ASSIGNMENT 3

Name: Michael Black
Student Number: 221402063
Date: 20/09/2023

I declare that this is my own, original work.

Signature: 

IMPLEMENTATION DETAILS

Stopping Conditions: 1000 generations
Initialization Procedure: random float values between 0 and 20
Values for population size investigated: 50-500
Values for mutation rate investigated: 0.01-0.1
Values for mutation magnitude investigated: random values between 0-10
Values for selection method investigated: Roulette wheel selection , r(0,1]

RESULTS

Number of iterations (typically): 1000
Largest Profit found: 650683.57
Optimal amount of grams of each alloy:

Alloy	**Number of grams**
Adamantium	12.97
Unobtainium	8.95
Dilithium	18.63
Pandemonium	15.11
Delerium	10.98
Millennium	10.52
Premium	11.97

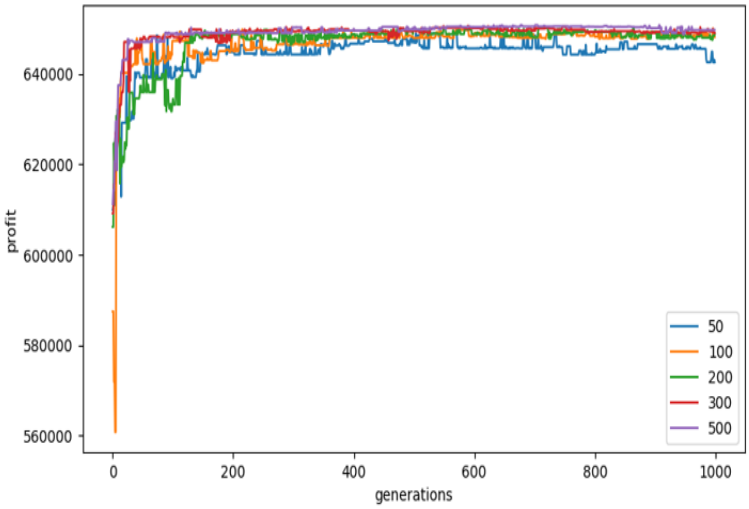

Figure 2 Varying population Size

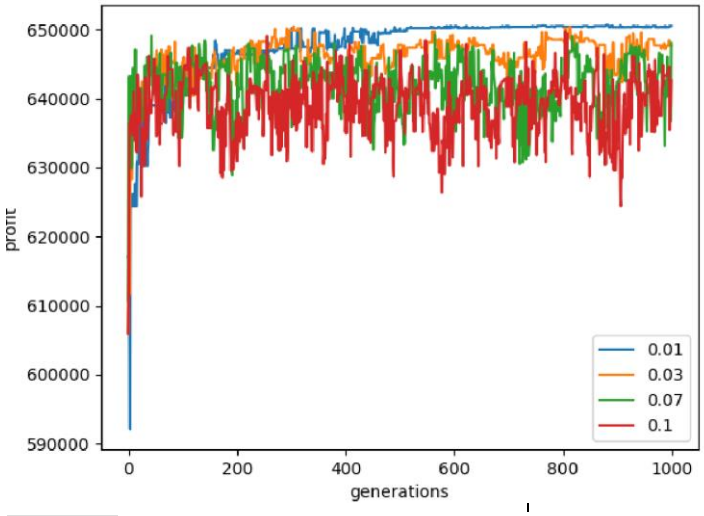

Figure 1 Varying mutation rate

OBSERVATIONS

Increasing the population size resulted in improved profit calculation due to increased diversity. Larger populations did result in slower generations since more calculations had to be done by the computer. The computational load was significantly decreased when the profit values were normalized.

Two point crossover proved to be more effective that single point crossover. Initialization of the population affects the convergence rate of the population. After trial and error, the solution space is determined to be somewhere between 0g and 20g. Initializing the population to values in this range allowed for a better initial guess as well as quicker convergence.

With high mutation rates, the profit value oscillates, this is due to the changes being too large and altering dna objects with good genes too much. The fitness values had to be normalized to obtain good results. Since profit can be directly related to fitness it is possible to use the profit values as is, however, for large values normalizing the profit values seemed to improve the results and performance by a large margin.

Random mutation proved to be the simplest and most effective way to implement mutations. Using static values for mutating magnitude did not work effectively in this case. The mutations of a gene in a dna object depended on the probability of the mutation(mutation rate). The magnitude is a random value between 0 and 10, any larger magnitudes than in this range had negative effects on the convergence of the population.

Roulette wheel section was favored over random selection. Roulette wheel selection selected individuals based on their fitness. Therefore, stronger individuals had a larger probability of being selected than lower fitness values.

```
import numpy as np
import math
import random
import matplotlib.pyplot as plt



one = [4,8,12,16,20,24,28] #Test DNA objects
two = [11,10,9,8,7,6,5]

DNA1 = np.array(one)
DNA2 = np.array(two)

DNA1.astype(float)
DNA2.astype(float)

sell_price = [11172, 6309, 14646, 10386, 8287, 10049, 8040]
materials = [[0.2, 0.7, 0.1], [0.3, 0.2, 0.5], [0.8, 0.1, 0.1], [0.1, 0.5,
0.4], [0.7, 0.1, 0.2], [0.4, 0.4, 0.2], [0.2, 0.6, 0.2]]
elec = [36, 20, 19, 16, 19, 38, 19]
plt_ratios = [0.2, 0.3, 0.8, 0.1, 0.7, 0.4, 0.2]

plt_ratios = np.array(plt_ratios)
mat = np.array(materials)
elec = np.array(elec)
sell = np.array(sell_price)

def Calc_Profit(dna):
    #Calculate Material Cost
    total_mat = np.dot(dna, materials) # returns 1x3 array where each element
represents total material in grams

    #MATERIAL COST
    #Calculate Plutonium
    plt_price = (1200 + 10*total_mat[0])*total_mat[0]

    #Calculate Radium
    rad_q = total_mat[1] #Store quantity of radium
    plt_floor = math.floor(total_mat[0]) #Rounds down grams of plutonium

    rad_q = rad_q - plt_floor #Subtracts from the amount of radium since for
each gram of plut, one gram of rad is free
```

```
    if(rad_q < 0):
        rad_q = 0

    rad_price = rad_q*300

    #Calculate Uranium
    if(total_mat[2] > 8):
        ur_price = (720*total_mat[2])  #10% discount

    if(total_mat[2] <= 8):
        ur_price = (800*total_mat[2])

    total_material_cost = plt_price+ur_price+rad_price


    #ELECTRICITY COST
    kWh = np.dot(dna, elec)
    elec_cost = math.exp(0.005*kWh)

    #STORAGE COST
    storage = 0
    for line in dna:
        storage += 10*line**3

    #PROFIT
    total = storage + elec_cost + total_material_cost

    sell_price = np.dot(sell, dna)

    profit = sell_price - total
    return profit

def roulette_wheel_selection(population, profits):

    i = 0

    total_fitness = sum(profits)
    probability = profits[i]/total_fitness

    cumulative = probability
    r = random.uniform(0,1)

    while cumulative < r:
        i+=1
        cumulative += profits[i]/total_fitness

    return population[i]
```

```
def two_point_crossover(parent1, parent2):
    #Create 2 random crossover points
    cp1 = random.randint(0,7)
    cp2 = random.randint(0,7)

    #Create mask
    offspring_1 = np.zeros(7)
    offspring_2 = np.zeros(7)

    #Crossover elements
    offspring_1[0:cp1] = parent1[0:cp1]
    offspring_1[cp1:cp2] = parent2[cp1:cp2]
    offspring_1[cp2:7] = parent1[cp2:7]

    offspring_2[0:cp1] = parent2[0:cp1]
    offspring_2[cp1:cp2] = parent1[cp1:cp2]
    offspring_2[cp2:7] = parent2[cp2:7]

    return offspring_1, offspring_2

def mutate(dna, mutation_rate):
    mutated = dna.copy() #prevents changing original individual
    for j in range(0,7):
        u = random.uniform(0,1)
        if u < mutation_rate:
            mutated[j] = random.uniform(0,20) #Place a random value
    return mutated


#initialize population
size = 300
# Initialize an empty list to store the arrays
population = []

#Create initial population
for _ in range(size):
    random_array = np.random.uniform(1, 15, size=7)
    population.append(random_array)



#Create array of values for matplotlib
total_profits1 = []
total_profits2 = []
total_profits3 = []
total_profits4 = []
```

```
gen = np.arange(1000)

generations = 1000



#initialize population
size = 300
# Initialize an empty list to store the arrays
population = []

#Create initial population
for _ in range(size):
    random_array = np.random.uniform(1, 15, size=7)
    population.append(random_array)
for k in range(0, generations):

#Calculate fitness
    profits = []
    for x in population:
        y = Calc_Profit(x)
        y = np.array(y)
        profits.append(y)

    maxProfit = np.max(profits)
    total_profits1.append(maxProfit)

    maxIndex = np.argmax(profits)
    #print(k, ": ", maxProfit)
    print(k, " Profit: ", maxProfit, " " ,population[maxIndex])

    #normalise fitness values, for maximiation problem we normalize to values
between 0 and 1
    minProfit = min(profits)
    profits = [(x-minProfit)/(maxProfit-minProfit) for x in profits]


    #Select and create new population
    x = 0
    selected = []
    for _ in range(size):
       selected_individual = roulette_wheel_selection(population, profits)
       selected.append(selected_individual)


    #Two point crossover
    population = []
    for j in range(0, size-1):
```

```
        offspring1, offspring2 = (selected[j], selected[j+1])
        population.append(offspring1)
        population.append(offspring2)

    mutation_rate = 0.01

    mutated_population = []

    for m in population:
        mutated_population.append(mutate(m, mutation_rate))

    population = mutated_population


plt.plot(gen, total_profits1, label='0.01')


plt.xlabel('generations')
plt.ylabel('profit')
plt.legend(loc='lower right')
plt.show()
```