

Advanced Programming Report - Knapsack Problem

Florian Doffemont - Jeoffrey Pereira - Jocelyn Hauf - Arthur Micol

Contents

1	Introduction	4
1.1	About This Report	4
1.2	Knapsack Problem	4
1.3	Multidimensional Knapsack Problem	4
1.4	Multiple Knapsack Problem	4
2	Brute-Force Approach	5
2.1	Definition	5
2.2	Knapsack Problem	5
2.2.1	Idea	5
2.2.2	Algorithm	6
2.2.3	Analyze	8
2.3	Multidimensional Knapsack Problem	9
2.3.1	Idea	9
2.3.2	Algorithm	10
2.3.3	Analyze	12
3	Branch and Bound Approach	13
3.1	Definition	13
3.2	Knapsack Problem	13
4	Three Greedy Approach	14
4.1	Definition	14
4.2	Knapsack Problem - Greedy 1	14
4.3	Knapsack Problem - Greedy 2	14
4.4	Knapsack Problem - Greedy 3	14
5	Dynamic Programming Approach	15
5.1	Definition	15
5.2	Knapsack Problem	15
6	Fully Polynomial Time Approximation Scheme Approach	16
6.1	Definition	16
6.2	Knapsack Problem	16

7	Randomized Approach	17
7.1	Definition	17
7.2	Knapsack Problem	17
8	Ant Colony Approach	18
8.1	Definition	18
8.2	Knapsack Problem	18
9	Personal Approach	19
9.1	Definition	19
9.2	Knapsack Problem	19
10	Conclusion	20

1 Introduction

1.1 About This Report

This report is going to talk about, describe the Advanced Programming project that we have been given to do in groups of 4. The subject of the project is to implement the Knapsack Problem with different types of algorithms. The document will have as section the different types of algorithms that will be explained, shown, analyzed and compared with the others. Some sections will also have treated other problems (Multidimensional...). All external sources like datasets will be quoted in the last section (Sources)

1.2 Knapsack Problem

The Knapsack problem consists in choosing the best combination of objects (which have a weight and a value) in order to have a maximum value in the backpack (which has a maximum weight not to exceed). It is a NP complete problem.

1.3 Multidimensional Knapsack Problem

Here, the goal is the same as the simple Knapsack problem, with the difference that we have to manage not only one dimension of the backpack (the weight) but several dimensions (weight, volume, etc..), we can see it as a vector with n dimensions. So an object will have a value and n dimensions. It will be necessary to have the best combination of object to have the best value in the backpack without that the objects "overflow" of all the dimensions of the backpack. Of course, the "size" of all dimensions are predefined and should not be exceeded.

1.4 Multiple Knapsack Problem

Here the problem consists in considering that one should not fill only one backpack, but several which have a different size. An object cannot be in several backpacks. So we have to find the best combination so that the value of the objects in the backpacks added together is maximum, without exceeding the capacity of the backpacks.

2 Brute-Force Approach

Author : Florian Doffemont

2.1 Definition

The Brute-Force method consists in trying all possible combinations of a problem. For example, to crack a password, we try all possibilities (a, b, ... aa, ab...).

For the Knapsack problem, we will try all the possibilities of objects in the backpack.

2.2 Knapsack Problem

2.2.1 Idea

First of all, we need an *Items* structure that will define an object that can go in the backpack. The structure is defined as follows:

```
class item:
    def __init__(self, pos, value, weight):
        self.pos = int(pos)
        self.value = float(value)
        self.weight = float(weight)
        self.ratio = float(value) / weight
```

The structure has 4 fields, *pos* which will indicate the position in a tree (useless in this problem), *value* which will store the value of the object, *weight* which will store the weight of the object and *ratio* which will store the ratio value/weight of the object (useless here).

After that, we need a function to read the data sets (taken from the site: http://artemisa.unicauca.edu.co/johnnyortega/instances_01_KP/).

With this function we will store in a list of *Items* the *Items* read and created. We will at the same time, store the maximum size of the backpack.

Now we just have to send the list of *Items* and the maximum size of the backpack to the *Brute-Force* algorithm.

2.2.2 Algorithm

The function is defined as follows:

- The function takes the list of items and the capacity of the knapsack as parameters.
- It returns the list of items that are in the knapsack with the optimal value
- It returns an empty list if no solution is found

So we have 2 parameters for the function :

```
def bruteForce(capacity, items):
```

At the beginning of the function, we will create 5 variables:

- *start_time* : Calculate the time of execution
- *combinaisons* : Create a list of all possible combinations of items
- *combinaisonsPossibles* : Create a list of all possible combinations of items that fit in the knapsack
- *combinaisonsPossiblesMax* : Create a list of all possible combinations of items that fit in the knapsack and that have the highest value
- *valeurMax* : We initialize the highest combination value to 0

```
start_time = time.time()
combinaisons = []
combinaisonsPossibles = []
combinaisonsPossiblesMax = []
valeurMax = 0
```

Now you have to store all the possibilities (correct or not) in the *combinaisons* list. We know that all possible combinations of items are 2^n . We use the binary system to create all possible combinations of items. For example, if we have 3 items, we will have 8 possible combinations of items : 000, 001, 010, 011, 100, 101, 110, 111 ...

000 means that we don't take any item, 001 means that we take the first item, 010 means that we take the second item, 011 means that we take the first and second items etc...

```
for i in range(2**len(items)):
    combinaisons.append([])
    for j in range(len(items)):
        if (i >> j) % 2 == 1:
            combinaisons[i].append(items[j])
```

With this list, we check if the combinations of items fit in the knapsack. If they fit, we add them to the list of possible combinations:

```
for combinaison in combinaisons:
    poids = 0
    for item in combinaison:
        poids += item.weight
    if poids <= capacity:
        combinaisonsPossibles.append(combinaison)
```

And finally, with this list containing the valid solutions, we check which combination of items has the highest value. If they have the same value, we add them to the list of possible combinations with the highest value. If they have a higher value, we empty the list of possible combinations with the highest value and we add the new combination.

If they have a lower value, we do nothing. We also update the highest value and the list of items that make up the highest value combination :

```
for combinaison in combinaisonsPossibles:
    valeur = 0
    for item in combinaison:
        valeur += item.value
    if valeur > valeurMax:
        valeurMax = valeur
        combinaisonsPossiblesMax = [combinaison]
    elif valeur == valeurMax:
        combinaisonsPossiblesMax.append(combinaison)
```

We obtain the list of items with the highest value and return it with the execution time :

```
end_time = time.time()
execution_time = end_time - start_time

return combinaisonsPossiblesMax, execution_time
```

2.2.3 Analyze

The complexity is of $O(n * 2^n)$ because we have as the "biggest" loop an n loop in a 2^n loop, so $n * 2^n$

So it means that if we have 10 items, it will take 10,240 operations to find the solution

If we have 20 items, it will take 20,971,520 operations to find the solution

If we have 30 items, it will take 32,212,254,720 operations to find the solution

So it is very slow, I think it is not recommended to use it with more than 20 items, but it always gives the optimal solution

2.3 Multidimensional Knapsack Problem

2.3.1 Idea

First of all, like the simple Knapsack Problem, we need an *ItemMD* structure that will define an object that can go in the backpack. The structure is defined as follows:

```
class itemMD:
    def __init__(self,pos,value,weight, ratio):
        self.pos = pos # Position in the list of item
        self.value = value # Value of the item
        self.weight = weight # weight(vector) of the item for each
                             dimension
        self.ratio = ratio # List of ratio for each dimension
```

The structure has 4 fields, *pos* which will indicate the position in a tree (useless in this problem), *value* which will store the value of the object, *weight* which will store the list of weight of the object for each dimension and *ratio* which will store the ratio value/weight of the object for each dimensions (useless here).

Then we have to create a new function to read the data from the datasets, because it is not the same structure as for the Simple Knapsack. This function will return :

- The list of ItemMD read and created
- The number of dimension of the backpack
- The list of "max weight" of each dimension
- The optimal value to find, which is provided in the file

Now we just have to send the list of *ItemMD*, the list of *weight* for each dimensions and the number of *dimension* to the *Brute-Force-MD* algorithm.

2.3.2 Algorithm

The function is defined as follows:

- It takes the list of itemMD, the list of capacities of the knapsacks and the number of dimensions as parameters.
- It returns the best value, the list of items that can fit in all the dimensions of the knapsack and the time of execution
- It returns an empty list if no solution is found

So we have 3 parameters for the function :

```
def bruteForceMD(itemMD, capacities, nbrDimensions):
```

At the beginning of the function, its the same than the Simple Knapsack, we will create 5 variables:

- *start_time* : Calculate the time of execution
- *combinaisons* : Create a list of all possible combinations of items
- *combinaisonsPossibles* : Create a list of all possible combinations of items that fit in the knapsack
- *combinaisonsPossiblesMax* : Create a list of all possible combinations of items that fit in the knapsack and that have the highest value
- *valeurMax* : We initialize the highest combination value to 0

```
start_time = time.time()
combinaisons = []
combinaisonsPossibles = []
combinaisonsPossiblesMax = []
valeurMax = 0
```

Now we have to store all the possibilities, we use the same method as for the Simple Knapsack :

```
for i in range(2**len(items)):
    combinaisons.append([])
    for j in range(len(items)):
        if (i >> j) % 2 == 1:
            combinaisons[i].append(items[j])
```

This is where the algorithm will change compared to the previous one, we will put in the *combinaisonsPossibles* list all the possibilities that fit in the backpack. Contrary to the previous algorithm that checked if the combination of objects fit in the backpack, here we will check that the list of objects fit not only in one dimension, but in the d dimensions:

```
for combinaison in combinaisons:
    poids = [0] * nbrDimensions
    for item in combinaison:
        for i in range(nbrDimensions):
            poids[i] += item.weight[i]
    if poids <= capacities:
        combinaisonsPossibles.append(combinaison)
```

And finally, with this list containing the valid solutions, we will do as for the previous algorithm, keep only the combination(s) that have the highest value :

```
for combinaison in combinaisonsPossibles:
    valeur = 0
    for item in combinaison:
        valeur += item.value
    if valeur > valeurMax:
        valeurMax = valeur
        combinaisonsPossiblesMax = [combinaison]
    elif valeur == valeurMax:
        combinaisonsPossiblesMax.append(combinaison)
```

We obtain the list of items with the highest value and return it with the execution time and the optimal value found :

```
end_time = time.time()
execution_time = end_time - start_time

return combinaisonsPossiblesMax, execution_time, valeurMax
```

2.3.3 Analyze

The complexity is of $O(n * 2^n)$ because we have as the "biggest" loop an n loop in a 2^n loop, so $n * 2^n$

So it means that if we have 10 items, it will take 10,240 operations to find the solution

If we have 20 items, it will take 20,971,520 operations to find the solution

If we have 30 items, it will take 32,212,254,720 operations to find the solution

So it is very slow, I think it is not recommended to use it with more than 20 items, but it always gives the optimal solution

3 Branch and Bound Approach

Author : Jeffrey Pereira

3.1 Definition

3.2 Knapsack Problem

4 Three Greedy Approach

Author : Jocelyn Hauf

4.1 Definition

4.2 Knapsack Problem - Greedy 1

4.3 Knapsack Problem - Greedy 2

4.4 Knapsack Problem - Greedy 3

5 Dynamic Programming Approach

Author : Arthur Micol

5.1 Definition

5.2 Knapsack Problem

6 Fully Polynomial Time Approximation Scheme Approach

Author :

6.1 Definition

6.2 Knapsack Problem

7 Randomized Approach

Author :

7.1 Definition

7.2 Knapsack Problem

8 Ant Colony Approach

Author :

8.1 Definition

8.2 Knapsack Problem

9 Personal Approach

Author :

9.1 Definition

9.2 Knapsack Problem

10 Conclusion