Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A lab Report 4

On

“Algorithm and Complexity”

[Course Code: COMP 314]

Submitted By:

Sabil Shrestha (51)

Submitted To:

Rajani Chulyadyo Mam

Submission Date: 22th November, 2020

Knapsack Problem

The knapsack 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. Knapsack Problem algorithm is a very helpful problem in combinatorics. It has several practical applications such as filling a lorry truck, fitting a cargo container, shopping bag in a supermarket and many logistical applications. The goal of the knapsack problem is to maximize the value we get from the corresponding weight given in the capacity of knapsack.

Knapsack algorithm can be divided into two types:

- **0/1 Knapsack**
  The 0/1 Knapsack problem using dynamic programming. In this Knapsack algorithm type, each package can be taken or not taken. Besides, the thief cannot take a fractional amount of a taken package or take a package more than once. This type can be solved by Dynamic Programming Approach.
- **Fractional Knapsack**
  Fractional Knapsack problem algorithm. Here we even take the fraction of the least valuable weight to maximize the profit or value we get. This method usually produces higher value than 0/1 method because whole capacity of knapsack or bag is utilized in this method. Fractional knapsack problem is mainly solved by using Greedy Strategy, but can be solved using brute force algorithm also.

Pseudocodes

1. **0/1 knapsack using bruteforce**

```
Knapsack_bruteforce_zero(wt, pro, cap)
1. n = len(p)
2. bit = get_combinations(n)
3. max_pro = 0
4. for b in bit:
5.    total_pro = 0
6.    for i in range(n):
7.        total_pro += int(b[i])*pro[i]
8.    total_wei = 0
9.    for i in range(n):
10.       total_wei += int(b[i])*wt[i]
11.   if total_wei <= cap and total_pro > max_pro:
12.       max_pro = total_pro
13.   return max_pro

def get_combinations(n)
1. return [bin(x)[2:].rjust(n,'0') for x in range(2**n)]
```

2. **Fractional Knapsack using brute force**

```
Knapsack_bruteforce_zero(wt, pro, cap)
1. n = len(p)
2. bit = get_combinations(n)
3. max_pro = 0
4. for b in bit:
5.    total_pro = 0
6.    for i in range(n):
7.        total_pro += int(b[i])*pro[i]
8.    total_wei = 0
9.    for j in range(n):
10.       total_wts += weight[j] * int(bit[j])
11.       total_prof += profit[j] * int(bit[j])
12.
13.       if total_wts >= capacity:
14.           previous_wt = total_wts - weight[j] * int(bit[j])
15.           total_wts = previous_wt
16.           previous_pro = total_prof - profit[j] * int(bit[j])
17.           total_prof = previous_pro
18.           remaining_wt = capacity - total_wts
19.           new_prof = previous_pro + remaining_wt * (profit[i] / weight[i])
```

```
            total_prof = new_prof
            break
    if total_pro > max_pro:
        max_pro = total_pro
    return max_pro
```

get_combinations(n)

```
1.  return [bin(x)[2:].rjust(n,'0') for x in range(2**n)]
```

3. Fractional Knapsack using greedy

Fractional_Knapsack_greedy(Array Weight, Array Value, int M)

```
1. for i <- 1 to size (Value)
2.  calculate cost[i] <- Value[i] / Weight[i]
3. Sort-Descending (cost)
4. i ← 1
5. while (i <= size(Value))
6.  if  Weight[i] <= M
7.      M ← M – W[i]
8.      total ← total + V[i];
9.  if  Weight[i] > M
10.         i ← i+1
```

4. 0 / 1 Knapsack problem using dynamic programming

0/1 knapsack_dynamic(n, Weight)

```
1. for w = 0, Weight
2. do V [0,w] ← 0
3. for i=0, n
4. do V [i, 0] ← 0
5. for w = 0, Weight
6. do if (wi≤ w & vi + V [i-1, w -  wi]> V [i -1,Weight])
7. then V [i, Weight] ← vi + V [i - 1, w - wi]
8. else V [i, Weight] ← V [i - 1, w]
```

Source Code

(brute_force.py)

```
#this gives range of strings 0000-1111

def get_all_combinations(n):
    return [bin(x)[2:].rjust(n,'0') for x in range(2**n)]

def knapsack_bruteforce(p, wt, capacity):
    assert len(p)  == len(wt),"p and wt must be same"

    n = len(p)
    bit_combinations = get_all_combinations(n)#0000 to 1111 all
    max_profit = 0
        for b in bit_combinations:  #0000 to 1111
        #calculate all the profits
        total_profit = 0
        for i in range(n):
            total_profit += int(b[i]) * p[i] #eg for 1010 (1*6+0*2+1*7+0*3)

        total_weight = 0
        for i in range(n):
            total_weight += int(b[i]) * wt[i]
        if total_weight <= capacity and total_profit > max_profit:
            max_profit = total_profit
           # solution = b

    #return (solution, max_profit)
    return max_profit
```

(brute_force_fractional.py)

```python
#this gives range of strings 0000-1111
def get_all_combinations(n):
    return [bin(x)[2:].rjust(n,'0') for x in range(2**n)]

def fractional_knapsack_bruteforce(p, w, capacity):
    assert len(p)  == len(w),"p and w must be same"

    n = len(p)
    bit_combinations = get_all_combinations(n)#0000 to 1111 all

    max_profit = 0
    #solution = ''
    for b in bit_combinations:  #0000 to 1111

        total_weight = 0
        for i in range(n):
            total_weight += int(b[i]) * w[i]


        if total_weight <= capacity:
            #calculate all the profits
            total_profit = 0
            for i in range(n):
                total_profit += int(b[i]) * p[i] #eg for 1010 (1*6+0*2+1*7+0*3)
        else:

            total_profit = 0
            anchor = 0
            anchor_weight = 0
            anchor_profit = 0
            for i in range(n):
```

```python
            for i in range(n):
                anchor += 1
                anchor_weight += int(b[i]) * w[i]
                anchor_profit += int(b[i]) * p[i]
                if(anchor_weight >= capacity):
                    #print(b)
                    previous_weight =anchor_weight - int(b[i]) * w[i]#previous anchor_weight
                    capacity_left = capacity- previous_weight
                    previous_profit = anchor_profit - int(b[i]) * p[i]
                    new_profit = previous_profit + capacity_left * (p[i] // w[i])
                    total_profit = new_profit

                    break

            # print(anchor)
            #print(total_profit)
            #print("----------------------")


        if total_profit > max_profit:
            max_profit = total_profit
            #solution = b

    #return (solution, max_profit)
    return max_profit

if __name__ == '__main__':
    pro = [60, 40, 100, 120]
    wt = [10, 40, 20, 30]
    capacity = 50
```

```python
    #combination, max_pro = fractional_knapsack_bruteforce(pro, wt, capacity)
    max_pro = fractional_knapsack_bruteforce(pro, wt, capacity)
    # print(combination)
    print(max_pro)
```

(greedy.py)

```
#this is to make ratio of weight and value for fractional
class Ratio:

    def __init__(self, weight, value, cap):
        self.weight = weight
        self.value = value
        self.cap = cap
        #ratio
        self.cost = value // weight

    def __lt__(self, other):
        return self.cost < other.cost

class fractionalknapsack:
    @staticmethod
    def maximumValue(weight, value, capacity):

        ratios = []
        for i in range(len(weight)):
            ratios.append(Ratio(weight[i], value[i], i))

        # sorting ratios in descending order
        ratios.sort(reverse=True)

        totalValue = 0
        for i in ratios:
            current_weight = int(i.weight)
            current_value = int(i.value)
            #if capacity left then add the total value of that one
            if capacity - current_weight >= 0:
                capacity -= current_weight
                totalValue += current_value
            else:
                #if the weight is full and only fractional is allowed then, now fract
ure the weight
                fraction = capacity / current_weight
```

```
                totalValue += current_value * fraction
                capacity = int(capacity - (current_weight * fraction))
                break
        return totalValue


# testing
if __name__ == "__main__":
    weight = [10, 40, 20, 30]
    value = [60, 40, 100, 120]
    capacity = 50

    print(fractionalknapsack.maximumValue(weight, value, capacity) )
```

(dynamic.py)

```
# Dynamic programming for 0/1 knapsack problem
def knapsackDynamic(capacity, weight, value, n):
    #making 2 by 2 matrix of all 0 (capcity * n)
   table = [[0 for x in range(capacity + 1)] for x in range(n + 1)]
   #Table in bottom up manner
   for i in range(n + 1):
      for c in range(capacity + 1):
          #this is making the side value of the table as 0 (weight 0 and value 0)
         if i == 0 or c == 0:
            table[i][c] = 0
            #if the current weight is smaller than total capcaity
         elif weight[i-1] <= c:
             #max(profit+ total baki profit, or upper cell ko profit)
            table[i][c] = max(value[i-1] + table[i-1][c-weight[i-1]], table[i-1][c])
         else:
            table[i][c] = table[i-1][c]
   return table[n][capacity]
#Main
value = [2,4,3,5,5]
weight = [3,4,1,2,6]
capacity = 12
```

```
n = len(value)
print(knapsackDynamic(capacity, weight, value, n))
```

Test case(knapsack_test.py)

```
import unittest
from greedy import fractionalknapsack
from dynamic import knapsackDynamic
from brute_force import knapsack_bruteforce
from brute_force_fractional import fractional_knapsack_bruteforce

class KnapsacktestCase(unittest.TestCase):
    def test_greedy_fractional(self):
        weight = [10, 40, 20, 30]
        values = [60,40,100,120]
        cap =50
        output =240

        self.assertEqual(fractionalknapsack.maximumValue(weight, values, cap), output
)

    def test_dynamic(self):
        value = [2,4,3,5,5]
        weights = [3,4,1,2,6]
        caps = 12
        n = len(value)
        outs = 15

        self.assertEqual(knapsackDynamic(caps, weights, value, n),outs)

    def test_brute_zero_one(self):
        prof = [5, 6, 7, 2]
        wt = [4, 2, 3, 1]
        caps = 8
        outs = 15
        self.assertEqual(knapsack_bruteforce(prof, wt, caps),outs)


    def test_fractional_brute(self):
        pros = [60, 40, 100, 120]
        wt = [10, 40, 20, 30]
```

```
        capss = 50
        output = 240

        self.assertEqual(fractional_knapsack_bruteforce(pros,wt,capss),output)



if __name__ == "__main__":
    unittest.main()
```

Output

```
In [2]: runfile('C:/Users/shres/Desktop/0 Study materials/0 This year/Algorithm and
Complexity/Lab/Lab 4/knapsack_test.py', wdir='C:/Users/shres/Desktop/0 Study materials/0
This year/Algorithm and Complexity/Lab/Lab 4')
....15

----------------------------------------------------------------------
Ran 4 tests in 0.002s

OK

In [3]:
```

Observation

The knapsack problem was solved using greedy approach, dynamic approach and brute force approach also. The knapsack problem was done using python programming language. First brute force algorithm was used to solve fractional knapsack and 0/1 knapsack. for 0/1 knapsack using brute force, all the possibilities that we can get from the input was calculated using bin. The higher profit from the possibilities was stored as maximum value in a variable. Later the maximum value obtained from the algorithm was returned. This method is slow as calculating all the possibilities is time consuming and has higher time complexity than other algorithm. Second, we did greedy method for solving fractional knapsack. Here firstly, all the profit to weight ratio was calculated. Then the highest ratios were taken multiplied by weight and profit until the capacity was full. The remaining capacity left was multiplied by fraction of the weight of the other ratio weight in order because this is a fractional knapsack problem. For, 0/1 knapsack problem using dynamic programming, table was made which is a matrix of size capacity + 1 and n + 1, then the outer area of matrix was assigned 0. The inner table were filled using the formula. The last value of the table is the solution to the 0/1 knapsack problem from input. The program was then tested with program knapsack_test.py. All the program passed the test successfully. The test was done using the imported functions from the python files. All the tests were successfully completed using unittest in python.