Kerry Holmes

Dr. Sutton

Algorithms

12/14/2016

The 0/1 Knapsack Problem

The 0 or 1 knapsack problem can be thought of as a kind of cost benefit analysis. The idea is that there is a collection of items to choose from, each with an associated weight and value, and you have a knapsack with a limited amount of space to carry them. You need to determine the highest value you can acquire while not violating the weight constraints for your knapsack. This does not necessarily mean that there is only one combination of items that achieves this maximum, however there can be only one maximum value. There are multiple versions of this problem, in this case, the 0 or 1 refers to the number of the item available. This means you have exactly two options for each item, leave it behind or take one of it. It is impossible to have two of any item. From this, it is possible to recursively solve the problem by computing the greatest value of smaller knapsacks, and comparing the possible values with or without the current item’s weight and value. The recursion starts with a knapsack that could contain every item; this does not mean it can necessarily hold the combined weight. Instead it simply means that the knapsack could possibly hold every item if their weights allowed. There are three cases for the recurrence. The base case is that the next smaller knapsack to compute has no available items, or no more available weight. The next case is that the current item has a weight that is above the maximum weight. This case recurses, decreasing the available size of the knapsack by one, which essentially determines the optimal value for a knapsack one size smaller. The third case is that it is possible to take this item. In this case, the value returned is the higher value between the optimal knapsack one size smaller, and that knapsack with this item’s value included and its weight subtracted from the available pool. It should be apparent from this explanation that certain values are repeatedly calculated. Calculating a knapsack of size 6 requires calculating the knapsacks for sizes from 5 to 0. Calculating the 5 knapsack would require calculating knapsacks from 4 to 0. Already, the 4 to 0 range has been calculated twice. This repetition happens throughout the entire process. The optimal values of smaller knapsacks do not change; however, they are still recalculated at every step. These repetitions are called overlapping subproblems. This simply means that the problem is broken up but not into distinct partitions. The subproblems generated by breaking up this problem have calculations in common that are performed separately. This problem also exhibits what is called the optimal substructure property. Meaning that an optimal solution can be made using optimal solutions to subproblems. This should be apparent from the explanation of this problem, because at almost every step in the algorithm the value of the next smallest subproblem is used to determine a new optimal solution. These two properties make it possible to implement the algorithm using dynamic programming instead of recursion. The goal is to speed up the runtime by instead calculating and storing the solutions to subproblems once and storing them for later use instead of recalculating them every time. This implementation uses the dynamic programming approach to this problem.

As inputs, the program takes in two vectors. One vector will contain the values of each item, and the other will contain the weights. The algorithm treats each array index as an item. That is to say the first index of the values index indicates the value of the first item, whose weight is given by the first index in the weights array and so on. At the beginning, two matrices (two dimensional vectors) are created with dimensions equal to the number of items plus 1 and the maximum weight plus 1. One to hold the maximum values of the knapsacks, and one to mark whether of not an item is taken in this solution or not. It also creates a vector to store the final results of the algorithm. The program loops a total number of times equal to the number of items multiplied by the maximum weight of the knapsack plus 1. In every iteration, there one of three branches are executed. These branches are very similar to the recursive ones above. The first branch executes if the weight of the item is greater than the maximum weight. This case will write the previously calculated best value over the one at this location. The second branch will execute if the value of the knapsack is greater taking the item. It will update the optimal solution at this location and mark the item as taken in the second matrix. The final branch executes in every other circumstance and will write the previously calculated best value over the one at this location.

The program itself is very easy to use. You only need to place the source into a directory and build it using the CMake commands, “cmake” and “make”. The knapsack function is defined by the knapsack.cpp and knapsack.hpp files which are already included in the main.cpp file. Running the program as is will perform the knapsack problem using random integers for weights and values of the items, 11 items, and a maximum weight of ten. By changing the corresponding constant values for the number of items and weight, you can perform the knapsack problem on larger data sets. It is also possible to perform the test on a defined set of items by replacing the random generation of items.