Different Approaches to Solve Knapsack Problem

Aishwarya Rana
Texas State University

Abstract

The Knapsack problem is a combinatorial optimization problem where one has to maximize the benefit of objects in a knapsack without exceeding its capacity. It is an NP-complete problem and as such an exact solution for a large input is practically impossible to obtain.
The main goal of the paper is to present a comparative study of the brute force, dynamic programming, and greedy method. The paper discusses the complexity of each algorithm in terms of time and in terms of required programming efforts.

Introduction

In this project we are going to use Brute Force, Dynamic Programming, and Greedy Algorithms to solve the Knapsack Problem where one has to maximize the benefit of items in a knapsack without extending its capacity. The main goal of this project is to compare the time complexity of these algorithms and find the best one.

The Knapsack Problem (KP)

The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

Different Approaches

Brute Force

Brute force is a straightforward approach to solving a problem, usually directly based on the problem's statement and definitions of the concepts involved. If there are *n* items to choose from, then there will be 2^n possible combinations of items for the knapsack. An item is either chosen or not chosen. A bit string of 0's and 1's is generated which is of length *n*. If the i^{th} symbol of a bit string is 0, then the i^{th} item is not chosen and if it is 1, the i^{th} item is chosen.

ALGORITHM BruteForce (W, wt, val, n)
Finds the best possible combination subsets of items for the KP
Input: Maximum capacity M
Array of Weights Wt contains the weights of all items
Array of Values Val contains the values of all items and
Number of items n

Output: Set of best possible combination of items in the knapsack

```
for each combination of subsets i.e. $2^n$ do
     j=n
     sumWeight = 0
     sumValue =0
     while (set[j] != 0 and j > 0)
          set[j] =0
          j =j – 1
      set[j]= 1
      for k = 1 to n do
          if (set[k] = 1) then
          sumWt = sumWt + wt[k]
          sumVal= sumVal + val[k]
 if ((sumVal> val) AND (sumWt <= M)) then
          MaxVal = tempVal
          Maxwt = tempWt

     result = set
 return result
```

Complexity = O(n*2^n)

Therefore, the complexity of the Brute Force algorithm is *O (n2^n)*. Since the complexity of this algorithm grows exponentially, it can only be used for small instances of the KP. Otherwise, it does not require much programming effort to be implemented.

Dynamic Programming

Dynamic Programming is a technique for solving problems whose solutions satisfy recurrence relations with overlapping subproblems. Dynamic Programming solves each of the smaller subproblems only once and records the results in a table rather than solving overlapping subproblems repeatedly. The table is then used to obtain a solution to the original problem. The classical dynamic programming approach works bottom-up.

ALGORITHM Dynamic Programming (wt, val, M)

Input: Array Weights wt contains the weights of all items
 Array Values val contains the values of all Items
 and maximum Capacity M
 Array Table is initialized with 0s; it is used to store the results from the dynamic programming algorithm.

Output: The last value of array Table [n, M] contains the optimal solution of the problem for the given Capacity M

```
for i = 0 to n do
     for j = 0 to M
          if j < Weights[i]     then
               Table[i, j] = Table[i-1, j]
```

```
                Else
                        Table[i, j] = maximum { Table[i-1, j]
        Values[i] + Table[i-1, j – Weights[i]]
return Table[n, M]
```

In the implementation of the algorithm instead of using two separate arrays for the weights and the values of the items, we used one array Items of type item, where item is a structure with two fields: weight and value.

Complexity = O (n*M)

Thus, the complexity of the Dynamic Programming algorithm is *O (N*M)*. In terms of memory, Dynamic Programming requires a two-dimensional array with rows equal to the number of items and columns equal to the capacity of the knapsack. This algorithm is probably one of the easiest to implement because it does not require the use of any additional structures.

Greedy Algorithm

Greedy programming techniques are used in optimization problems. They typically use some heuristic or common-sense knowledge to generate a sequence of suboptimum that hopefully converges to an optimum value.

Possible greedy strategies to the 0/1 Knapsack problem:

1. Choose the item that has the maximum value from the remaining items; this increases the value of the knapsack as quickly as possible.
2. Choose the lightest item from the remaining items which uses up capacity as slowly as possible allowing more items to be stuffed in the knapsack.
   Choose the items with as high a value per weight as possible
   We implemented and tested all three of the strategies. We got the best results with the third strategy - choosing the items with as high value-to-weight ratios as possible.

ALGORITHM Greedy Algorithm (M, Wt, Val, n)

Input: Array Weights contains the weights of all items Array Values contains the values of all items
Output: Total load i.e. sum that it can take

Assume knapsack holds maximum capacity M and items have value val and weight wt and number of items N
Rank items by value/weight ratio: val / wt
Thus: val / wt ≥ val / wt, for all i ≤ j i.e. taking the maximum ratio
Consider items in order of decreasing ratio
Take as much of each item as possible

Complexity
Sorting by any advanced algorithm is O(NlogN) therefore, complexity of the greedy algorithm is, O(nlogn).

Test Cases:

For the comparison of these three methods of knapsack, we used different test cases having different number of items between 10 and 1000 and different maximum capacity between 10 and 1000 that a knapsack can hold. Then we generated random set of integers of weight and profit (i.e. value). Random weight is generated in such a way that their sum is greater or equal to twice the maximum capacity. Following table and graph shows the performance comparison of those three methods.

Table 1 Time Execution of all three methods of Knapsack with 10 different values of N and M

Number of Items N	Brute Force O(n*2^n)	Dynamic Programming O(n*M)	Greedy Algorithm O(nlogn)	Maximum Capacity M
10	52	0	3	120
22	57969	1	3	250
50		3	3	330
100		6	3	480
200		7	3	525
350		9	5	614
500		11	5	755
680		14	6	805
870		17	7	940
940		19	9	1000

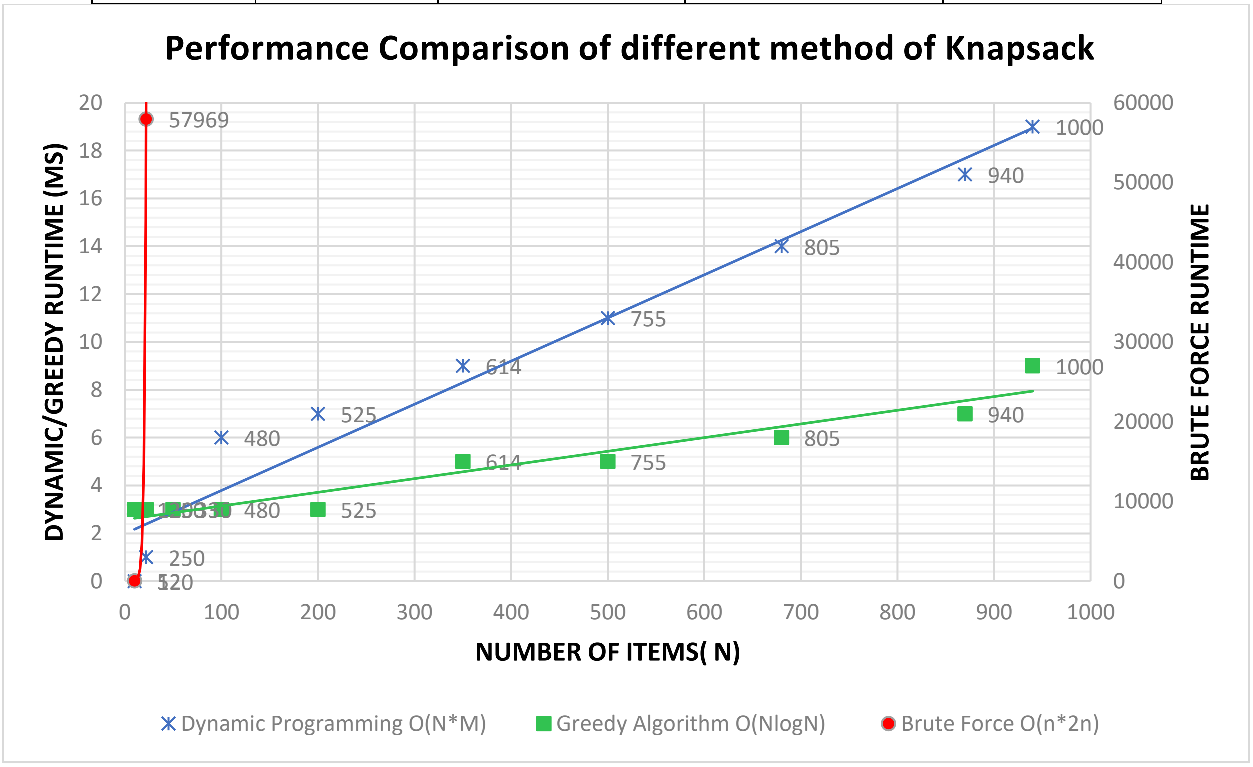

Figure 1 Performance comparison of three methods of knapsack in ten different cases

From above table and graph, greedy algorithm is more efficient than other two methods. Since, Brute force and Greedy algorithm's time complexity does not depend on maximum capacity, but dynamic programming depend on the maximum capacity. So, we can see that, brute force has the exponential growth, so it has longer execution time whereas dynamic and greedy has shorter execution time. Also, we can observe that when value on n and maximum capacity is small dynamic programming seems to be efficient, but as the number of items and maximum capacity increases the execution time also increases and in this case greedy execution time is efficient than dynamic algorithm. Brute Force's time complexity is exponentially growing, so takes very long time to execute. So, in my computer the larger n value for Brute force is 22 where it takes less than 3 minutes.

For further analysis of the result, we took constant number of items with different maximum capacity in increasing order and different number of items with constant maximum capacity. The results are as follows:

Table 2 Time execution for all three methods of Knapsack (N constant)

Number of Items N	Brute Force O(n*2^n)	Dynamic Programming O(n*M)	Greedy Algorithm O(nlogn)	Maximum Capacity M
20	8108	1	2	250
20	7969	2	3	500
20	8030	3	3	1000
20	7656	3	2	1500
20	7492	5	2	3000

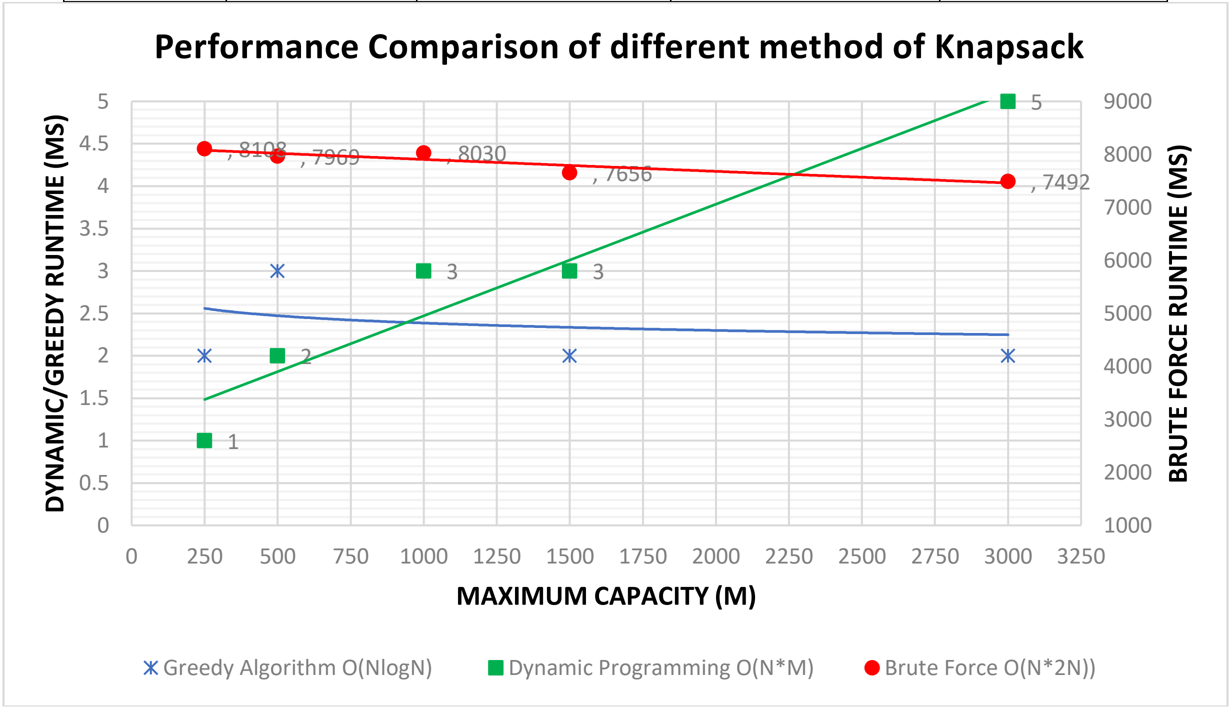

Figure 2 Performance comparison of three methods of knapsack in ten different cases where N is constant

Table 3 Time execution of three methods of Knapsack (M constant)

Number of Items N	Dynamic Programming O(n*M)	Greedy Algorithm O(nlogn)	Maximum Capacity M
25	2	2	500
50	2	3	500
100	5	2	500
200	6	2	500
300	7	4	500
500	8	4	500

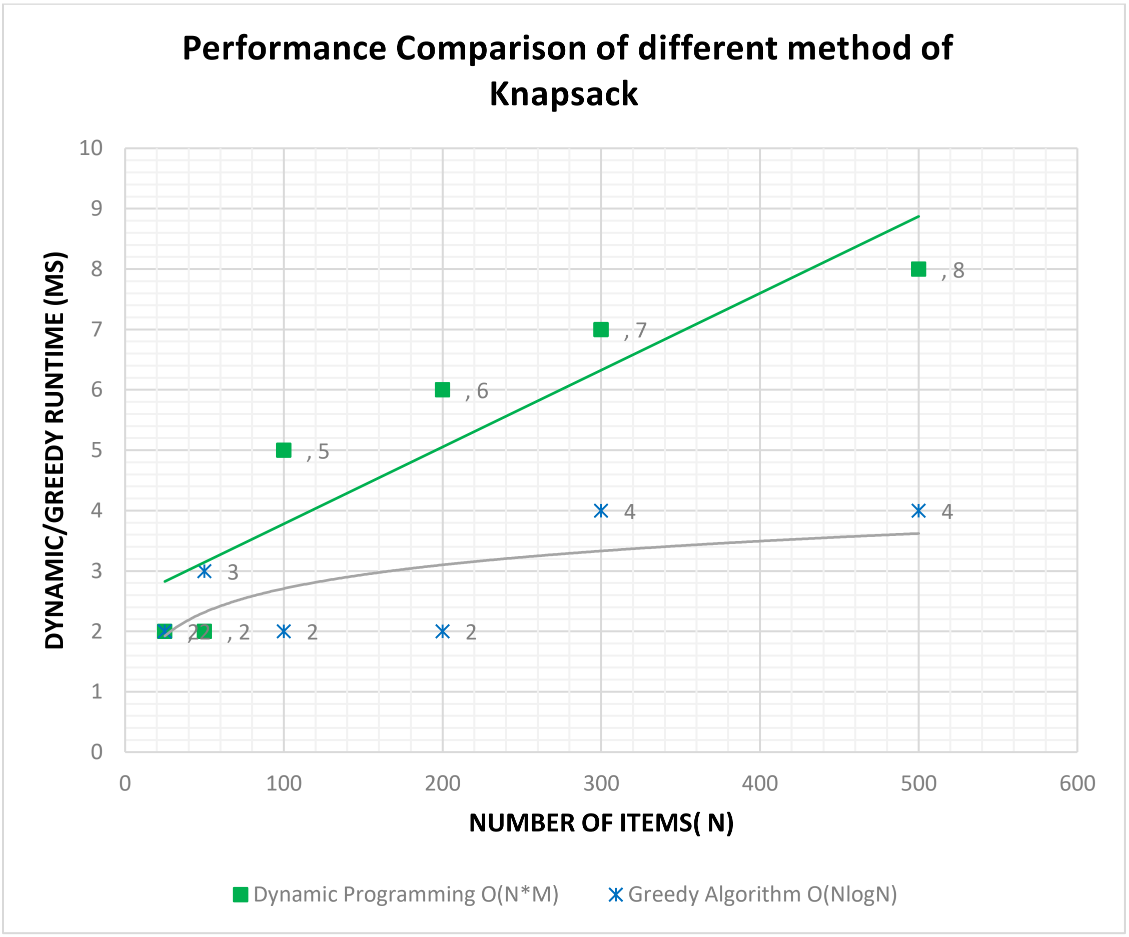

Figure 3 Performance comparison of three methods of knapsack in ten different cases where M is constant