Literature review on comparing between different approaches to solve the 0/1 knapsack problem

Bhumi K. Joshi
Information and Technology Department
L.D. College of Engineering Gujarat Technological University
Ahmedabad, Gujarat, India
email:joshibhumi16@gmail.com

ABSTRACT

The purpose of this paper is to analyze several algorithm design paradigms applied to a single problem - the 0/1 Knap- sack Problem. 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 algorithms. The paper discusses the complexity of each algorithm in terms of time requirements, and in terms of required programming efforts. Our experimental results show that the most promising approaches are dynamic programming.

1 INTRODUCTION

In this project we are going to use Brute Force, Dynamic Programming, genetic 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 results of these algorithms and find the best one.

1.1. The Knapsack Problem (KP)

The Knapsack Problem is an example of a combinatorial optimization problem, which seeks for a Best solution from among many other solutions. It is concerned with a knapsack that has positive integer volume (or capacity) V. There are n distinct items that may potentially be placed in the knapsack. Item i has a positive integer volume Vi and positive integer benefit Bi. In addition, there are Qi copies of item I available, where quantity Qi is a positive integer .Let Xi determines how many copies of item i are to be placed into the knapsack. The goal is to:

Maximize

$$\sum_{n=1}^{N} B_i X_i$$

Subject to the constraints

$$\sum_{n=1}^{N} V_i X_i \le V$$

And $0 \le X_i \le Q_i$

If one or more of the Qi is infinite, the KP is unbounded; otherwise, the KP is bounded [1]. The bounded KP can be either 0-1 KP or Multi constraint KP. If Qi = 1 for i = 1, 2,, N, the problem is a 0-1 knapsack problem In the current paper, we have worked on the bounded 0-1 KP, where we cannot have more than one copy of an item in the knapsack.

2 Different Approaches

2.1 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 2n possible combinations of items for the knapsack. An item is either chosen or not chosen. A bit stringof0'sand1's is generated which is of length n. If the ith symbol of a bit string is 0, then the ith item is not chosen and if it is 1, the ith item is chosen.

ALGORITHM

```
  Brute Force (Weights [1 N], Values [1 N], A[1N])
 Finds the best possible combination  of  items  for the KP
  Input: Array Weights contains the weights of all items
  Array Values contains the values of all items Array A initialized with 0s is used to generate the
bit strings
  Output: Best possible combination of items in the knapsack best_Choice[1 .. N]
  for i = 1 to $2^n$
  do j ← n
  temp_Weight ←0
  temp_Value ← 0
  while ( A[j] != 0 and j >0)
  A[j] ← 0 j ← j -1
  A[j] ← 1 for k ← 1 to n do
   if (A[k] = 1) then
  temp_Weight ← temp_Weight + Weights[k]
   temp_Value ← temp_Value + Values[k]
  if ((temp_Value >best_Value) AND (temp_Weight≤ Capacity)) then
  best_Value ← temp_Value
  best_Weight ← temp_Weight
   best_Choice ← A
  return best_Choice
```

Complexity

$$\sum_{i=0}^{2^n}[\sum_{j=n}^{1}+\sum_{k=1}^{n}]=\sum_{i=1}^{2^n}[1+..+1(\text{n times})\ +1+..+1(\text{n times})]$$
$$=(2n)*[1+1+1..+1](2^n\text{times})$$
$$=O(2n*2^n)$$
$$=O(n*2^n)$$

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 in order to be implemented. Besides the memory used to store the values and weights of all items, this algorithm requires a two one dimensional arrays (A[] and best_Choice[]).

2.2 Dynamic Programming

Dynamic Programming is a technique for solving problems whose solutions satisfy recurrence relations with overlapping sub problems. Dynamic Programming solves each of the smaller sub problems only once and records the results in a table rather than solving overlapping sub problems over and over again. The table is then used to obtain a solution to the original problem. The classical dynamic programming approach works bottom-up[2].

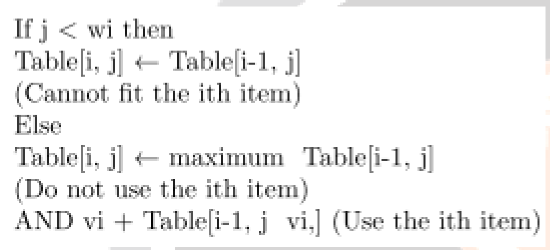
To design a dynamic programming algorithm for the 0/1 Knapsack problem, we first need to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller instances. Consider an instance of the problem defined by the first I items, 1≤i≤*N*

Weights w1,, wi

Values v1,, vi,

And knapsack capacity j,1 ≤ j≤ Capacity.

Let Table[i, j] be the optimal solution of this instance (i.e. the value of the most valuable sub- sets of the first i items that fit into the knapsack capacity of j). We can divide all the subsets of the first i items that fit the knapsack of capacity j into two categories subsets that do not include the ith item and subsets that include the ith item. This leads to the following recurrence:



```
If j < wi then
Table[i, j] ← Table[i-1, j]
(Cannot fit the ith item)
Else
Table[i, j] ← maximum  Table[i-1, j]
(Do not use the ith item)
AND vi + Table[i-1, j  vi,] (Use the ith item)
```

The goal is to find Table [N, Capacity] the maximal value of a subset of the knapsack. The two boundary conditions for the KP are:

- The knapsack has no value when there no items included in it (i.e. i = 0).

Table [0, j] = 0 for j≥0

- The knapsack has no value when its capacity is zero (i.e. j = 0), because no items can be included in it.

ALGORITHM

Dynamic Programming (Weights [1 N], Values [1 N], Table [0 ... N, 0 Capacity])

Input: Array Weights contains the weights of all items

Array Values contains the values of all items

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 (Table [N, Capacity]) contains the optimal solution of the problem for the given Capacity

```
for i = 0 to N do
   for j = 0 to Capacity if j <Weights[i] then
   Table[i, j] ← Table[i-1, j] else
   Table[i, j] ← maximum Table[i-1, j] AND
   Values[i] + Table[i-1, j Weights[i]]
   return          Table[N,          Capacity]
```

In the implementation of the algorithm in- stead 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.

To find which items are included in the optimal solution, we use the following algorithm:

```
n ← N c ←Capacity
Start at position Table[n, c]
While the remaining capacity is greater than 0 do
If Table[n, c] = Table[n-1, c] then
Item n has not been included in the optimal solution
Else Item n has been included in the optimal solution
Process Item n
Move one row up to n-1
Move to column c - weight (n)
```

Complexity

$$\sum_{i=0}^{N}\sum_{j=0}^{capacity} = \sum_{i=0}^{N}[1+1+1++1] \text{ (Capacity times)}$$

= Capacity * [1+1+1++1] (N times)
= Capacity * N
=O(N*Capacity)

Thus, the complexity of the Dynamic Programming algorithm is O (N*Capacity). 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.

2.3 Greedy Algorithm

```
ALGORITHM (Weights [1 N], Values [1 N])
Input: Array Weights contains the weights of all items
Array Values contains the values of all items
 Output:   Array Solution which indicates the items are included in the knapsack (1) or not(0)
Integer CumWeight
Compute the value-to-weight ratios ri=vi/wi, i = 1, , N, for the items given Sort the items
in non-increasing order of the value-to-weight ratios
   for all items do
         if the current item on the list fits into the knap- sack
         then place it in the knapsack else
         proceed to the next one
```

Complexity

1. Sorting by any advanced algorithm is O(NlogN)
2. $\sum_{i=0}^{N} 1$=[1+1+1.1] (N times)=N=O(N)

From (1) and (2), the complexity of the greedy algorithm is, O(NlogN) + O(N) = O(NlogN). In terms of memory, this algorithm only requires a one dimensional array to record the solution string.

3 Result Analysis

The comparative study of the brute force, greedy and dynamic programming algorithms shows that the complexities of these algorithms are as shown bellow in the table. Therefore we can say that Dynamic algorithm is having less time complexity than other mentioned.

Metric	Brute force	Dynamic	Greedy
Ex. Time	$O(n2^n)$	O(W*N)	O(NlogN)

If we implemented knapsack problem in c programming for different values of weight and profit. Result of both methods gives same optimal solution and different time.

Method	Input Data	Capacity	Profit	Time
Greedy	W={1,3,4,5} V={1,4,5,6}	9	11	15.86
	W={1,2,5,6,7} , V={1,6,18,20,28}	11	40	25.73
	W={10,20,30,40,50} V={10,30,64,50,60}	100	156	21.68
Dynamic	W={1,3,4,5} V={1,4,5,6}	9	11	17.69
	W={1,2,5,6,7} , V={1,6,18,20,28}	11	40	18.47
	W={10,20,30,40,50} V={10,30,64,50,60}	100	156	19.65

4 Conclusion

We conclude that for particular one knapsack problem we can implement two methods greedy and dynamic. But when we implemented both method for different data set values then we notice something like, we consider comparison parameter as optimal profit or total value for filling knapsack using available weight then dynamic and greedy both are gaining same profit. If we consider time then dynamic take less amount of time compare with greedy. So we can conclude that dynamic is better than greedy with respect to time.

5 References

[1] Gossett, Eric. Discreet Mathematics with Proof. New Jersey: Pearson Education Inc., 2003.
[2] Levitin, Anany. The Design and Analysis of Algorithms. New Jersey: Pearson Education Inc., 2003.
[3] Mitchell, Melanie. AnIntroduction to Genetic Algorithms. Massachusettss: The MIT Press, 1998.

[4] Obitko, Marek. Basic Description. IV. Genetic Algorithm. Czech Technical University (CTU). ¡http://cs.felk.cvut.cz/ xobitko/ga/gaintro.html¿