GA Python Code

Base code available for ISE/DSA 5113 Homework Assignment

GA Python Code Explanation

- These slides walk though highlights of the base Python code provided for the GA problem.
- The base problem is the same knapsack problem that you encountered in previous assignments.

GA-5113-students.py

```
#the intial framework for a binary GA
#author: Charles Nicholson
#for ISE/DSA 5113

#need some python libraries
import copy
import math
from random import Random
import numpy as np

#to setup a random number generator, we will specify a "seed" value
#need this for the random number generation -- do not change
seed = 51132021
myPRNG = Random(seed)

#to get a random number between 0 and 1, use this:             myPRNG.random()
#to get a random number between lwrBnd and upprBnd, use this:  myPRNG.uniform(lwrBnd,upprBnd)
#to get a random integer between lwrBnd and upprBnd, use this: myPRNG.randint(lwrBnd,upprBnd)

#number of elements in a solution
n = 150

#create an "instance" for the knapsack problem
value = []
for i in range(0,n):
    #value.append(round(myPRNG.expovariate(1/500)+1,1))
    value.append(round(myPRNG.triangular(150,2000,500),1))

weights = []
for i in range(0,n):
    weights.append(round(myPRNG.triangular(8,300,95),1))

#define max weight for the knapsack
maxWeight = 2500

#change anything you like below this line ------------------------------------
```

Same random number generation approach as seen previously.

Same problem setup as seen previously

Feel free to modify the main code as you please

```
#Student name(s):
#Date:

#monitor the number of solutions evaluated
solutionsChecked = 0


populationSize = 150 #size of GA population
Generations = 100   #number of GA generations

crossOverRate = 0.8  #currently not used in the implementation; neeeds to be used.
mutationRate = 0.05  #currently not used in the implementation; neeeds to be used.
eliteSolutions = 10  #currently not used in the implementation; neeed to use some type of elitism
```

You should probably change these

You should probably change these

Main GA Code

Most of the GA logic is here

Functions:
- initializePopulation()
- tournamentSelection() [or rouletteWheel()]
- breeding()
- insert()

```python
def main():
    #GA main code
    Population = initializePopulation()

    #optional: you can output results to a file -- i've commented out all of the file out put for now
    #f = open('out.txt', 'w')  #---uncomment this line to create a file for saving output

    for j in range(Generations):

        mates=tournamentSelection(Population,10)  #<--need to replace this with roulette wheel selection,
                                                   #e.g.:  mates=rouletteWheel(Population)
        Offspring = breeding(mates)
        Population = insert(Population, Offspring)

        #end of GA main code

        maxVal, meanVal, minVal, stdVal=summaryFitness(Population)     #check out the population at each generation
        print("Iteration: ", j, summaryFitness(Population))            #print to screen; turn this off for faster results

        #f.write(str(minVal) + " " + str(meanVal) + " " + str(varVal) + "\n")  #--uncomment this line to write to file

    #f.close()   #---uncomment this line to close the file for saving output

    print (summaryFitness(Population))
    bestSolutionInPopulation(Population)
```

computes and then prints population statistics for each generation

Returns best solution in population

```
#create initial population
#calls the "createChromosome" f
#adding each to a list of chromosomes (a.k.a., the "population")
def initializePopulation(): #n is size of population; d is dimensions of chromosome

    population = []
    populationFitness = []

    for i in range(populationSize):
        population.append(createChromosome(n))
        populationFitness.append(evaluate(population[i]))

    tempZip = zip(population, populationFitness)
    popVals = sorted(tempZip, key=lambda tempZip: tempZip[1], reverse = True)

    #the return object is a reversed sorted list of tuples:
    #the first element of the tuple is the chromosome; the second element is the fitness value
    #for example:  popVals[0] is represents the best individual in the population
    #popVals[0] for a 2D problem might be  ([1, 0], 483.3)
    #-- chromosome is the list [1, 0] and the fitness is 483.3

    return popVals
```

population: list of chromosomes
populationFitness: fitness values of associated chromosomes

create chromosome and add it to the list “population”
Evaluate the same chromosome and add its fitness to the list “populationFitness”

“zip” is a Python function – here it creates a single list that is composed of two lists

This logic can be very similar to the logic you use to create a single initial solution for the hill climber problems.

```python
#create an binary-valued chromosome
def createChromosome(d):
    #this code as-is expects chromosomes to be stored as a list, e.g., x = []
    #write code to generate chromosomes, most likely want this to be randomly generated

    x = []   #i recommend creating the solution as a list

    return x
```

Basic evaluation function is the same as before, except this time I did not return a list of "value" and "weight", but just the knapsack value. I created a separate function to compute the weight.

Regardless, you may want to use the weight to penalize infeasible solutions. This is up to you.

I also included a 'helper function' to determine the number of items selected in a chromosome.

```
#function to evaluate a solution x
def evaluate(x):

    a=np.array(x)
    b=np.array(value)

    totalValue = np.dot(a,b)     #compute the value of the knapsack selection

    #you will VERY LIKELY need to add some penalties or sometype of modification of the totalvalue to compute the chromosome fitness
    #for instance, you may include penalties if the knapsack weight exceeds the maximum allowed weight

    fitness  = totalValue

    return fitness   #returns the chromosome fitness
```

```
#function to compute the weight of chromosome x
def calcWeight(x):

    a=np.array(x)
    c=np.array(weights)

    totalWeight = np.dot(a,c)    #compute the weight value of the knapsack selection

    return totalWeight   #returns total weight
```

```
#function to determine how many items have been selected in a particular chromosome x
def itemsSelected(x):

    a=np.array(x)
    return np.sum(a)   #returns total number of items selected
```

Main GA Code

Most of the GA logic is here

Functions:
- initializePopulation()
- tournamentSelection() [or rouletteWheel()]
- breeding()
- insert()

computes and then prints population statistics for each generation

Returns best solution in population

```
def main():
    #GA main code
    Population = initializePopulation()

    #optional: you can output results to a file -- i've commented out all of the file out put for now
    #f = open('out.txt', 'w')  #---uncomment this line to create a file for saving output

    for j in range(Generations):

        mates=tournamentSelection(Population,10)  #<--need to replace this with roulette wheel selection,
                                                  #e.g.:  mates=rouletteWheel(Population)
        Offspring = breeding(mates)
        Population = insert(Population, Offspring)

        #end of GA main code

        maxVal, meanVal, minVal, stdVal=summaryFitness(Population)     #check out the population at each generation
        print("Iteration: ", j, summaryFitness(Population))            #print to screen; turn this off for faster results

        #f.write(str(minVal) + " " + str(meanVal) + " " + str(varVal) + "\n")  #--uncomment this line to write to file

    #f.close()   #---uncomment this line to close the file for saving output

    print (summaryFitness(Population))
    bestSolutionInPopulation(Population)
```

```python
#performs tournament selection;
#k chromosomes are selected (repeats allowed); best advances to the mating pool
#function returns the mating pool with size equal to the initial population
def tournamentSelection(pop,k):

    #randomly select k chromosomes; the best joins the mating pool
    matingPool = []

    while len(matingPool)<populationSize:

        ids = [myPRNG.randint(0,populationSize-1) for i in range(k)]
        competingIndividuals = [pop[i][1] for i in ids]
        bestID=ids[competingIndividuals.index(min(competingIndividuals))]
        matingPool.append(pop[bestID][0])

    return matingPool
```

- `ids` is a list of k randomly selected indexes selected for the “tournament”
- `competingIndividuals` is a list of the fitness values of the randomly selected individuals
- `bestID` is the index of the best individual of the tournament
- The best individual is added to the mating pool (another list);
- This is repeated until the mating pool size is the same as the population size

Main GA Code

Most of the GA logic is here

Functions:
- initializePopulation()
- tournamentSelection() [or rouletteWheel()]
- breeding()
- insert()

```
def main():
    #GA main code
    Population = initializePopulation()

    #optional: you can output results to a file -- i've commented out all of the file out put for now
    #f = open('out.txt', 'w')  #---uncomment this line to create a file for saving output

    for j in range(Generations):

        mates=tournamentSelection(Population,10)  #<--need to replace this with roulette wheel selection,
                                                  #e.g.:  mates=rouletteWheel(Population)
        Offspring = breeding(mates)
        Population = insert(Population, Offspring)

        #end of GA main code

        maxVal, meanVal, minVal, stdVal=summaryFitness(Population)     #check out the population at each generation
        print("Iteration: ", j, summaryFitness(Population))            #print to screen; turn this off for faster results

        #f.write(str(minVal) + " " + str(meanVal) + " " + str(varVal) + "\n")  #--uncomment this line to write to file

    #f.close()   #---uncomment this line to close the file for saving output

    print (summaryFitness(Population))
    bestSolutionInPopulation(Population)
```

computes and then prints population statistics for each generation

Returns best solution in population

```python
def breeding(matingPool):
    #the parents will be the first two individuals, then next two, then next two and so on

    children = []
    childrenFitness = []
    for i in range(0,populationSize-1,2):
        child1,child2=crossover(matingPool[i],matingPool[i+1])

        child1=mutate(child1)
        child2=mutate(child2)

        children.append(child1)
        children.append(child2)

        childrenFitness.append(evaluate(child1))
        childrenFitness.append(evaluate(child2))

    tempZip = zip(children, childrenFitness)
    popVals = sorted(tempZip, key=lambda tempZip: tempZip[1], reverse = True)

 return popVals
```

list of children
list of associated fitness values of children

Parent selection from mating pool
-- first two are chosen to mate
-- then next two are chosen to mate
-- etc.

Mutation is called here

Offspring appended to the children list

Offspring fitnesses appended to the childrenFitness list

same logic as applied to population in previous code

You need to write this logic

```
#implement a crossover
def crossover(x1,x2):

        #with some probability (i.e., crossoverRate) perform breeding via crossover,
        #i.e. two parents (x1 and x2) should produce two offsrping (offspring1 and offspring2)
        # --- the first part of offspring1 comes from x1, and the second part of offspring1 comes from x2
        # --- the first part of offspring2 comes from x2, and the second part of offspring2 comes from x1

        #if no breeding occurs, then offspring1 and offspring2 can simply be copies of x1 and x2, respectively



    return offspring1, offspring2  #two offspring are returned
```

```python
def breeding(matingPool):
    #the parents will be the first two individuals, then next two, then next two and so on

    children = []
    childrenFitness = []
    for i in range(0,populationSize-1,2):
        child1,child2=crossover(matingPool[i],matingPool[i+1])

        child1=mutate(child1)
        child2=mutate(child2)

        children.append(child1)
        children.append(child2)

        childrenFitness.append(evaluate(child1))
        childrenFitness.append(evaluate(child2))

    tempZip = zip(children, childrenFitness)
    popVals = sorted(tempZip, key=lambda tempZip: tempZip[1], reverse = True)

    return popVals
```

list of children
list of associated fitness values of children

Parent selection from mating pool
-- first two are chosen to mate
-- then next two are chosen to mate
-- etc.

Mutation is called here

Offspring appended to the children list

Offspring fitnesses appended to the childrenFitness list

same logic as applied to population in previous code

```python
#function to mutate solutions
def mutate(x):

    #NOTICE: i did not use the mutation rate nor mutate anything... fix this!

    return x
```

Main GA Code

Most of the GA logic is here

Functions:
- initializePopulation()
- tournamentSelection() [or rouletteWheel()]
- breeding()
- insert()

```
def main():
    #GA main code
    Population = initializePopulation()

    #optional: you can output results to a file -- i've commented out all of the file out put for now
    #f = open('out.txt', 'w')  #---uncomment this line to create a file for saving output

    for j in range(Generations):

        mates=tournamentSelection(Population,10)  #<--need to replace this with roulette wheel selection,
                                                  #e.g.:  mates=rouletteWheel(Population)

        Offspring = breeding(mates)
        Population = insert(Population, Offspring)

        #end of GA main code

        maxVal, meanVal, minVal, stdVal=summaryFitness(Population)     #check out the population at each generation
        print("Iteration: ", j, summaryFitness(Population))            #print to screen; turn this off for faster results

        #f.write(str(minVal) + " " + str(meanVal) + " " + str(varVal) + "\n")  #--uncomment this line to write to file

    #f.close()   #---uncomment this line to close the file for saving output

    print (summaryFitness(Population))
    bestSolutionInPopulation(Population)
```

computes and then prints population statistics for each generation

Returns best solution in population

```python
#insertion step
def insert(pop,kids):

    #this is not a good solution here...
    #essentially this is replacing the entire previous generation with the offspring
    #at least consider keeping the the best solution found so far?
    #maybe want to keep the top 5? 10? solutions from pop + kids? -- it's up to you.

    return kids
```

Computes the max, mean, min, and standard deviation of the fitness values of the population

```python
#perform a simple summary on the population:
#returns the best chromosome fitness, the average population fitness, and the variance of the population fitness
def summaryFitness(pop):
    a=np.array(list(zip(*pop))[1])
    return np.max(a), np.mean(a), np.min(a), np.std(a)


#the best solution should always be the first element...
def bestSolutionInPopulation(pop):
    print ("Best solution: ", pop[0][0])
    print ("Items selected: ", itemsSelected(pop[0][0]))
    print ("Value: ", pop[0][1])
    print ("Weight: ", calcWeight(pop[0][0]))
```

Prints statistics for the best solution in the population