دانشگاه صنعتي اميركبير
(پلی تکنیک تهران)
دانشکده علوم کامپیوتر

گزارش سوم

نگارش
محمدرضا اردستانی

استاد راهنما
مهدی قطعی

فروردین 14 1399

فـهـرست مـطـالـب

صفـحه

فصل اول
جستجوی محلی

مقدمه

جستجوی محلی در بعضی از مسائلی که شما **meta knowledge** درباره ی محیط اطرافتان ندارید یک راهکار بی جایگزین میباشید.
در این گزارش نحوه عملکرد **genetic algorithm** ک یک روش سرچ محلی میباشد را در مسئله ای که ما روشی برای پیدا کردن جواب بهینه آن داریم میسنجیم.

فـصل دوم
حل 0/1 Knapsack with different algorithm

Dynamic programming and Brute-force algorithm -2-1

صورت مسئله 0/1 knapsack به صورت استاندارد تعریف شده و ما قبلا با روش های حل آن به وسیله naïve idea یا همون brute force آشنا شده ایم . (برای یادآوری آن ها هم به وبسایت هایی در رفرنس ارجاع داده ام که میتوان برای یادآوری آن ها به وبسایت های ذکر شده رجوع کرد).

Time complexity of Brute force method = BigO(2^n)

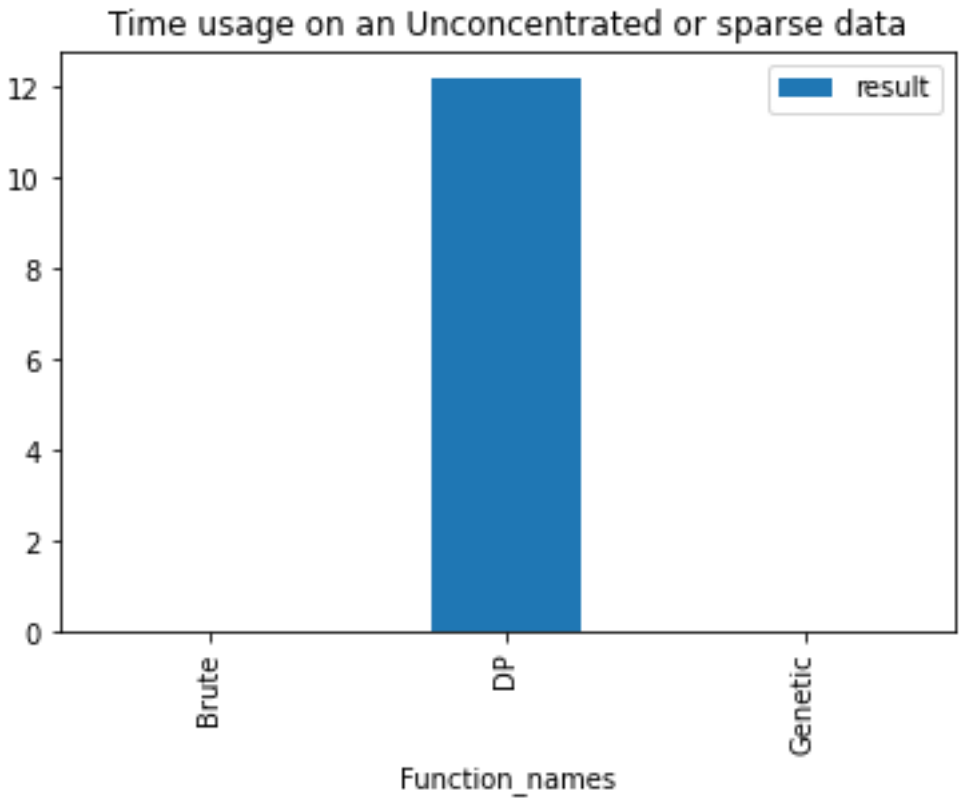
Time complexity of DP = Theta(n*w)

N= number of items that we can chose from it , W= Bag's capacity (kg)

اما نکته ای که در حل این مسئله به روشی به غیر از داینامیک پروگرمیکنگ وجود دارد این است که داینامیک پروگرمیکنگ در کیس هایی که داده ها unconcentrate or sparse هستند زمانی بیشتر از brute force or Genetic algorithm میگیرد و علت آن این است که در روش حل مسئله با داینامیک پروگرمینگ اولا فرض شده که وزن ها همگی integerهستند و نه floatو نیز در حل مسئله یک جدول که تعداد columnهای آن به اندازه ی W+1 میباشد تولید میکند. این کار وقتی مشکل ساز میشود که ما عملا داده های دور از هم داشته باشیم یا داده هایمان اعشاری باشند که برای تبدیل مسئله به داده های با وزن غیر اعشاری باید همه ی وزن ها و value ها را به اندازه ی بزرگترین رقم اعشاری که در داده ها وجود دارد در10 ضرب کینم (جزئیات بیشتر در رفرنس آمده است).

به طور مثال اگر آیتم های ما تنها دو مورد با value=[1,1] and Weight =[1,1000000] باشد و ماکسیمم ظرفیت کوله پشتی هم 900000 باشد باید یک جدول به سایز (1+9000000)*2ساخته شود و همه ی خانه های آن پر شود. درصورتی که با بروت فورس این مسئله نهایتا در 2^2 = 4 حالت ساده بررسی شده و تمام میشود. یا ژنتیک الگوریتم هم این مسئله را در کمترین زمان ممکن حل میکند.

ریزالت تست شده برای این تست کیس در جدول زیر آورده شده است:



توجه شود که بروت فورس همه ی 2^ n حالت را ممکن است بررسی نکند. بعضی شاخه ها به علت بالارفتن وزن آن ها از وزن مجاز حذف میشوند.

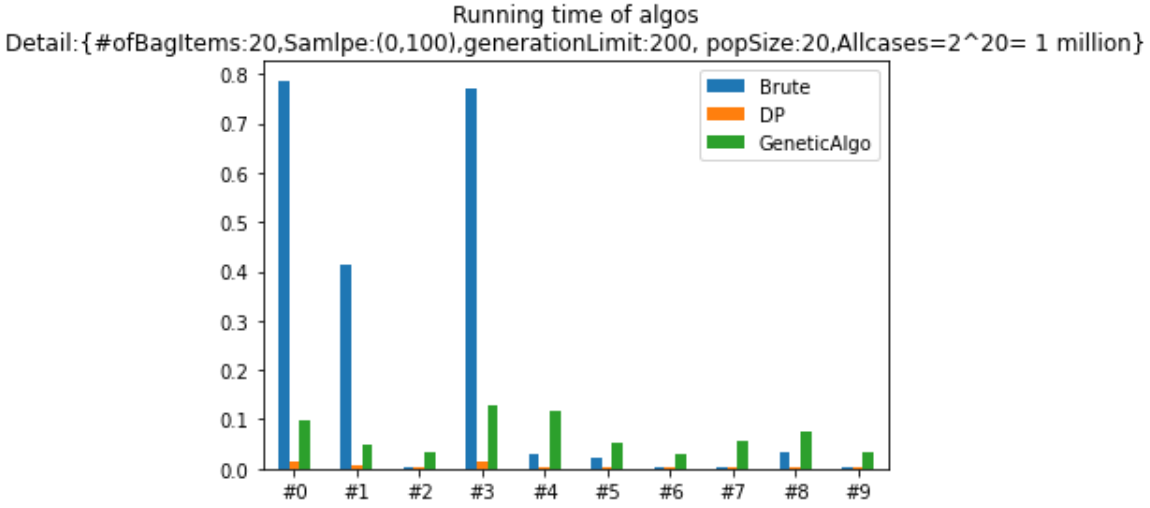
Genetic algorithm -2-2

برای پیاده سازی و فهم این روش از منابعی که در مراجع به آن ها اشاره کرده ام استفاده کرده ام.

این روش به طور هوشمندانه ای از جواب های مطلوب فعلی جواب های مطلوب تری پیدا میکند. به روش های آن Elitism میگوند.

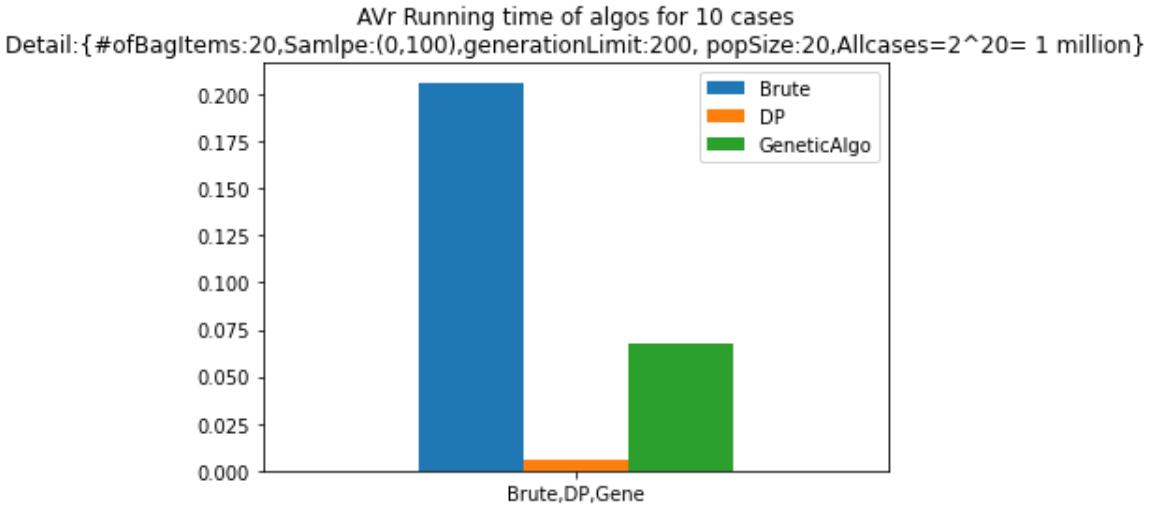
این الگوریتم را با داینامیک پروگرمینگ و بروت فورس از جنبه های مختلفی مقایسه کرده ام.

زمان اجرا روش های مختلف بر روی 10 تست کیس تصادفی



الگوریتم بروت فورس واریانس زیادی دارد و دلیل آن این است که در بعضی موارد الگوریتم شاخه های بیشتری را حذف میکند. مثلا زمانی که سایز کوله پشتی کم است و وزن داده ها زیاد، پس ناچارا خیلی از حالت ها که وزن آن ها زیاد میشوند حذف میشوند.

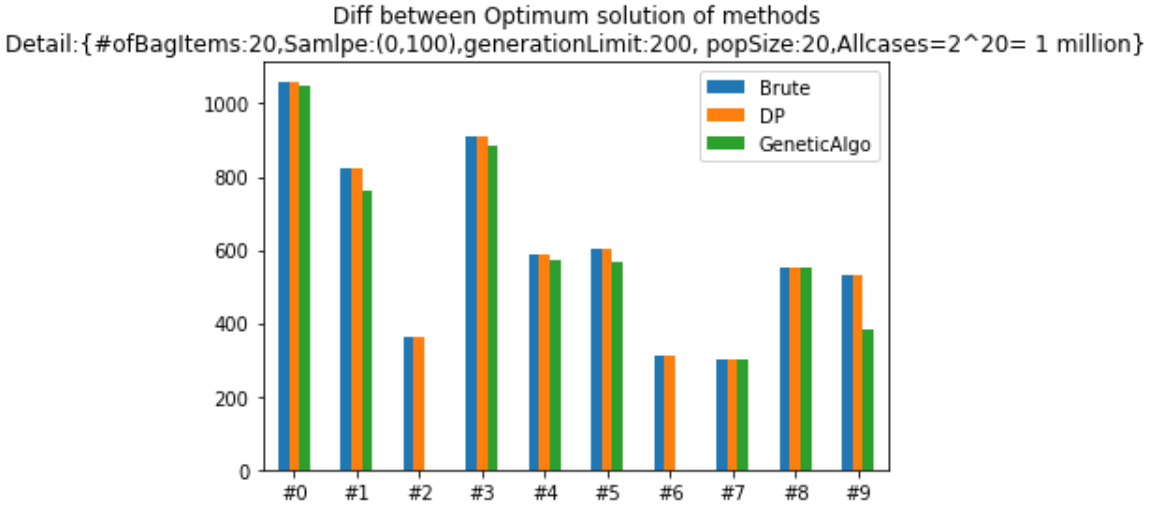
متوسط زمان اجرا بر روی 10 تمونه تصادفی



داینامیک پروگرمینگ درصورتی که وزن ها unconcentrated نباشد عملکرد مناسبی را دارد که در اینجا هم داده ها همین خاصیت را داشته اند.

زمان اجرای ژنتیک الگوریتم هم یک سوم بروت فورس بوده است که البته هرچه n بالا تر برود اختلاف زمان اجرای ژنتیک کمتر میشود.

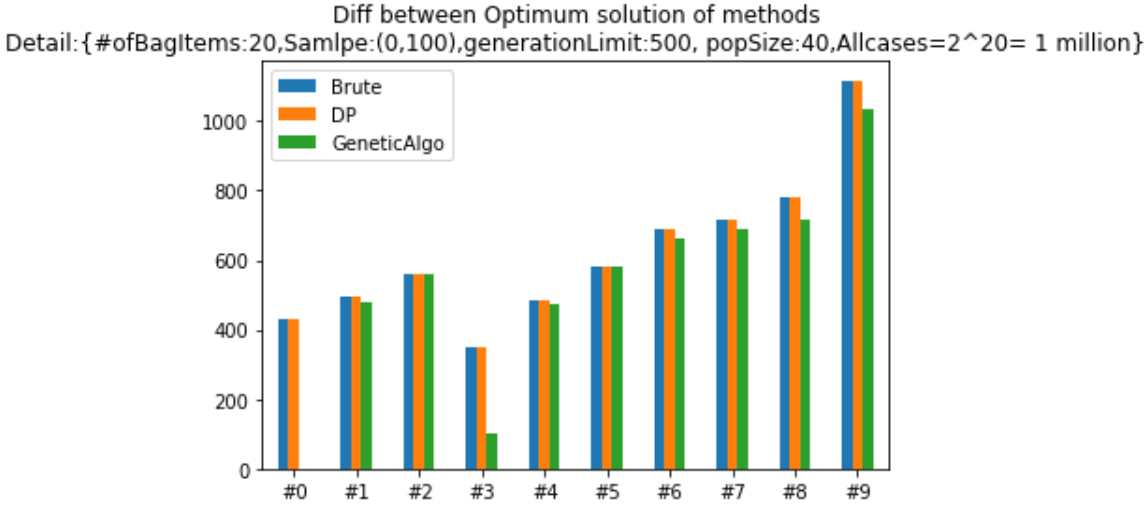
مقدار optimum value برای روش های مختلف



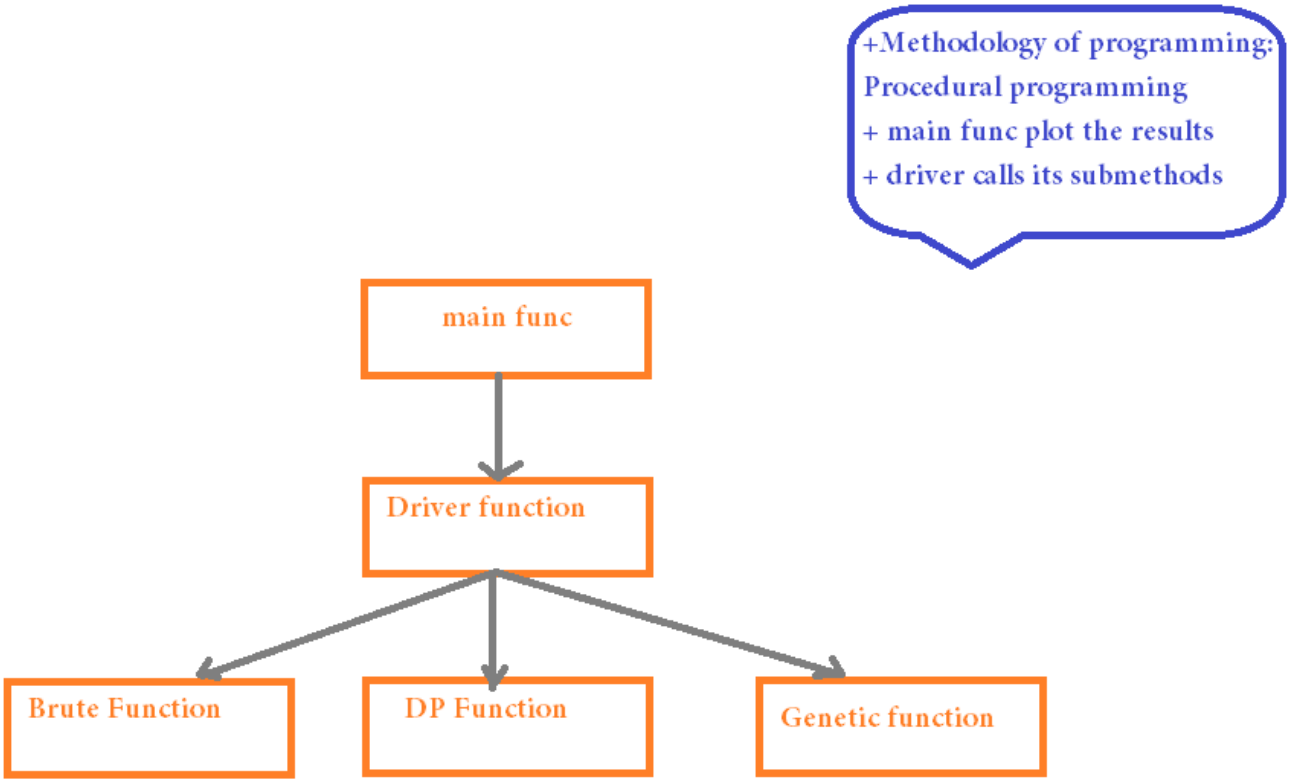
میبینم که ژنتیک الگوریتم تنها با محدودیت generation 200 و هر generation سایز 20 عدد ژنتیک توانسته در کیس های 0و1و3و4و5و9 نزدیک جواب بهیته برسد ولی در کیس 2 و 6

جوابی پیدا نکرده است. علت هم به خاطر کم بودن تعداد جنریشن های آن هست. اما در کیس شماره 7و8 هم دقیقا جواب بهیته را پیدا کرده است.

اگر تعداد limit generation را اندکی بالاتر ببریم حتی جواب های بهتری نیز خواهیم گرفت.



کد برنامه به زبان پایتون در انتهای گزارش و ساختار کلی برنامه خلاصه در زیر اورده شده است.



منابع و مراجع

برای پیاده سازی از این سایت الهام گرفته ام:

[sorting tow lists that refrence each other](https://stackoverflow.com/questions/9764298/how-to-sort-two-lists-which-reference-each-other-in-the-exact-same-way)

[unzip functin](https://stackoverflow.com/questions/19339/transpose-unzip-function-inverse-of-zip)

[1/0 Knapsack GeeksForGeeks](https://www.geeksforgeeks.org/python-program-for-dynamic-programming-set-10-0-1-knapsack-problem/?ref=rp)

[Dynamic programming explanation for 1/0 bag](https://www.youtube.com/watch?v=xCbYmUPvc2Q)

[Dynamic programming explanation for 1/0 bag](https://www.youtube.com/watch?v=nLmhmB6NzcM)

[Genetic algorithm explanation](https://www.youtube.com/watch?v=uQj5UNhCPuo)

[Genetic algorithm](https://www.youtube.com/watch?v=S5C_z1nVaSg)

[Genetic algorithm](https://www.youtube.com/watch?v=MacVqujSXWE)

[Float weight version of the 0/1 Bag problem](https://www.quora.com/How-do-I-work-around-with-knapsack-0-1-problem-when-weights-are-floating-point-numbers)

Appendix :

کد برنامه به زبان پایتون :

```
# Solving 0/1 knapsack problem with DP and Genetic algorithm and comp
aring them
```

```
#libraries
import random as rd
import time
import pandas as pd # for creating data frame and plotting data
import matplotlib.pyplot as plt
```

```
## Brute force
```

```
# A naive recursive implementation of 0-1 Knapsack Problem

# Returns the maximum value that can be put in a knapsack of
# capacity W
def BknapSack(W, wt, val, n):
  # Base Case
  if n == 0 or W == 0 :
    return 0

  # If weight of the nth item is more than Knapsack of capacity
  # W, then this item cannot be included in the optimal solution
  if (wt[n-1] > W):
    return BknapSack(W, wt, val, n-1)

  # return the maximum of two cases:
  # (1) nth item included
  # (2) not included
  else:
    return max(val[n-1] + BknapSack(W-wt[n-1], wt, val, n-1),
               BknapSack(W, wt, val, n-1))

# end of function knapSack
```

```
## DP algorithm
```

```
# A Dynamic Programming based Python
```

```
# Program for 0-1 Knapsack problem
# Returns the maximum value that can
# be put in a knapsack of capacity W
def DPknapSack(W, wt, val, n):
  K = [[0 for x in range(W + 1)] for x in range(n + 1)]

  # Build table K[][] in bottom up manner
  for i in range(n + 1):
    for w in range(W + 1):
      if i == 0 or w == 0:
        K[i][w] = 0
      elif wt[i-1] <= w:
        K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])
      else:
        K[i][w] = K[i-1][w]

  return K[n][W]
```

```
## Genetic algorithm
```

```
def generateGene(lenOfGene):
  return rd.choices([0,1], k =lenOfGene)
# will be used for generating initial population
def generatePopulation(PopSize, geneLen):
  return  [generateGene(geneLen) for _ in range(PopSize)]

def fitnessFunc(gene , vals , weights , weightLimit ):
  if ( (len(gene) != len(vals)) and (len(gene) != len(weights)) ):
    raise ValueError("genome must be of the lenght of values and weig
hts")
  v = 0 # value of the gene
  w = 0 # weight of the gene
  for i in range(len(gene)):
    if (gene[i]== 1):
      v += vals[i]
      w += weights[i]

      if w > weightLimit:
        return 0
  return v

def pairSelection(weightsOfGenes):
  #population = [ listOfGenes , theirFitness ]
```

```
  return rd.choices(population = [i for i in range(0,len(weightsOfGen
es))] ,
                    weights= weightsOfGenes,
                    k = 2 )
def crossOverFunc(geneA,geneB):
  # we cut them and cross them over at index i
  # i has randomly selected
  if ( len(geneA) != len(geneB) ):
    raise ValueError("Cross over func needs 2genes with the same leng
ht")

  if len(geneA)<2:
    return a,b # there is no point to cut gene with len =1

  cp = rd.randint(1, len(geneA)-1)
  return geneA[0:cp]+ geneB[cp:], geneB[0:cp] + geneA[cp:]

def mutation(gene, probability):
  #probability is a float number in [0,1]
  index = rd.randint(0,(len(gene)-1))
  gene[index] = gene[index] if rd.random()> probability else int(not
gene[index])
  return gene

def runEvolution(generationLimit, popSize , items,weightLimit):
  # Note: popSize should be even number and not odd
  # WEIGHTLIMIT IS THE BAG CAPACITY
  # items = [ [vals], [weights]]
  currentBestVal = 0
  totalGenerations = 0
  notChanging = 0 # num of generations that best solution has not cha
nged
  geneLen = len(items[0])
  population = [] # it will be = [ (gene , itsFitness)]

  # generate initial population & calculate their fitness
  genes = generatePopulation(popSize, geneLen)
  fitnesses=[fitnessFunc(g , items[0] , items[1] , weightLimit) for g
 in genes]
  population = list(zip(genes,fitnesses))
  #sorting population to become handy to work with them
  population.sort(key=lambda x: x[1],reverse=True)
  currentBestVal = population[0][1] # fitness of the best gene
  #iteration:
  while (notChanging < generationLimit):
```

```
    # running elitism ( replication+crossover+mutation (+...) )
    #1_replication:
    nextPopulation = population[0:2]
    #2&3_+crossover+mutation
    for _ in range(0,(popSize-2)//2):
      pi = pairSelection([b for a,b in population[2:]]) #parentsIndic
es
      # we need to increase indices by 2
      offspringA , offspringB = crossOverFunc(population[(pi[0]+2)][0
],population[(pi[1]+2)][0])
      offspringA = mutation(offspringA , 0.5)
      offspringB = mutation(offspringB , 0.5)
      fitnessA  = fitnessFunc(offspringA , items[0] , items[1] , weig
htLimit)
      fitnessB = fitnessFunc( offspringB , items[0] , items[1] , weig
htLimit)
      nextPopulation.append((offspringA,fitnessA))
      nextPopulation.append((offspringB,fitnessB))

    #update bestVal, notChanging, population, ++totalgeneration
    totalGenerations +=1
    population = nextPopulation
    #sorting
    population.sort(key=lambda x: x[1],reverse=True)
    if (currentBestVal >= population[0][1] ):
      notChanging +=1
    else:
      notChanging = 0
      currentBestVal = population[0][1]

    # if notChanging > generationLimit return currentBestVal,totalGen
erations

  return currentBestVal , totalGenerations
```

```
## Driver code
```

```
### Setting parameters and defining driver function
```

```
def driverFunc(values, weigths, generationLimit, popSize):
  val = values
```

```
  wt = weigths
  n = len(val)
  # capacity =4:Bag's capacity would be roughly 1/4 of objects' weigh
ts
  capacity  = rd.randint(1,10)
  W = int(sum(wt)/capacity  )
  Bt = 0
  DPt = 0
  Gt = 0
  Bresult =-1
  DPresult =-1
  Gresult=-1

  # starting time
  start = time.time()
  Bresult = BknapSack(W, wt, val, n)    #
  # end time
  end = time.time()
  Bt = end - start

  # starting time
  start = time.time()
  DPresult = DPknapSack(W, wt, val, n)    #
  # end time
  end = time.time()
  DPt = end - start

  # starting time
  start = time.time()
  bestval , numGeneration = runEvolution(generationLimit, popSize , [
 val, wt ] ,W)    #
  # end time
  end = time.time()
  Gt = end - start
  return Bresult,Bt,DPresult,DPt,bestval,numGeneration,Gt
```

```
## Illustration and evaluation
```

```
# Integer number for val and weight
##val = rd.sample(range(0, 100),numOfItems)
##wt = rd.sample(range(0, 100), numOfItems)

## testing Float numbers
#val = []
```

```
#wt = []
#for i in range(0, numOfItems):
#    # any random float between 50.50 to 500.50
#    # don't use round() if you need number as it is
#   x = round(rd.uniform(50.50, 500.50), 2)
#    y = round(rd.uniform(50.50, 500.50), 2)
#    val.append(x)
#    wt.append(y)
##

#Exaggerated weights dogs DP method(When data points are sparse or co
ncentrated)
#val = [1,1]
#wt = [1,100000000]
##
```

```
data = {'Function_names': ['Brute','DP','Genetic',],'result': []}
Bresult,Bt,DPresult,DPt,bestval,numGeneration,Gt= driverFunc([1,1],[1
,10000000],500, 10)
data['result'] = [Bt,DPt,Gt]
df = pd.DataFrame(data,columns=['Function_names','result'])
df.plot(x ='Function_names', y='result', kind = 'bar',title='Time usa
ge on an \nUnconcentrated or sparse data')
plt.show()
```

```
# running 10 times genetic algo and get a sense of how much we were n
ear to the optimum solution
# and also comparing its running times
```

```
# name+time = time of that funciton /  Name+res: result of that func
numOfItems = 20
Btime = []
DPtime = []
Genetime = []
Bres = []
DPres = []
Generes = []
index = ['#0', '#1', '#2','#3', '#4', '#5', '#6','#7', '#8','#9']
for i in range(0,10):
  val = rd.sample(range(0, 100),numOfItems)
  wt  =  rd.sample(range(0, 100), numOfItems)
  Bresult,Bt,DPresult,DPt,bestval,numGeneration,Gt=driverFunc(val,wt,
200, 20)
  Btime.append(Bt)
```

```
  DPtime.append(DPt)
  Genetime.append(Gt)
  Bres.append(Bresult)
  DPres.append(DPresult)
  Generes.append(bestval)
```

```
df = pd.DataFrame({'Brute': Btime,
                   'DP': DPtime , 'GeneticAlgo':Genetime}, index=ind
ex)
ax = df.plot.bar(rot=0,title='Running time of algos\nDetail:{#ofBagIt
ems:20,Samlpe:(0,100),generationLimit:200, popSize:20,Allcases=2^20=
1 million}')
```

```
df = pd.DataFrame({'Brute': (sum(Btime)/len(Btime)),
                   'DP':(sum(DPtime)/len(DPtime)), 'GeneticAlgo':(su
m(Genetime)/len(Genetime))}, index=['Brute,DP,Gene'])
ax = df.plot.bar(rot=0,title='AVr Running time of algos for 10 cases\
nDetail:{#ofBagItems:20,Samlpe:(0,100),generationLimit:200, popSize:2
0,Allcases=2^20= 1 million}')
```

```
df = pd.DataFrame({'Brute': Bres,
                   'DP': DPres , 'GeneticAlgo':Generes}, index=index
)
ax = df.plot.bar(rot=0,title='Diff between Optimum solution of method
s\nDetail:{#ofBagItems:20,Samlpe:(0,100),generationLimit:200, popSize
:20,Allcases=2^20= 1 million}')
```

```
#Seting Max number of genes that could be generated equal 1% of all c
ases
```

```
# name+time = time of that funciton /  Name+res: result of that func
numOfItems = 20
Btime = []
DPtime = []
Genetime = []
Bres = []
DPres = []
Generes = []
index = ['#0', '#1', '#2','#3', '#4', '#5', '#6','#7', '#8','#9']
```

```
for i in range(0,10):
  val = rd.sample(range(0, 100),numOfItems)
  wt  =  rd.sample(range(0, 100), numOfItems)
  Bresult,Bt,DPresult,DPt,bestval,numGeneration,Gt=driverFunc(val,wt,
500, 40)
  Btime.append(Bt)
  DPtime.append(DPt)
  Genetime.append(Gt)
  Bres.append(Bresult)
  DPres.append(DPresult)
  Generes.append(bestval)
```

```
df = pd.DataFrame({'Brute': Bres,
                   'DP': DPres , 'GeneticAlgo':Generes}, index=index
)
ax = df.plot.bar(rot=0,title='Diff between Optimum solution of method
s\nDetail:{#ofBagItems:20,Samlpe:(0,100),generationLimit:500, popSize
:40,Allcases=2^20= 1 million}')
```

```
## future developments
```

```
Modify "selection" function in "pairSelection" so that we don't opt o
ne gene twice
```