Discrete Optimisation Assignment 1

August 18, 2020

1 Linear and Quadratic Knapsack Problems

1.1 Neil Fabiao, Student number: 2216748

1.1.1 Problem 1 Linear Knapsack Problem

Consider the following pairs :

$(v_i, w_i) = \{(2,7),(6,3),(8,3),(7,5),(3,4),(4,7),(6,5)$, (5,4),(10,15),(9,10),(8,17),(11,3),(12,6),(15,11),(6,6) (8,14),(13,4),(14,8),(15,9),(16,10),(13,14),(14,17),(15,9),(26,24) (13,11),(9,17),(25,12),(26,14) }

Total capacity $W = 30$

Greedy Algorithm:

[1]:
```
import numpy as np

def geedy_knapsack(vali,weig,capacity,Knapsack_bag):

    #2. Use of a score or efficiency function, i.e. the profit to weight ratio
    def ratio(val,wei):
        score = []
        for i in range(len(val)):
            #appending the score of each iteration of the value and weight arrays
            score = np.append(score,(val[i]/wei[i]))
        return score

    #3. Sort the items non-increasingly according to the efficiency function and␣
↪retrieving the indicies
    #Note: here we are sorting the items using quick sort
    scores_ratio = ratio(vali,weig)
    indicies = np.argsort(-scores_ratio, kind = 'quicksort')

    #4. Add into knapsack the items with the highest score, taking note of their␣
↪accumulative weights
    #until no item can be added.
    Knapsack_index = np.zeros(len(vali)) # this variable will store the indicies␣
↪of the items in the knapsack
```

```
    Knapsack_bag # this is the bag or accumulator variable that will store the␣
 ↪elements in the bag

    # Add item with highest score, that is below capacity W.
    for i in range(len(vali)):
        current_item = indicies[i] #this is to keep track of the current␣
 ↪indicies when adding to the bag
        if((Knapsack_bag+weig[current_item]) <= capacity ): # only add the items␣
 ↪to the bag until not greater of equal to the capacity
            Knapsack_index[current_item] = 1  # assign 1 to item in the current␣
 ↪index that satisfies our condition
            Knapsack_bag += weig[current_item] # add the weight of the items

    Total_weight = np.sum(weig[Knapsack_index==True])
    Invididual_weight =  weig[Knapsack_index==True]
    Total_profit = np.sum(vali[Knapsack_index==True])
    Individual_profit =  vali[Knapsack_index==True]
    return(Total_weight,Invididual_weight,Total_profit,Individual_profit)
```

[2]:
```
#1. Identify the available items with their weights and values and take note of␣
 ↪the maximum capacity of the bag.
value = np.
 ↪array([2,6,8,7,3,4,6,5,10,9,8,11,12,15,6,8,13,14,15,16,13,14,15,26,13,9,25,26])
weight = np.
 ↪array([7,3,3,5,4,7,5,4,15,10,17,3,6,11,6,14,4,8,9,10,14,17,9,24,11,17,12,14])
W_capacity = 30


a,b,MAX_greedy,d = geedy_knapsack(value,weight,W_capacity,Knapsack_bag = 0)

print("Total weight of the items in the knapsack is, ",a,' and the individual␣
 ↪weights are:',b)
print("Total profit of the items in the knapsack is, ",MAX_greedy,"and the␣
 ↪individual profits:",d)
```

```
Total weight of the items in the knapsack is,  30  and the individual weights
are: [ 3  3  5  3  4 12]
Total profit of the items in the knapsack is,  70 and the individual profits: [
6  8  7 11 13 25]
```

Polynomial Time Approximation Algorithm (PTAS) for the linear knapsack problem:

1. Consider all sets of up to at most k items (k <= 5)

$S = \{F \subset \{1,2,\cdots,28\} : |F| = k, w(F) < W\}$

2. For all F in S :

Pack F into the knapsack

Greedily fill the remaining capacity

End

3. Return highest valued item combination set

Note $|F|$ denotes the number of item in the set F; $w(F)$ denotes the total weights of the items in F; Use $k = 10$; when creating S do not take subsets F with less than 3 items.

In this algorithm we find all the possible combination from the random items that we select, given that the random items are selected it is necessary to ensure that all the avaliable possible combinations selected from this process are not less than 3 items.

[3]:
```
import itertools
from itertools import combinations

#I implemented this algorithm with ratio, given that the sorting procedure␣
 ↪allows the program to find the best elements using
#their indexes. Thus the best combinations will be selected first and iterate␣
 ↪starting from the best to worst.
#Note: The best set may yield items with weight larger than capacity and this is␣
 ↪not considered by ratio thus the if statement
#allows for better selection of items in the list.

def ratio(val,wei):
        score = []
        for i in range(len(val)):
            #appending the score of each iteration of the value and weight arrays
            score = np.append(score,(val[i]/wei[i]))
        return score

Combination_weights = []
Combination_values = []

first_pass = True
iteration = 0


while first_pass or (np.sum(Combination_values[iteration-1])<MAX_greedy+1) or␣
 ↪(np.sum(Combination_values[iteration-1]) < maximo) :
    first_pass = False
    k = 5
    #1. Consider all sets of up to at most k items (k <= 5)
    Random_indicies = np.random.choice(len(value),k, replace=False)
    Combination_scores = []


    output = sum([list(map(list, combinations(Random_indicies, i))) for i in␣
 ↪range(len(Random_indicies) + 1)], [])
```

```
   output = output[29:] #select the sets with more than 3 items per combination␣
↪and this starts at index 29

   for i in range(len(output)):
       Random_Values = value[output[i]]
       Random_Weights = weight[output[i]]
       scores_ratio = ratio(Random_Values,Random_Weights)
       Combination_scores.append(np.sum(scores_ratio))

   Combination_array = np.asarray(Combination_scores)
   indicies_sorted = np.argsort(-Combination_array, kind = 'quicksort')

   Knapsack_index = np.zeros(len(value)) # this variable will store the␣
↪indicies of the items in the knapsack
   Knapsack_bag = 0 # this is the bag or accumulator variable that will store␣
↪the elements in the bag

   #2. For all F in S
   for i in range(len(indicies_sorted)):
       Index_of_Max = output[indicies_sorted[i]]
       Current_sum = np.sum(weight[Index_of_Max])

       if( (Knapsack_bag+Current_sum) <= 30 ):
           Knapsack_index[Index_of_Max] = 1  # assign 1 to item in the current␣
↪index that satisfies our condition
           Knapsack_bag += np.sum(Current_sum)


   new_Weight = weight[Knapsack_index==False]
   new_Value = value[Knapsack_index==False]
   Sum_weights_left = np.sum(weight[Knapsack_index==True])
   Sum_value_left = np.sum(value[Knapsack_index==True])
   new_Capacity =  W_capacity-Knapsack_bag

   #Greedily fill the remaining capacity

   #selecting 3 beacuse the min element in the bag that can be added has the␣
↪min weight of 4
   if (new_Capacity <= 30 and new_Capacity>2):
       #'Start Greedy algorithm'
       a,b,c,d =␣
↪geedy_knapsack(new_Value,new_Weight,W_capacity,Sum_weights_left)
       Sum_weight = a+Sum_weights_left
       Weights_final = np.append(b,weight[Knapsack_index==True])
       Sum_value = c+Sum_value_left
       Values_final = np.append(d,value[Knapsack_index==True])
```

```
        Combination_weights.append(Weights_final)
        Combination_values.append(Values_final)
    else:
        #'No need for greedy algorithm given that the weight left is less than␣
↪the avaliable weight'
        Sum_weight = Sum_weights_left
        Weights_final = weight[Knapsack_index==True]
        Sum_value = Sum_value_left
        Values_final = value[Knapsack_index==True]
        Combination_weights.append(Weights_final)
        Combination_values.append(Values_final)


    iteration+=1
    res = [sum(e) for e in Combination_values] #sums all the results from the␣
↪Combination_values
    #3. Return highest valued item combination set
    maximo = np.array(res).max()
```

[4]:
```
print('These are the best value combination',Combination_values[iteration-1])
print('These are the best weigtht combination ',Combination_weights[iteration-1])
print('Tha max value is :',np.sum(Combination_values[iteration-1]))
```

```
These are the best value combination [13  8 11 14 25]
These are the best weigtht combination  [ 4  3  3  8 12]
Tha max value is : 71
```

1.1.2 Problem 2: Quadratic Knapsack Problem

In the algorithm there are 6 steps.

In step 1: You select at random 7 items

Step 2 create all possible pair,

Step 3: Sorts each pair based on the formula given

Step 4: Add the best pair in the knapsack and go to Step 1 and select another 7 item form the remaining items and repeat the process from steps 1 to 4. If on the other hand you cannot add the best pair in the knapsack then use the greedy method (the one used in linear knapsack problem) to put item from the remaining items.

Problem 2 stats with the following statement:

- Consider a QKP problem with :

[5]:
```
n = 15 #this variable may not be used during the code since len of vi and wi is␣
↪equal to 15
```

```
vi = np.array([7,6,13,16,5,10,9,23,18,12,9,22,17,32,8])

wi = np.array([13,14,14,15,15,9,26,24,13,11,9,12,25,12,26])

W_Maxcapacity = 50


pij = np.array([  [12,7,6,13,8,11,7,15,23,14,15,17,9,15],

[15,13,10,15,9,10,8,17,11,13,12,16,15],

[11,16,6,8,14,13,4,14,8,15,9,16],

[10,13,14,14,17,15,14,6,24,13,4],

[9,7,25,12,6,6,16,10,15,14],

[2,13,12,16,9,11,23,10,21],

[8,18,4,13,14,14,17,15],

[9,16,12,3,14,14,27],

[15,16,13,14,7,17],

[28,5,19,6,18],

[13,4,13,16],

[11,19,13],

[15,12],

[16]  ])

#Converted the above pij matrix to an inverted matrix padded with 0 ,this is to␣
 ↪ensure that
#the program can traverse to the list using for loops with ease and assign 0 to␣
 ↪positions non existing in the matrix.
#Note: this was done manually altough it can be done using some numpy or matrix␣
 ↪manipulation principles.
#The final shape has to be 15 by 15 such that the it is possible to achieve same␣
 ↪size as N
pij_invertedMatrix = np.array([ [12,  7,  6, 13,  8, 11, 7,  15, 23, 14, 15, 17,␣
 ↪ 9, 15, 0],
                                [15, 13, 10, 15,  9, 10,  8, 17, 11, 13, 12, 16,␣
 ↪15,  0, 0],
```

```
                                [11, 16,  6,  8, 14, 13,  4, 14,  8, 15,  9, 16,␣
↪ 0,  0, 0],
                                [10, 13, 14, 14, 17, 15, 14,  6, 24, 13,  4,  0,␣
↪ 0,  0, 0],
                                [ 9,  7, 25, 12,  6,  6, 16, 10, 15, 14,  0,  0,␣
↪ 0,  0, 0],
                                [ 2, 13, 12, 16,  9, 11, 23, 10, 21,  0,  0,  0,␣
↪ 0,  0, 0],
                                [ 8, 18,  4, 13, 14, 14, 17, 15,  0,  0,  0,  0,␣
↪ 0,  0, 0],
                                [ 9, 16, 12,  3, 14, 14, 27,  0,  0,  0,  0,  0,␣
↪ 0,  0, 0],
                                [15, 16, 13, 14,  7, 17,  0,  0,  0,  0,  0,  0,␣
↪ 0,  0, 0],
                                [28,  5, 19,  6, 18,  0,  0,  0,  0,  0,  0,  0,␣
↪ 0,  0, 0],
                                [13,  4, 13, 16, 0,   0,  0,  0,  0,  0,  0,  0,␣
↪ 0,  0, 0],
                                [11, 19, 13,  0, 0,   0,  0,  0,  0,  0,  0,  0,␣
↪ 0,  0, 0],
                                [15, 12,  0,  0, 0,   0,  0,  0,  0,  0,  0,  0,␣
↪ 0,  0, 0],
                                [16,  0,  0,  0, 0,   0,  0,  0,  0,  0,  0,  0,␣
↪ 0,  0, 0] ])
```

[6]:
```
#this function calculates efficiency score as given in the first expression for␣
 ↪the assignment.
def efficiency_score_first(Pair,Matrix,Weig,Valu):
    Random_i = 0
    Random_j = 0
    efficiency_score = []


    for i in range(len(Pair)-1):
        Random_i = Pair[i][1]
        Random_j = Pair[i+1][1]

        W_i = Weig[Random_i]
        W_j = Weig[Random_j]

        Pairs_ij = Matrix[Random_i][Random_j]
        score = (Pairs_ij)/(W_i+W_j)
        efficiency_score = np.append(efficiency_score,score)

    efficiency_score = efficiency_score[efficiency_score !=0] #remove elements␣
 ↪that are equal to 0
```

```
    efficiency_sorted = np.argsort(-efficiency_score, kind = 'quicksort')

    return efficiency_sorted
```

[7]:
```
#this function calculates efficiency score as given in the second expression for␣
 ↪the assignment.
def efficiency_score_second(Pair,Matrix,Weig,Valu):
    Random_i = 0
    Random_j = 0
    efficiency_score = []


    for i in range(len(Pair)-1):
        Random_i = Pair[i][1]
        Random_j = Pair[i+1][1]

        W_i = Weig[Random_i]
        W_j = Weig[Random_j]

        V_i = Valu[Random_i]
        V_j = Valu[Random_j]

        Pairs_ij = Matrix[Random_i][Random_j]
        score = (Pairs_ij+V_i +V_j)/(W_i+W_j)
        efficiency_score = np.append(efficiency_score,score)

    efficiency_score = efficiency_score[efficiency_score !=0] #remove elements␣
 ↪that are equal to 0
    efficiency_sorted = np.argsort(-efficiency_score, kind = 'quicksort')

    return efficiency_sorted
```

[10]:
```
from itertools import compress,product

Knapsack_bag = 0 # this is the bag or accumulator variable that will store the␣
 ↪elements in the bag
pairs_Implementation1=[]
Accumulator_weight = 0
Accumulator_value = 0

print('This is implementation using the first efficiency score :\n')
k = 7
Knapsack_index = np.zeros(len(vi)) # this variable will store the indicies of␣
 ↪the items in the knapsack
                                   #as stated above the len will be 15. Note: if␣
 ↪requirements change just use n where len(vi)
```

```
Random_indicies = np.random.choice(len(wi)-1,k, replace=False)

Pairs_random_indicies = list(product(Random_indicies,repeat=2))

#remove indicies with the same value for example (0,0),(7,7),(10,10)
del Pairs_random_indicies[0]
del Pairs_random_indicies[7]
del Pairs_random_indicies[14]
del Pairs_random_indicies[21]
del Pairs_random_indicies[28]
del Pairs_random_indicies[35]
del Pairs_random_indicies[42]

efficiency_sorted =␣
 ↪efficiency_score_first(Pairs_random_indicies,pij_invertedMatrix,wi,vi)


#step 4 Add into knapsack the pair of items with the highest score, ensuring␣
 ↪that the accumulated
#weight does not exceed the maximum capacity.
for i in range(len(efficiency_sorted)):
    Random_i = Pairs_random_indicies[efficiency_sorted[i]][0]
    Random_j = Pairs_random_indicies[efficiency_sorted[i]][1]

    Weight_i = wi[Random_i]
    Weight_j = wi[Random_j]

    Value_i = vi[Random_i]
    Value_j = vi[Random_j]

    #add the pairs to the list as arrays
    Current_sum = Weight_i + Weight_j
    Current_sumvalue = Value_i + Value_j

    if( (Knapsack_bag+Current_sum) <= W_Maxcapacity ):
        # assign 1 to item in the current index that satisfies our condition
        Knapsack_index[Random_i] = 1
        Knapsack_index[Random_j] = 1
        Knapsack_bag += np.sum(Current_sum)

        pair = np.array([ [Value_i,Weight_i],[Value_j,Weight_j] ] )

        Accumulator_weight +=np.sum(Current_sum)
        Accumulator_value +=np.sum(Current_sumvalue)
```

```
        print('Now the bag has',Knapsack_bag,'kgs with pairs of values and␣
 ↪weights added being:\n',pair)


#note: have to fix this such that you have the values for repeated items
new_Weight_QKP = wi[Knapsack_index==False]
new_Value_QKP = vi[Knapsack_index==False]
Sum_weights_left_QKP = np.sum(wi[Knapsack_index==True])
Sum_value_left_QKP = np.sum(vi[Knapsack_index==True])
new_Capacity_QKP =  W_Maxcapacity-Knapsack_bag

#print('This is the weight left',new_Weight_QKP)
#print('This is the value left',new_Value_QKP)
print('We are left with',new_Capacity_QKP,'kg')
pairs_Implementation1 = np.append(pair,pairs_Implementation1)


#selecting 2 beacuse the min element in the bag that can be added has the min␣
 ↪weight of 4
if ( (new_Capacity_QKP <= W_Maxcapacity) and (new_Capacity_QKP != 0 and␣
 ↪new_Capacity_QKP > 5) ):
    print('\nStart Greedy algorithm')
    #'Start Greedy algorithm'
    print('We start with the new max capacity:',new_Capacity_QKP)
    a,b,c,d =␣
 ↪geedy_knapsack(new_Value_QKP,new_Weight_QKP,W_Maxcapacity,Sum_weights_left_QKP)
    Sum_weight_QKP = a+Sum_weights_left_QKP
    Weights_final_QKP = np.append(b,wi[Knapsack_index==True])
    Sum_value_QKP = c+Sum_value_left_QKP
    Values_final_QKP = np.append(d,vi[Knapsack_index==True])


    print('The capacity left in the end is:',W_Maxcapacity-Sum_weight_QKP)
    print('The added greedy item has weight :',b, 'and value :',d)
    Accumulator_weight = Accumulator_weight+b
    Accumulator_value = Accumulator_value+d

else:
    print('No need for greedy algorithm given that the weight left is less than␣
 ↪the avaliable weight')

print('\nThe sum of values for implementation 1 is :',int(Accumulator_value))
print('The sum of weights for implementation 1 is :',int(Accumulator_weight))
```

```
This is implementation using the first efficiency score :

Now the bag has 28 kgs with pairs of values and weights added being:
```

```
 [[ 7 13]
 [16 15]]
Now the bag has 50 kgs with pairs of values and weights added being:
 [[ 7 13]
 [ 9  9]]
We are left with 0 kg
No need for greedy algorithm given that the weight left is less than the
avaliable weight

The sum of values for implementation 1 is : 39
The sum of weights for implementation 1 is : 50
```

[9]:
```
from itertools import compress,product
Knapsack_bag = 0 # this is the bag or accumulator variable that will store the␣
 ↪elements in the bag
pairs_Implementation2=[]
Accumulator_weight = 0
Accumulator_value = 0
print('This is implementation using the second efficiency score :\n')

k = 7
Knapsack_index = np.zeros(len(vi)) # this variable will store the indicies of␣
 ↪the items in the knapsack
                                   #as stated above the len will be 15. Note: if␣
 ↪requirements change just use n where len(vi)


Random_indicies = np.random.choice(len(wi)-1,k, replace=False)

Pairs_random_indicies = list(product(Random_indicies,repeat=2))

#remove indicies with the same value for example (0,0),(7,7),(10,10)
del Pairs_random_indicies[0]
del Pairs_random_indicies[7]
del Pairs_random_indicies[14]
del Pairs_random_indicies[21]
del Pairs_random_indicies[28]
del Pairs_random_indicies[35]
del Pairs_random_indicies[42]


efficiency_sorted =␣
 ↪efficiency_score_second(Pairs_random_indicies,pij_invertedMatrix,wi,vi)

#step 4 Add into knapsack the pair of items with the highest score, ensuring␣
 ↪that the accumulated
#weight does not exceed the maximum capacity.
```

```
for i in range(len(efficiency_sorted)):
    Random_i = Pairs_random_indicies[efficiency_sorted[i]][0]
    Random_j = Pairs_random_indicies[efficiency_sorted[i]][1]

    Weight_i = wi[Random_i]
    Weight_j = wi[Random_j]

    Value_i = vi[Random_i]
    Value_j = vi[Random_j]

    #add the pairs to the list as arrays
    Current_sum = Weight_i + Weight_j
    Current_sumvalue = Value_i + Value_j
    if( (Knapsack_bag+Current_sum) <= W_Maxcapacity ):
        # assign 1 to item in the current index that satisfies our condition
        Knapsack_index[Random_i] = 1
        Knapsack_index[Random_j] = 1
        Knapsack_bag += np.sum(Current_sum)

        Accumulator_weight +=np.sum(Current_sum)
        Accumulator_value +=np.sum(Current_sumvalue)

        pair = np.array([ [Value_i,Weight_i],[Value_j,Weight_j] ] )
        pairs_Implementation2 = np.append(pair,pairs_Implementation2)
        print('Now the bag has',Knapsack_bag,'kgs with pairs of values and␣
 ↪weights added being:\n',pair)


#note: have to fix this such that you have the values for repeated items
new_Weight_QKP = wi[Knapsack_index==False]
new_Value_QKP = vi[Knapsack_index==False]
Sum_weights_left_QKP = np.sum(wi[Knapsack_index==True])
Sum_value_left_QKP = np.sum(vi[Knapsack_index==True])
new_Capacity_QKP =  W_Maxcapacity-Knapsack_bag

print('We are left with',new_Capacity_QKP,'kg')



#selecting 2 beacuse the min element in the bag that can be added has the min␣
 ↪weight of 4
if ( (new_Capacity_QKP <= W_Maxcapacity) and (new_Capacity_QKP != 0 and␣
 ↪new_Capacity_QKP > 5) ):
    print('\nStart Greedy algorithm')
    #'Start Greedy algorithm'
    print('We start with the new max capacity:',new_Capacity_QKP)
```

```
    a,b,c,d = geedy_knapsack(new_Value_QKP,new_Weight_QKP,W_Maxcapacity,Sum_weights_left_QKP)
    Sum_weight_QKP = a+Sum_weights_left_QKP
    Weights_final_QKP = np.append(b,wi[Knapsack_index==True])
    Sum_value_QKP = c+Sum_value_left_QKP
    Values_final_QKP = np.append(d,vi[Knapsack_index==True])

    print('The capacity at the end is:',W_Maxcapacity-Sum_weight_QKP)
    print('The added greedy item has weight :',b, 'and value :',d)
    Accumulator_weight = Accumulator_weight+b
    Accumulator_value = Accumulator_value+d
else:
    print('No need for greedy algorithm given that the weight left is less than the avaliable weight')


print('\nThe sum of values for implementation 2 is :',int(Accumulator_value))
print('The sum of weights for implementation 2 is :',int(Accumulator_weight))
```

```
This is implementation using the second efficiency score :

Now the bag has 26 kgs with pairs of values and weights added being:
 [[13 14]
 [32 12]]
Now the bag has 47 kgs with pairs of values and weights added being:
 [[10  9]
 [32 12]]
We are left with 3 kg
No need for greedy algorithm given that the weight left is less than the
avaliable weight

The sum of values for implementation 2 is : 87
The sum of weights for implementation 2 is : 47
```

[ ]: