0-1 Knapsack Problem

November 10, 2019

GAA Project 2019 . Course teacher: ***Dr. Vibhor Kant***

Team Member	Roll Number
Ansh Mittal	17UCS028
Anshul Jain	17UCS029
Anshul Kiyawat	17UCS030
Anshu Musaddi	17UCS185

1 Importing libraries

[1]:
```
import random
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook
```

2 Defining the Problem

- The problem consists of n items in our case it is 30.
- Every value has weights and values and we have given a bag with capacity of ten times the minimum weight of items.
- We want to maximize the profit.

2.1 Initializing Input parameters

[2]:
```
n = 30

values = [11,12,13,14,15,16,17,18,19,10,25,38,55,15,62,
          99,35,12,86,15,69,56,12,98,35,69,86,87,51,29]
weight = [63,52,46,25,45,95,69,53,64,36,78,89,52,69,14,
          87,36,58,79,65,27,81,35,86,93,87,41,65,63,57]
capacity = int(min(weight))*10
```

2.2 Initializing probablity values & other variables

[3]:
```
cross_over_prob = 0.7
mutation_prob = (1/n)

populationSize = 500
noOfGenerations = 50

# As it is generational model offsprings formed is same as population size
noOfOffsprings = populationSize
```

3 Defining the functions required for Genetic Algorithm

3.1 Validating the chromosome

Make all the bits 0 after the point where capacity is full so that it can fit in the given bag

[4]:
```
def validate(chrm,weight,capacity):
    cnt = 0
    for i in range(len(chrm)):
        if cnt <= capacity:
            cnt+=chrm[i]*weight[i]
        if cnt > capacity:
            chrm[i]=0
    return chrm
```

3.2 Fitness function

We define fitness function as the sum of values of the items that are considered in the respective chromosome.

[5]:
```
def fitness(chrm,values):
    cnt = 0
    for i in range(len(chrm)):
        cnt+=chrm[i]*values[i]
    return cnt
```

3.3 Random initialization of a chromosome

The following function initializes a unique random chromosomes.The size of chromosome i.e number of items n passes as a parameter. The following function will return list as a chromosome.

Binary representation is used for this problem

[6]:
```
def randinit(n):
    chrm = [0]*n
    noOfIteration = random.randint(0,n)
    for i in range(noOfIteration+1):
```

```
        chrm[random.randint(0,n)-1] = 1
    return chrm
```

3.4 Bit Flipping

Flip the value.

- 0 -> 1
- 1 -> 0

[7]:
```
def flip(chrm,point):
    if chrm[point]==1:
        chrm[point] = 0
    elif chrm[point]==0:
        chrm[point] = 1
    return chrm
```

3.5 Mutation

Randomly flip the value of a allele

- Bit flipping is used for mutation
- the probability of a gene to mutate is 1/n
- After mutation offsprings is validated

[8]:
```
def mutation(chrm,n):
    point = random.randint(0,n-1)
    chrm = flip(chrm,point)
    return chrm
```

3.6 Crossover

- One point crossover is used for crossover
- Crossover point is choosen randomly

[9]:
```
def crossover(chrm1,chrm2,n):
    point = random.randint(0,n)
    chrm3 = chrm1[:point]
    chrm3.extend(chrm2[point:])
    chrm4 = chrm2[:point]
    chrm4.extend(chrm1[point:])
    return chrm3,chrm4
```

3.7 Parents Selection

We are taking two parents randomly and selecting one with better fitness value

[10]:
```
def parentSelection(chrm1,chrm2,values):
    if(fitness(chrm1,values)>=fitness(chrm2,values)):
```

```
        return chrm1
    return chrm2
```

3.8 Mating Pool Creation

Randomly creating mating pool of size 2 by parent selection

[11]:
```
def matingPoolCreation(populationSize,population,values):
    matingPool = []
    while len(matingPool)<2:
        rand1 = random.randint(0,populationSize-1)
        rand2 = random.randint(0,populationSize-1)
        parent = parentSelection(population[rand1],population[rand2],values)
        if parent not in matingPool:
            matingPool.append(parent)
    return matingPool
```

3.9 Best Chromosome Selection

We are selecting best chromosome in a generation on the basis of their fitness values

[12]:
```
def bestchromosome(population,values):
    p = pd.DataFrame(np.zeros((len(population), 2)), columns=['Chromosome',␣
 ↪'Fitness'], dtype=object)
    for i in range(len(population)):
        p.at[i, 'Chromosome'] = population[i]
        p.at[i, 'Fitness'] = fitness(population[i], values)
    p = p.sort_values(by=['Fitness'], ascending=False)
    return p.iloc[0,0], p.iloc[0, 1]
```

3.10 Plotting graph

We are Plotting line plot with

- X- Axis - Generation number
- Y - Axis - Fitness values

[13]:
```
def plotting(bestFitness):
    plt.plot(bestFitness)
    plt.savefig('Output.png')
```

4 Execution of the Genetic Algorithm

4.1 Initialising

[14]:
```
print("values = {}\nweight = {}\ncapacity = {}".format(values,weight,capacity))

# Opening file for writing output
f = open("bestforeachgeneration.txt","w+")

# Randomly initializing the population
population = []
while len(population) < populationSize:
    chromosome = randinit(n)
    chromosome = validate(chromosome,weight,capacity)
    if chromosome not in population:
        population.append(chromosome)

print("\n best in population of initial generation is {}".
 ↪format(bestchromosome(population,values)))
```

```
values = [11, 12, 13, 14, 15, 16, 17, 18, 19, 10, 25, 38, 55, 15, 62, 99, 35,
12, 86, 15, 69, 56, 12, 98, 35, 69, 86, 87, 51, 29]
weight = [63, 52, 46, 25, 45, 95, 69, 53, 64, 36, 78, 89, 52, 69, 14, 87, 36,
58, 79, 65, 27, 81, 35, 86, 93, 87, 41, 65, 63, 57]
capacity = 140

 best in population of initial generation is ([0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0], 230)
```

4.2 Execution

[15]:
```
%%time
bestFitness = []
for i in range(noOfGenerations):
    offspring = []
    while len(offspring)<noOfOffsprings:

        # Creating mating pool
        matingPool = matingPoolCreation(populationSize,population,values)

        # Applying crossover using crossover probability
        rand = random.random()
        if(rand<=cross_over_prob):
            matingPool[0],matingPool[1] =␣
 ↪crossover(matingPool[0],matingPool[1],n)

        # Applying mutation using mutation probability
        rand = random.random()
```

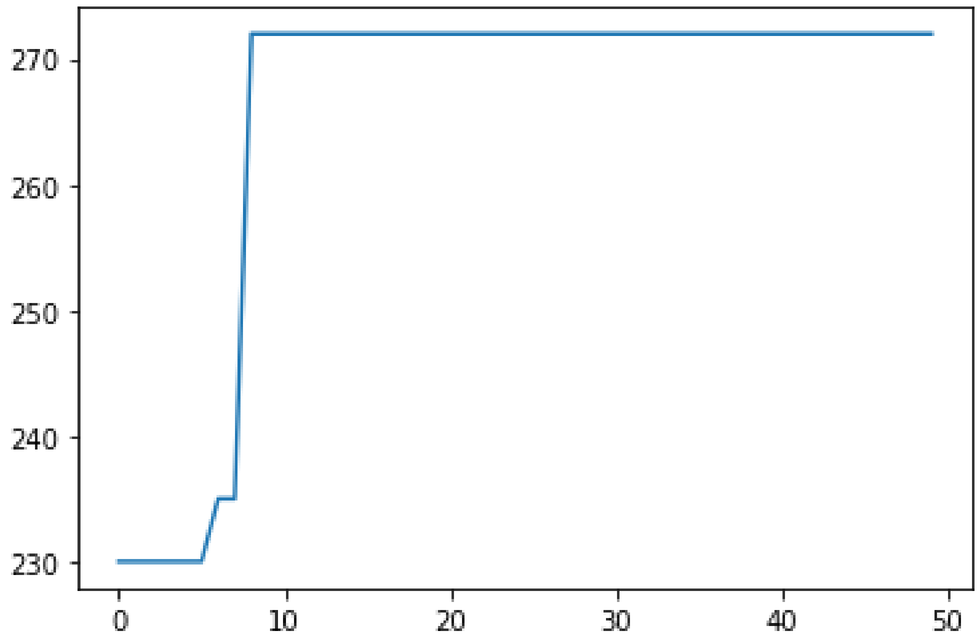
```
        if(rand<=mutation_prob):
            matingPool[0] = mutation(matingPool[0],n)
        rand = random.random()
        if(rand<=mutation_prob):
            matingPool[1] = mutation(matingPool[1],n)

        for j in matingPool:
            offspring.append(validate(j,weight,capacity))

    # Feeding new generation into population
    population.extend(offspring)
    del(offspring)

    # Finding best chromosome of generation
    finalchromosome,bestFit = bestchromosome(population,values)
    f.
↪write("\n-----------------------------------------------------------------" +
          ␣
↪"-----------------------------------------------------------------" +
            str(i+1) + "\tGeneration\n")
    f.write("chromosome {} \t fitness {}\n".format(finalchromosome,bestFit))
    bestFitness.append(bestFit)
```

```
CPU times: user 9.55 s, sys: 36.1 ms, total: 9.58 s
Wall time: 9.58 s
```

4.3 Generated Solution

[17]:
```
print("\n\nbest in population of last generation is {}\nchromosome {} ".
 ↪format(bestFitness[noOfGenerations-1],finalchromosome))
# Plotting and visualizing the best chromosome of every generation
plotting(bestFitness)
f.close()
```

```
best in population of last generation is 272
chromosome [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0,
0, 0, 0, 1, 0, 0, 0]
```


270
260
250
240
230
0
10
20
30
40
50