to right or vice versa.

2. **Memoization:** The above-mentioned method is not time efficient as the time complexity runs to exponential time. A better way is to store some values of the recursive calls in a 2 dimensional array, such that when a recursive call is called with the same parameters which match the particular indices of the 2 dimensional array, then an additional breakdown of recursive tree is not required. The way stored in the 2-dimensional array is directly accessed and returned. This saves a lot of time taken by the program.

3. **Tabulation:** Another way of solving the problem is without Recursion and rather using by loops. This is called tabulation, wherein each value is calculated until the required resultant value is reached via a loop. All these values are stored in a 2-dimensional array. The 2D is filled bottom-up, starting with the base cases and gradually building up to the final solution. The maximum value for a subproblem is either obtained by including the current item and subtracting its weight from the remaining capacity, or by excluding the item and considering the remaining items. The final answer is stored in the last cell of the table.

4. **Space Optimized Tabulation:** The previous solution makes a 2D array of size N*(W+1). Since each row relation depends on its previous row values, just 2 vectors can be made and these can be changed as the loop runs. This reduces the size of array to 2*(W+1), that is 2 arrays of size W+1.

5**. Space Optimized Tabulation (MORE!):** Another optimization that can be made to this code is to use a one-dimensional array instead of a two-dimensional array to store the values. This can be done by iterating through the weight array in reverse order and only updating the values of the array that are necessary. The code runs when traversing the weight array in the reverse order as well because the value of the arrays do not depend on the values to their right, so that can be another approach. This reduces the space complexity from O(2*W) to O(W), W is the maximum weight. Additionally, using an unordered_map or unordered_set can help reduce the constant factor in the time complexity for looking up values.

III. ANALYSIS

METHOD	TIME COMPLEXITY	SPACE COMPLEXITY
RECURSION	$O(2^N)$	O(N)
MEMOIZATION	O(NW)	O(NW)
TABULATION	O(NW)	O(NW)
SPACE OPTIMIZED TABULATION	O(NW)	O(W)
SPACE OPTIMIZED TABULATION (MORE)	O(NW)	O(W)

Where N is the number of items and W is the capacity of the knapsack.