Assignment 2

Modified Brute-Force Algorithms and Genetic Algorithms

Justin Self

AERO 500 Aerospace Modeling and Simulation

May 14, 2025

Problem 1

Task 1 (15 points):

Code - Develop a Modified Brute-Force Algorithm (MBFA) in Python. Results - Use you MBFA to solve the KNAPSACK problem with values and weights given below. Submit your code and a list of the three best solutions you found.

KNAPSACK Problem: You have 20 items with the following value/mass properties:

$$\text{values} = [23, 21, 8, 1, 3, 7, 18, 19, 17, 15, 24, 22, 6, 28, 4, 2, 27, 20, 5, 10] \text{ dollars}$$

$$\text{weights} = [7, 2, 6, 9, 1, 5, 6, 1, 3, 4, 7, 9, 3, 7, 3, 4, 5, 1, 5, 4] \text{ kg}$$

The total weight of the knapsack is limited to 45 kg

Solution.

Best results were obtained after running 1 million Monte Carlo trials. The best solution obtained a total value of 215 dollars·kg with a total mass of 45 kg, and the item list given by the binary selection array (1 for select, 0 for do not select). The full code for this task is included in Section 1.

Run	Total Value, $·kg	Total Mass, kg	Selection Array
1	213	45	[1 1 0 0 1 0 0 1 1 1 1 0 1 1 0 0 1 1 0 1]
2	215	44	[1 1 0 0 1 0 1 1 1 1 1 0 0 1 0 0 1 1 0 0]
3	215	44	[1 1 0 0 1 0 1 1 1 1 1 0 0 1 0 0 1 1 0 0]

Table 1: Best Results to the Knapsack Problem using 1M MC iterations and MBFA

1 Full Code: Task 1

```python
# TASK 1
print("~~~ TASK 1 ~~~")
import random
#random.seed(10)
max = 0
randIndex = []
numbers = []
for i in range(10):
    numbers.append(random.randint(0,100))
for n in numbers:
    if (n > max):
        max = n

for i in range(5):
    randIndex.append(random.randint(0,9))

for t in randIndex:
    if (numbers[t] > max):
        max = numbers[t]
```

```

#print(numbers)
#print(randIndex)
#print(max)

# knapsack
values = [23, 21, 8, 1, 3, 7, 18, 19, 17, 15, 24, 22, 6, 28, 4, 2, 27, 20, 5, 10]
      # in dollars
weights = [7, 2, 6, 9, 1, 5, 6, 1, 3, 4, 7, 9, 3, 7, 3, 4, 5, 1, 5, 4]
      # kg
maxMass = 45 # maximum allowable mass (weight?)

# maximize value, subject to mass constraint
# no minimum number of items

candidate = []
numTrials = 1000000 # 1 mill
import numpy as np

for i in range(numTrials): # super loop
    # assign random 0s and 1s to vector of length(values)
    itemIndex = np.random.randint(0,2,size=len(values))

    # Figure out total mass from the selection
    massTotal = np.dot(itemIndex,weights)

    # Total value from the selection
    valueTotal = np.dot(itemIndex,values)

    if massTotal <= maxMass:
        candidate.append((valueTotal,massTotal,itemIndex.copy())) # candidate
    solutions, copy the initial setup list

# Sort and print
candidate.sort(reverse=True,key=lambda x:x[0])
#print(candidate)
print("The best solution over " + str(numTrials) + " trials is: ")
print("Total value ($), total mass (kg), item choices [array]")
print(*candidate[0])


```

Algorithm 1: Task 1

Problem 2

Task 2 (30 points):

Code - Develop a simple Genetic Algorithm (GA) in Python using the template supplied. Results - Use your GA to solve the KNAPSACK problem with values and weights given below. Graph - Submit your code and a graph of the average population and best member fitness versus trial for 50 generations. Discussion - Run your code several times and comment on how the GA performs on different runs, and for runs with different numbers of generations. To use the GA module, you simply need to have the two Python files in the same folder. You DO NOT need to use pip.

Solution. The plot shown in Figure 1 was developed using the following parameters:

Mutation rate = 0.1 (helped to slow the convergence rate for larger population sizes to see interesting features)

Generations = 50

Population size = 1000

Number of survivors = 100

A maximum fitness value of 215 was observed across many simulations which experimented with changing the population size, mutation rate, number of generations, and number of survivors after each selection process. Lower population size yielded lower and non-optimal fitness value. Mutation rate didn't affect much for large population sizes (test range: 0.0001 to 0.5) but for smaller population sizes, too small a mutation rate resulted in never achieving the optimal solution. Number of generations had a notable affect on the convergence time of the response (fitness value) to the optimal value. High number of generations essentially assured convergence to the optimal solution (215), assuming the number of survivors after each selection process was sufficient. Number of survivors after each selection had a significant effect on the simulation convergence to the optimal solution. For example, for a population size of 1000, mutation rate of 0.1, 100 generations, and a survival number per selection of 100, the simulation converged to the optimal value in three generations. When number of survivors was reduced to 50, the solution converged in four generations. When number of survivors was reduced to 25, the solution did not converge to the optimal fitness value, but did converge (in six generations) to the optimal value at 28 survivors per selection.

Assuming low number of survivors means higher computational efficiency, in this configuration, the best ratio of survivors to total initial population size was about 0.028. A representative group of testing with different parameters is shown in Table 2. The last run at a population of 10000 was extremely slow to run, which shows the effect on number of survivors (1500 is too high) on an otherwise very fast-running and high-performing (convergence in two generations) simulation. It is not recommended to run this configuration due to very slow run time.

Table 2: Representative Results from Genetic Algorithm Tests

Population Size	Generations	Mutation Rate	Number of Survivors	Solution	Gens to Converge
10	50	0.25	3	174	15
100	50	0.25	30	215	7
1000	50	0.25	300	215	3
10000	50	0.25	300	215	2
10000	50	0.25	1500	215	2

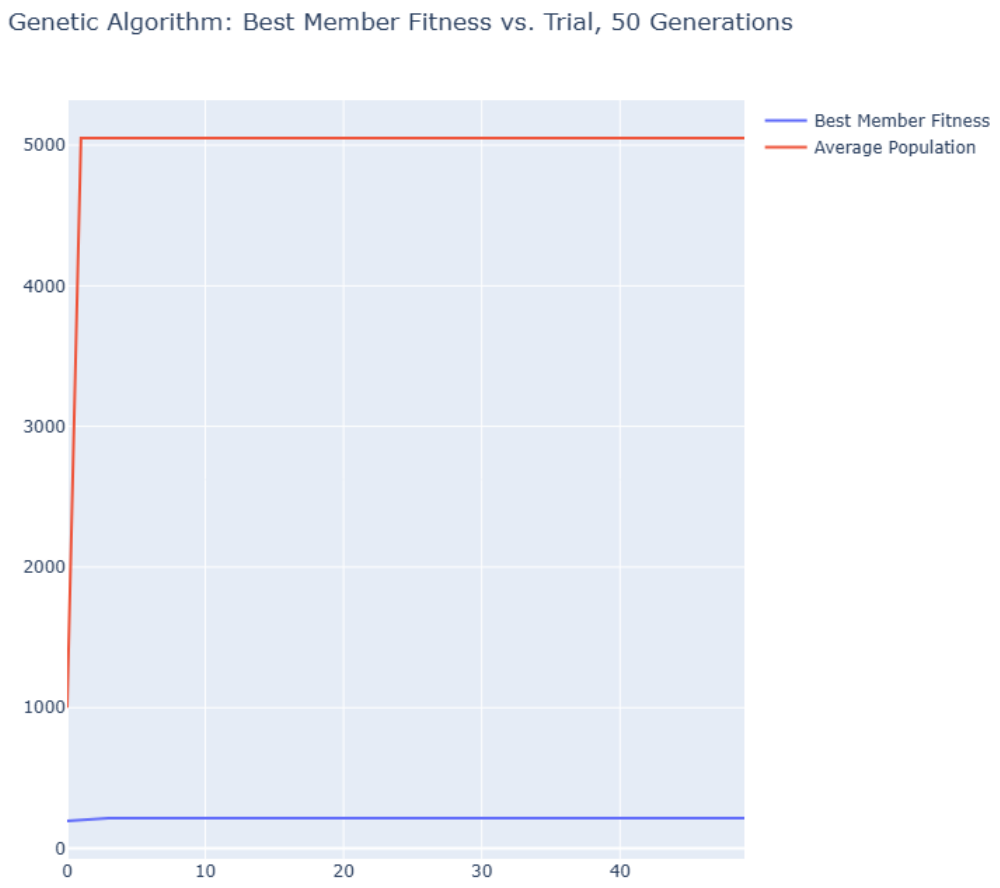

Figure 1: Genetic Algorithm Solution to the Knapsack Problem

The full script is given in Section 2 and the GA.py file containing classes and their attributes is given in Section 3.

2 Full Code: Task 2 Script

```
print("~~~ TASK 2 ~~~")
import GA
import plotly.express as px
```

```
import plotly.graph_objects as go
import pandas as pd

# Step 1 - Initialize Population
myPop = GA.Population(populationSize = 1000, numGenes = 20)

for c in myPop.members:
    c.calculateFitness()


# loop here through numGenerations
numGenerations = 50
p = 0.5
y = []
x = []
averagePop = []

for i in range(numGenerations):
    # # # Step 2 - Selection
    #ratio = 0.25 # take best 25%, rounded
    numSurv = 100
    parents = myPop.selection(numSurv)
    #if i % 300 == 0: # print only every 20
    print("Best Parent Fitness is: ",parents[0].fitness,myPop.getCount())

    # for plotting
    y.append(parents[0].fitness)
    x.append(i)
    averagePop.append(myPop.getCount())

    # Parents have a child (test)
    children = []
    for momIndex in range(len(parents)):
        for dadIndex in range(momIndex + 1,len(parents)):
            newChild = parents[momIndex] + parents[dadIndex]
            newChild.mutate(p)
            newChild.calculateFitness()
            children.append(newChild)
            #print("Breeding gives:", momIndex,dadIndex)

    myPop.setMembers(parents+children)

 # pop of 1e4 works best
fig = go.Figure(layout_title_text="Genetic Algorithm: Best Member Fitness vs.
    Trial, " + str(numGenerations) + " Generations")
fig.add_trace(go.Scatter(x = x,y = y,mode='lines',name='Best Member Fitness'))
fig.add_trace(go.Scatter(x = x,y = averagePop,mode='lines',name='Average
    Population'))
fig.write_html('Figtitle.html', auto_open=True)

```

Algorithm 2: Task 2- Script

3 Full Code: Task 2-GA.py

```python
import numpy as np
import random

# # # added #
values = [23, 21, 8, 1, 3, 7, 18, 19, 17, 15, 24, 22, 6, 28, 4, 2, 27, 20, 5, 10]
weights = [7, 2, 6, 9, 1, 5, 6, 1, 3, 4, 7, 9, 3, 7, 3, 4, 5, 1, 5, 4]
maxWeight = 45

class Chromosome:

    def __init__(self, numGenes):
        self.genes = random.choices([0, 1], k = numGenes)
        self.fitness = 0
        self.numGenes = numGenes

    def mutate(self,p):
        # Implement random mutation
        # input = some probability of mutation happening (0.01)
        chances = random.random()
        if chances < p:
            # pick bit
            bit = random.randint(0,self.numGenes-1)
            # flip the gene value of this bit
            self.genes[bit] ^=1 # exclusive OR # stackOverflow for this!!


    def __add__(self, other): # self = Mom, other = Dad
        # Implement single point crossover with random crossover point
        cPoint = random.randint(0,self.numGenes-1)
        # need to grab Mom and Dad genes on either side of cPoint
        # make a baby
        momGenes = self.genes[0:cPoint]
        dadGenes = other.genes[cPoint:]
        baby = Chromosome(self.numGenes)
        baby.genes = momGenes + dadGenes
        return baby

    def calculateFitness(self):
        # chrom.fitness = some function of chrom.genes
        totalValue = np.dot(self.genes,values)
        totalWeight = np.dot(self.genes,weights)

        if totalWeight <= maxWeight:
            self.fitness = float(totalValue)
        else:
            self.fitness = 0

class Population:

    def __init__(self, populationSize, numGenes):
        self.members = [Chromosome(numGenes) for i in range(populationSize)]

```

```
53      def selection(self, numSurv):
54          # Implement Selection
55          # Step 1 - Sort members by fitness
56          self.members.sort(key=lambda x:x.fitness,reverse=True)
57          # Step 2 - return some number of members based on the ratio provided
58          #numSurv = int(len(self.members)*ratio)
59          return self.members[0:numSurv]
60
61      def setMembers(self,chromosomes):
62          self.members = chromosomes
63
64      def getCount(self):
65          return len(self.members)
66
```

Algorithm 3: Task 2-GA.py

Problem 3

Task 3 (5 points):

Compare and contrast the two solution methods, MBFA and GA. Comment on how each method converges on a solution versus number of trials/generations. Also, provide your thoughts on why one method may work better than the other for solving this problem.

Solution. The modified brute-force and genetic algorithms are both useful tools to simulate and find optimal solutions to some problems. The simulations developed in this project exemplified how powerful even the simplest forms of these algorithms can be to an optimization problem. The advantages of the MBF algorithm was its simplicity in implementation, but it required a high number of Monte Carlo trials to converge to the optimal solution. Several runs had to be performed to converge on the solution, as seen in **??** , which took three full trials (3 million MC trials) to obtain the solution twice.

The genetic algorithm was more complicated to implement, but the level of complexity was overwhelmed by its relative robustness. Several parameters can be changed within a certain reasonable range and the optimal solution was still found within a few generations. There was no need for anywhere near 1 million trials; the best solutions found in this project only considered, at most, a population size of 10,000 with only 20 genes each, and convergence was observed at even modest number of generations. Several tests converged in less than three, sometimes only two, generations (population size: 10000, mutation rate 0.2, number of survivors per selection: 250 converged in two generations).

The MBFA works simply by producing as combinations as possible, then sorting out the candidate solutions by whether they are below the weight value or not. The solutions, as many as there are, are then sorted with the highest value solution being the winner. This works, but you simply have to run a large number of MC trials until the solver "happens upon" what is the real optimal solution. The GA, on the other hand, is much more efficient since it chooses and keeps only the best candidates at every generation. When the population size is large enough, there are large numbers of possible child states that can produce the optimal solution that the best candidates are created more readily than in the totally stochastic MBFA model. Adding mutation into the scheme allows for more genetic diversity which in turn creates space for the best candidates to be created even sooner.