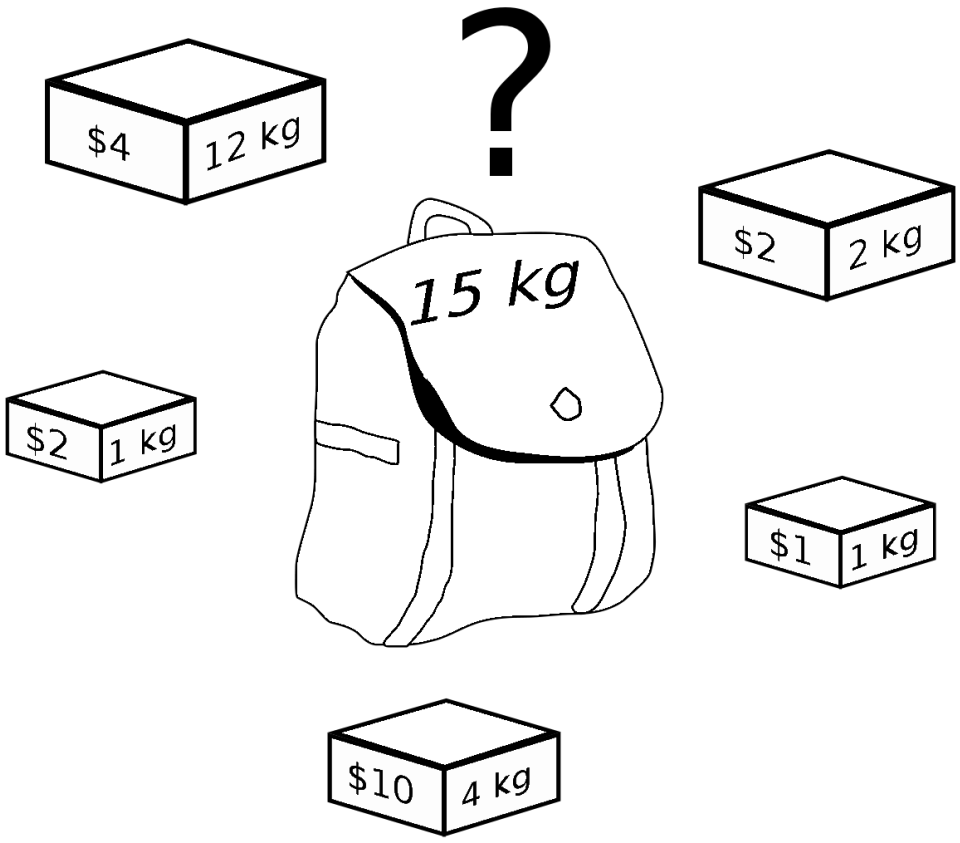
Lab VII

Knapsack Problem



Course: CS 2302

Section: 12:00 p.m. – 1:20 p.m.

Author: Oswaldo Escobedo

Instructor: Dr. Fuentes

TA: Harshavardhini Bagavathyraj

Introduction

The knapsack problem consists that a thief carries with him a backpack that can only carry a weight W and must pay a debt D, so he enters a house and finds several items that have a value v (with which he can pay his debt) and a weight w. If there is an answer, it will be possible for the thief to steal items worth D or more while having a weight of W or less.

In other words, it is a combinatorial optimization problem, that is, it searches for the best solution among a finite set of possible solutions to a problem. Therefore, the objective of this laboratory is to create a program that solves the knapsack problem using three different algorithms which are the following:

Backtracking algorithm: this algorithm solves the problem recursively by trying all possible combinations, in this case it will try all possible combination from each item's value and weight. The problem with this algorithm is that it only works with smallest instances of a problem, that is because its time complexity is $O(2^n)$. Therefore, it will take a long time to solve a problem, may be dozens of years!!

Greedy Algorithm: this algorithm always makes the choice that seems to be the best at that moment. It consists of finding a globally-optimal solution. In this problem, we will sort the items in descending order by using the value to weight ratio (vwr = v/w). Then go through the list and add item[i] to the knapsack if there is still room.

Randomize Algorithm: is an algorithm that uses randomness as part of its logic. It is used to reduce the running time when solving a task. In this case, it will be used to randomly sort the list of items [[v[0],w[0]] … [v[i],w[i]]], then go through the list and add an item to the knapsack if there ir still room for it, if after finishing the first iteration the debt D has not been paid, then try another random order, this will continue for a fixed number of times.

Proposed Solution, Design and Implementation

Knapsack_bt (W,D,v,w)

First of all, start by creating the first method which will solve the problem using the backtraking algorithm. For this the teacher gave us a snippet of the first lines of code:

```
def knapsack(W,D,v,w):
    # W is the remaining knapsack capacity, D is the remaining debt, v and w are lists of the
    # same length where item 0 has value v[0] and weight w[0], item 1 has value v[1] and
    # weight w[1], and so on.
    if W<0: # knapsack capacity exceeded
       return False
    if D<0: # debt paid
       return True
```

Our task, then was to check if the code provided by Dr. Fuentes was correct or not and to add the recursive calls. First, we checked if the code was well designed and that it worked properly. However, after analyzing for some minutes, we discovered that the code breaks the design rule, which states that even if the recursive call works, the original call doesn't work. In this case (problem), our priority will always be to verify if the debt has already been paid, so we decided to switch the place of the two base cases, that way the method would prioritize the verification of whether the debt has been paid or not. Second, we started doing the recursive call:

- In the first call I decided to add the weight of the first item to the backpack and decrease the value of the debt that I owe with the value of the first item. Then I decided to continue with the value and weight of the next article. The following recursive call describes the following:
    - Knapsack_bt (W-w[0],D-v[0],v[1:],w[1:])
- On the second recursive call I decided not to add the first article to the backpack, but instead I decided to analyze whether the next article is more convenient to add to the knapsack or not. The following recursive call describes the following:
    - Knapsack_bt (W,D,v[1:],w[1:])

With this in mind, I continued to create the recursive statement, in which I added the above recursive calls and added a Boolean 'or' between them. This because we only want to know if at least one combination of the set provides us with a solution to the problem.

Knapsack_g(W,D,v,w)

This method, like the previous one, receives 2 integers (one representing the weight limit ‘W’of the backpack and the other representing the debt ‘D’ that we owe) and two lists (one list contains the weight ‘w’ of each item and the other list represents the value ‘v’ of each item).

To begin solving this problem, we will have to create a list of lists, in which each list will contain the ratio of each item, that is, v[i]/w[i], the value and weight of each element. In short, the 2d list will look like this:

- [[v[i]/w[i]max, v[i], w[i]], … [[v[i]/w[i]min, v[i], w[i]]]

We will then sort that 2d list in descending order based on the ratio v[i]/w[i]. After that, we will create a for loop that will stop when we have visited all the items in the 2d list.

Then, we will create two variables in which one will store the value and the other the weight of the item we are currently in. This to keep the method organized, clean and readable. Next, we'll add an if statement that will be true if the backpack still has room for the weight of the current item, that is, W> = w [i], inside this condition we will decrement the room of the knapsack and we will decrement the debt that we owe.

When we leave the for loop we will put an if condition, which will be true and will return true if we have finished paying the debt. Otherwise, it will return false, meaning that we have not been able to pay the debt. Note: It should be noted that there will be cases in which this algorithm will not find the solution even if one exists.

Knapsack_r(W,D,v,w)

This method, like the previous ones, will receive the same parameters, however we will add an extra one, which will be called attempts, its purpose will be mentioned later.

To begin solving this problem, we will have to create a list of lists, in which each list will contain the value and weight of each element. In short, the 2d list will look like this:

- [[v[0], w[0]], … [v[i], w[i]]]

Then we will create a for loop that will stop for a fixed number of times, that is, it will stop until the last attempt is reached, that’s the purpose of the variable ‘attempt’, so that it is up to us to

decide how many iterations of this for loop will do. Note: the more ‘attempts’ we give, the more accurate the result will be but the slower the running time it will be.

Inside this for loop we will randomly sort the 2d list by using random.shuffle(), then we will create two variable that will contain the values of D and W, this to keep and not lost the original values each iteration on the nested for loop. After, we will create a nested for loop that will stop until we have reach the last item, inside this loop we will create two variables: one will contain the value of the current item and the other will contain the weight of the current item.

Next, we'll add an if statement that will be true if the backpack still has room for the weight of the current item, that is, W> = w [i], inside this condition we will decrement the room of the knapsack and we will decrement the debt that we owe.

In addition, inside this nested loop we will add a condition which will be true and will return true if we have finished paying the debt. Finally, if we finished iterating the first for loop then it means that there was no possible combination that could pay the debt D and carry W weight, so we return False.

Experimental Results

Backtrack Algorithm Results

```
print('~~~~~~~~~~ Problem 1 ~~~~~~~~~~~~~~~')
W = 35
D = 270
w = [10,  6, 10,  6, 14,  8,  5, 13,  4,  1]
v = [39, 47, 47, 29, 71, 22, 50, 29, 51, 20]


Backtracking -->  False

print('~~~~~~~~~~ Problem 2 ~~~~~~~~~~~~~~~')
W =  10
D = 130
w = [10,  6, 10,  6, 14,  8,  5, 13,  4,  1]
v = [39, 47, 47, 29, 71, 22, 50, 29, 51, 20]


Backtracking -->  False
```

```
print('~~~~~~~~~~ Problem 3 ~~~~~~~~~~~~~~~~')
W = 102
D = 404
w = [10,  8,  5,  6,  9, 13, 13, 14, 13, 14,  6, 11, 12,  5, 13, 11,  9,
       10, 14,  9]
v = [47, 15,  7, 17, 29, 12, 45, 24, 26, 10, 37, 38, 14, 35, 44, 37, 27,
       45, 36, 40]
```

```
Backtracking -->  True
```

```
print('~~~~~~~~~~ Problem 4 ~~~~~~~~~~~~~~~~')
W = 150
D = 600
w = [10, 14,  4,  5,  8, 12,  5,  7,  7, 11,  9,  5, 10, 14,  4,  4, 14,
        7,  8,  9]
v = [39, 49, 47, 40, 20, 27, 31, 34, 17, 10, 29, 36, 41, 48, 45, 24, 15,
       17, 14, 40]
```

```
Backtracking -->  False
```

```
print('~~~~~~~~~~ Problem 5 ~~~~~~~~~~~~~~~~')
W = 200
D = 960
w = [ 8, 13, 13,  9,  5, 14, 13,  4,  8,  7, 13,  8, 12,  9, 13,  8,  5,
        9,  5,  7,  4,  7, 13, 13,  6,  8,  4,  5,  9, 10,  5,  4,  6, 10,
        7,  9, 13, 14, 12,  5, 10,  7,  9, 12,  9, 10,  5,  8, 11,  9]
v = [21, 26, 25, 23, 42, 32, 45, 33, 40, 20, 44, 13,  9, 31, 47, 21, 31,
       18, 41, 36, 32, 43, 20, 40, 23, 16, 10, 44, 38,  6, 11, 13, 43,  7,
       35, 21,  7, 25, 47, 34, 33, 46, 26, 17, 23, 28, 42, 16, 28, 30]
```

Running Time is very long, due to the big size of the lists.

As you can see, this algorithm will always find a solution if it exists. However, this will only work for small instances of a problem, for example, in problem 5 I ran the code for a day and still did not return a result, this is because the complexity of this algorithm is O (2 ^ n), so we may have to wait weeks, months, even years to obtain a result.

Greedy Algorithm Results

```
print('~~~~~~~~~~ Problem 1 ~~~~~~~~~~~~~~~~')
W = 35
D = 270
w = [10,  6, 10,  6, 14,  8,  5, 13,  4,  1]
v = [39, 47, 47, 29, 71, 22, 50, 29, 51, 20]
```

```
Greedy -->  False
```

```
print('~~~~~~~~~~ Problem 2 ~~~~~~~~~~~~~~~~')
W =  10
D = 130
w = [10,  6, 10,  6, 14,  8,  5, 13,  4,  1]
v = [39, 47, 47, 29, 71, 22, 50, 29, 51, 20]
```

```
Greedy -->  False
```

```
print('~~~~~~~~~~ Problem 3 ~~~~~~~~~~~~~~~~')
W = 102
D = 404
w = [10,  8,  5,  6,  9, 13, 13, 14, 13, 14,  6, 11, 12,  5, 13, 11,  9,
       10, 14,  9]
v = [47, 15,  7, 17, 29, 12, 45, 24, 26, 10, 37, 38, 14, 35, 44, 37, 27,
       45, 36, 40]
```

```
Greedy -->  True
```

```
print('~~~~~~~~~~ Problem 4 ~~~~~~~~~~~~~~~~')
W = 150
D = 600
w = [10, 14,  4,  5,  8, 12,  5,  7,  7, 11,  9,  5, 10, 14,  4,  4, 14,
        7,  8,  9]
v = [39, 49, 47, 40, 20, 27, 31, 34, 17, 10, 29, 36, 41, 48, 45, 24, 15,
       17, 14, 40]
```

```
Greedy -->  False
```

```
print('~~~~~~~~~~ Problem 5 ~~~~~~~~~~~~~~~~')
W = 200
D = 960
w = [ 8, 13, 13,  9,  5, 14, 13,  4,  8,  7, 13,  8, 12,  9, 13,  8,  5,
        9,  5,  7,  4,  7, 13, 13,  6,  8,  4,  5,  9, 10,  5,  4,  6, 10,
        7,  9, 13, 14, 12,  5, 10,  7,  9, 12,  9, 10,  5,  8, 11,  9]
v = [21, 26, 25, 23, 42, 32, 45, 33, 40, 20, 44, 13,  9, 31, 47, 21, 31,
       18, 41, 36, 32, 43, 20, 40, 23, 16, 10, 44, 38,  6, 11, 13, 43,  7,
       35, 21,  7, 25, 47, 34, 33, 46, 26, 17, 23, 28, 42, 16, 28, 30]
```

```
Greedy -->  False
```

If we analyze this algorithm, it returns the expected result for all 5 test cases, however, sometimes the ideal result will not return if it exists, for example, imagine that with a single

article we can fill the backpack and pay our debt, but due to the way this algorithm works, it may look for another solution not so efficient and it will fill up the knapsack quickly, and thus giving us an incorrect result.

Random Algorithm Results

```
print('~~~~~~~~~~ Problem 1 ~~~~~~~~~~~~~~~~')
W = 35
D = 270
w = [10,  6, 10,  6, 14,  8,  5, 13,  4,  1]
v = [39, 47, 47, 29, 71, 22, 50, 29, 51, 20]


Random --> False

print('~~~~~~~~~~ Problem 2 ~~~~~~~~~~~~~~~~')
W =  10
D = 130
w = [10,  6, 10,  6, 14,  8,  5, 13,  4,  1]
v = [39, 47, 47, 29, 71, 22, 50, 29, 51, 20]


Random --> False

print('~~~~~~~~~~ Problem 3 ~~~~~~~~~~~~~~~~')
W = 102
D = 404
w = [10,  8,  5,  6,  9, 13, 13, 14, 13, 14,  6, 11, 12,  5, 13, 11,  9,
       10, 14,  9]
v = [47, 15,  7, 17, 29, 12, 45, 24, 26, 10, 37, 38, 14, 35, 44, 37, 27,
       45, 36, 40]


Random --> True


Random --> False

print('~~~~~~~~~~ Problem 4 ~~~~~~~~~~~~~~~~')
W = 150
D = 600
w = [10, 14,  4,  5,  8, 12,  5,  7,  7, 11,  9,  5, 10, 14,  4,  4, 14,
        7,  8,  9]
v = [39, 49, 47, 40, 20, 27, 31, 34, 17, 10, 29, 36, 41, 48, 45, 24, 15,
       17, 14, 40]


Random --> False
```

```
print('~~~~~~~~~~ Problem 5 ~~~~~~~~~~~~~~~')
W = 200
D = 960
w = [ 8, 13, 13,  9,  5, 14, 13,  4,  8,  7, 13,  8, 12,  9, 13,  8,  5,
        9,  5,  7,  4,  7, 13, 13,  6,  8,  4,  5,  9, 10,  5,  4,  6, 10,
        7,  9, 13, 14, 12,  5, 10,  7,  9, 12,  9, 10,  5,  8, 11,  9]
v = [21, 26, 25, 23, 42, 32, 45, 33, 40, 20, 44, 13,  9, 31, 47, 21, 31,
       18, 41, 36, 32, 43, 20, 40, 23, 16, 10, 44, 38,  6, 11, 13, 43,  7,
       35, 21,  7, 25, 47, 34, 33, 46, 26, 17, 23, 28, 42, 16, 28, 30]
```

```
Random --> False
```

This algorithm, as previously stated, is not as effective as the previous ones, since it is based on the random factor. Because of this, although there are many 'attempts' in the iterations, most of the time a correct or false result will be returned to us, such is the case of problem 3, which returned the correct result 80% of the time and the 20% returned an incorrect result. On the other hand, there is the fact that this algorithm will always solve the problem in a faster way than using backtracking.

Conclusion

In conclusion, this algorithm helped me to practice and improve my understanding of the different types of algorithms that exist. Well this made me understand that there are going to be algorithms that will solve a problem very quickly but will not always return the correct result, while there will be others that will always return the correct result even if it takes years to do so. It is a very important and interesting topic, as this will make future personal projects decide which is the best to use and why.

Academic Honesty Certification

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.

Oswaldo Escobedo

Appendix

```
#   Course: CS 2302
#   Assignment: Lab VII
#   Author: Oswaldo Escobedo
#   Instructor: Dr. Fuentes
#   TA: Harshavardhini Bagavathyraj
#   Date of Last Modification: 05/11/2020
#   Purpose of the Program: To implement three different algorithms
#   to solve the knapsack problem.

import numpy as np
import random

def knapsack_bt(W,D,v,w):
# W is the remaining knapsack capacity, D is the remaining debt, v and w are lists of the
# same length where item 0 has value v[0] and weight w[0], item 1 has value v[1] and
# weight w[1], and so on.
    if len(v) <= 0 or len(w) <= 0:
        return False
    if D <= 0: # debt paid
        return True
    if W <= 0: # knapsack capacity exceeded
        return False
    return knapsack_bt(W-w[0],D-v[0],v[1:],w[1:]) or knapsack_bt(W,D,v[1:],w[1:])

def knapsack_g(W,D,v,w):
    ratio = [ [v[i]/w[i],v[i],w[i]] for i in range(len(v))] # List that contains [Ratio (v[i]/w[i]), Value, Weight]
    vwr = sorted(ratio,reverse=True) # Sort in descending order
```

```
    #knapsack = [] # List that contains value[i] of item[i]
    for i in range(len(vwr)):
        weight = vwr[i][2]
        value = vwr[i][1]
        if W >= weight: # Add item to the knapsack if there is still room in W
            #knapsack.append(value)
            W -= weight # Decrease room of W
            D -= value # Decrease debt of D
    if D <= 0: # Debt paid
        return True
    return False

def knapsack_r(W,D,v,w,attempts=50000):
    items = [ [v[i],w[i]] for i in range(len(v))]
    for i in range(attempts):
        random.shuffle(items) # Randomly sort the items
        #knapsack = []
        Dc = D # Copy the value of D
        Wc = W # Copy the value of W
        for j in range(len(items)):
            weight = items[j][1]
            value = items[j][0]
            if Wc >= weight: # Add item to the knapsack if there is still room in W
                #knapsack.append(value)
                Wc -= weight # Decrease room of W
                Dc -= value # Decrease debt of D
            if Dc <= 0: # Debt paid
                return True
    return False
```

```
if __name__ == "__main__":

    # Sample knapsack datasets
    print('~~~~~~~~~~ Problem 1 ~~~~~~~~~~~~~~~')
    W = 35
    D = 270
    w = [10,  6, 10,  6, 14,  8,  5, 13,  4,  1]
    v = [39, 47, 47, 29, 71, 22, 50, 29, 51, 20]
    print('W is: ',W)
    print('w is: ',sum(w))
    print('D is: ',D)
    print('v is: ',sum(v))
    print('Backtracking --> ',knapsack_bt(W,D,v,w))
    print('Greedy --> ',knapsack_g(W,D,v,w))
    print('Random -->',knapsack_r(W,D,v,w))

    print('~~~~~~~~~~ Problem 2 ~~~~~~~~~~~~~~~')
    W =  10
    D = 130
    w = [10,  6, 10,  6, 14,  8,  5, 13,  4,  1]
    v = [39, 47, 47, 29, 71, 22, 50, 29, 51, 20]
    print('W is: ',W)
    print('w is: ',sum(w))
    print('D is: ',D)
    print('v is: ',sum(v))
    print('Backtracking --> ',knapsack_bt(W,D,v,w))
    print('Greedy --> ',knapsack_g(W,D,v,w))
    print('Random -->',knapsack_r(W,D,v,w))
```

```
print('~~~~~~~~~~ Problem 3 ~~~~~~~~~~~~~~~')
W = 102
D = 404
w = [10,  8,  5,  6,  9, 13, 13, 14, 13, 14,  6, 11, 12,  5, 13, 11,  9,
     10, 14,  9]
v = [47, 15,  7, 17, 29, 12, 45, 24, 26, 10, 37, 38, 14, 35, 44, 37, 27,
     45, 36, 40]
print('W is: ',W)
print('w is: ',sum(w))
print('D is: ',D)
print('v is: ',sum(v))
print('Backtracking --> ',knapsack_bt(W,D,v,w))
print('Greedy --> ',knapsack_g(W,D,v,w))
print('Random -->',knapsack_r(W,D,v,w))

print('~~~~~~~~~~ Problem 4 ~~~~~~~~~~~~~~~')
W = 150
D = 600
w = [10, 14,  4,  5,  8, 12,  5,  7,  7, 11,  9,  5, 10, 14,  4,  4, 14,
      7,  8,  9]
v = [39, 49, 47, 40, 20, 27, 31, 34, 17, 10, 29, 36, 41, 48, 45, 24, 15,
     17, 14, 40]
print('W is: ',W)
print('w is: ',sum(w))
print('D is: ',D)
print('v is: ',sum(v))
print('Backtracking --> ',knapsack_bt(W,D,v,w))
print('Greedy --> ',knapsack_g(W,D,v,w))
```

```
print('Random -->',knapsack_r(W,D,v,w))


print('~~~~~~~~~ Problem 5 ~~~~~~~~~~~~~~')
W = 200
D = 960
w = [ 8, 13, 13,  9,  5, 14, 13,  4,  8,  7, 13,  8, 12,  9, 13,  8,  5,
       9,  5,  7,  4,  7, 13, 13,  6,  8,  4,  5,  9, 10,  5,  4,  6, 10,
       7,  9, 13, 14, 12,  5, 10,  7,  9, 12,  9, 10,  5,  8, 11,  9]
v = [21, 26, 25, 23, 42, 32, 45, 33, 40, 20, 44, 13,  9, 31, 47, 21, 31,
      18, 41, 36, 32, 43, 20, 40, 23, 16, 10, 44, 38,  6, 11, 13, 43,  7,
      35, 21,  7, 25, 47, 34, 33, 46, 26, 17, 23, 28, 42, 16, 28, 30]
print('W is: ',W)
print('w is: ',sum(w))
print('D is: ',D)
print('v is: ',sum(v))
#print('Backtracking --> ',knapsack_bt(W,D,v,w))
print('Greedy --> ',knapsack_g(W,D,v,w))
print('Random -->',knapsack_r(W,D,v,w))
```