CS7267 Programming Assignment #4 (Fall 2019)

By Neeraj Sharma

Overview

As per the Assignment following two problems were addressed using Genetic Algorithm

1. Knapsack Problem.
2. Problem of our own – I have implemented Maximization problem.

Program Run Results

================knapsack problem===========

Best Population:

[0, 1, 0, 0, 1, 1, 1]

Fitness of the best population: 28

Mutation Probability: 0.1

Crossover probability: 0.6

Stopping Criterion: 150 rounds

================Maximization problem========

Best Population:

[1, -3, 1, 3, -1, 3]

Fitness of the best population: 32

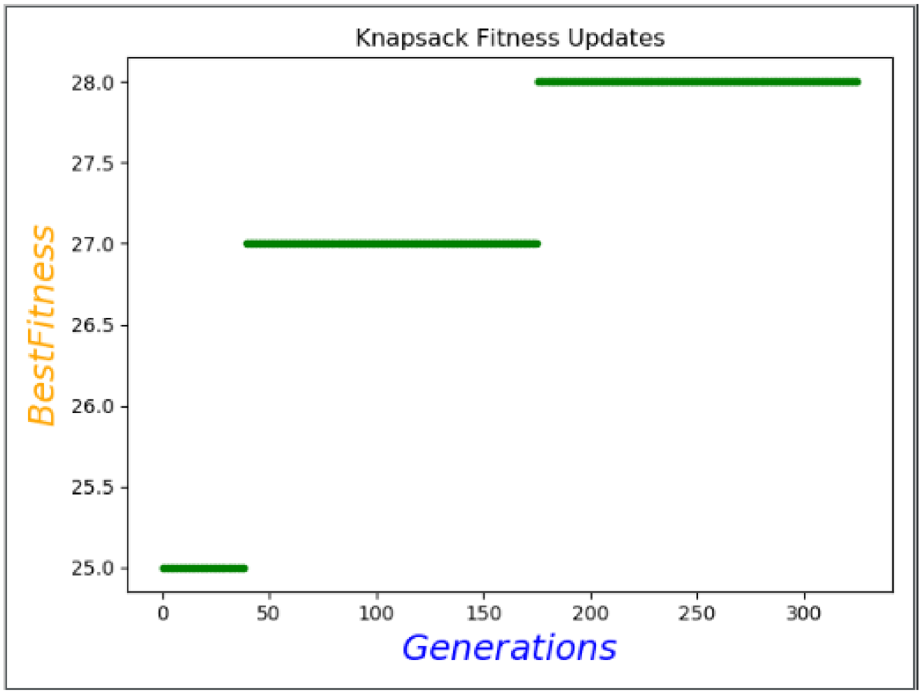
Mutation Probability: 0.1

Crossover probability: 0.6

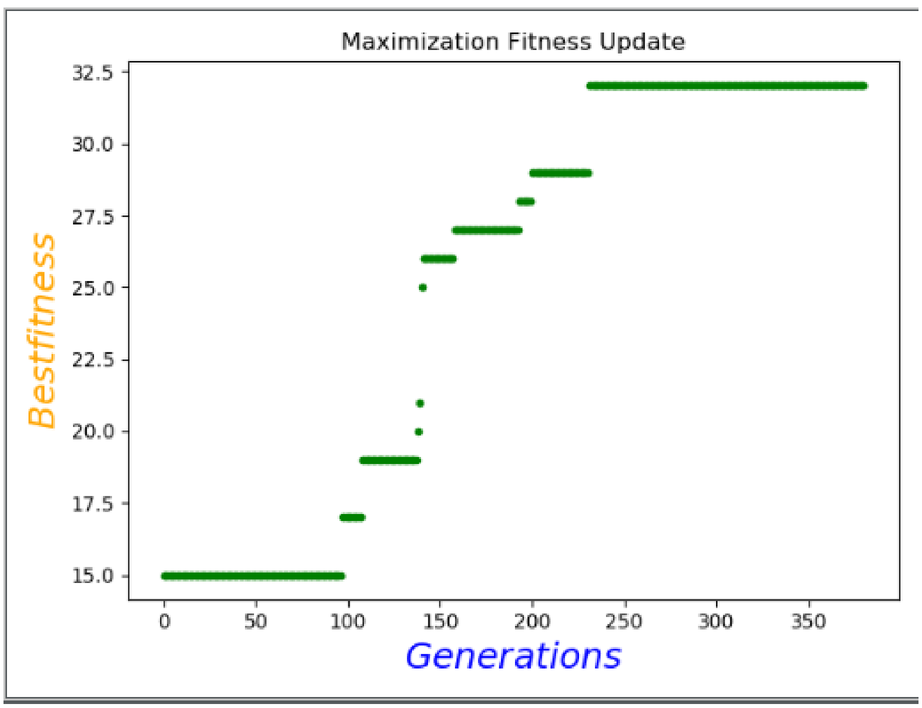
Stopping Criterion: 150 rounds

Explanation: **Knapsack Graph** as per the graph, we can see highest fitness 28 was achieved after around 160 generations.

Maximization Graph – as per the graph we can see highest fitness 32 was achieved after around 225 generations.

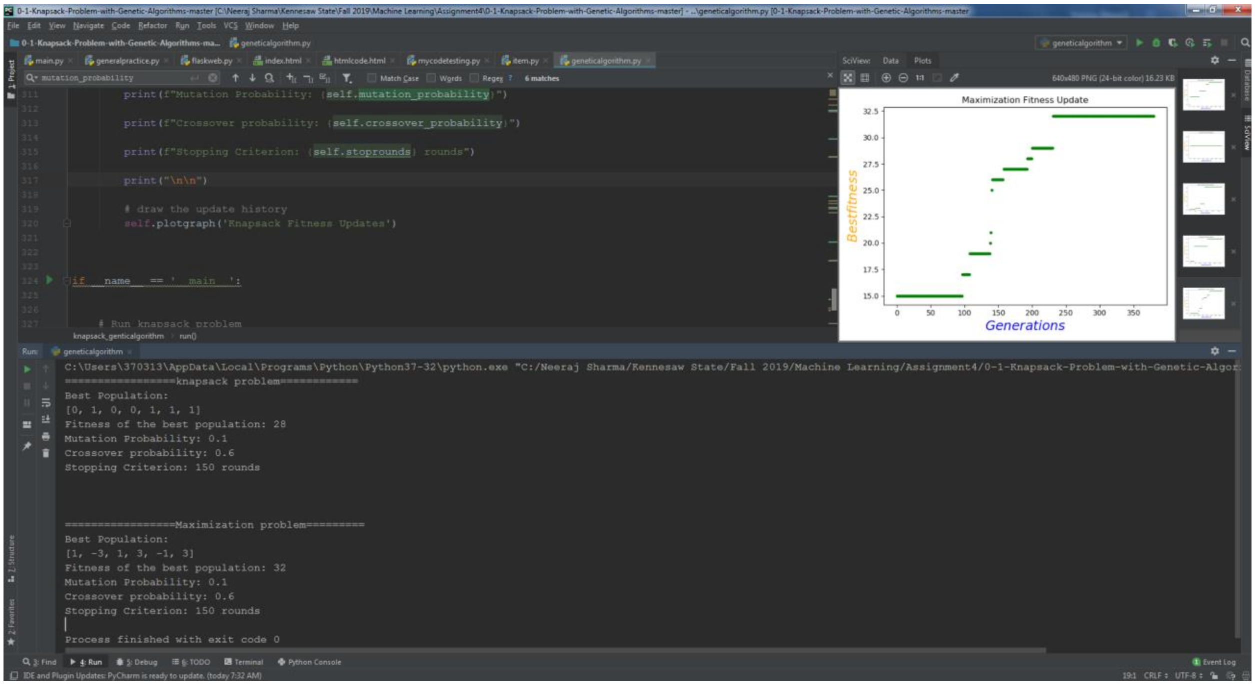


Knapsack Fitness Updates
28.0
27.5
27.0
26.5
26.0
25.5
25.0
BestFitness
0
50
100
150
200
250
300
Generations



Maximization Fitness Update
32.5
30.0
27.5
25.0
22.5
20.0
17.5
15.0
Bestfitness
0
50
100
150
200
250
300
350
Generations

Screen shots of the program run in the editor



Program Code:

Single program below covers both problem using two different classes.

```
#Assignment 4: Genetic Algorithm
#Name: Neeraj Sharma


import numpy as np
from random import randint
import random
import matplotlib.pyplot as plt
```

```python
class maximization_geneticalgorithm:
    def __init__(self):
        self.inputs = [2, -4, 5, 3, -1, 1]

        # initial crossover probability
        self.crossover_probability = 0.6

        # initial mutation probability
        self.mutation_probability = 0.1

        #poulation declaration
        self.populations = None

        # Each iteration number of population
        self.num_population = 6

        # fitness history initialized
        self.fitness_history = []

        # criteria to stop
        self.stoprounds = 150

    # function to generate initial population
    def generatepopulation(self):
        new_population = np.random.randint(low=-4, high=4,
size=(self.num_population, 6))
        self.populations = new_population

    # function to calculate the fitness for each population
    def calculatefitness(self, population):
        # print(f"population created {population}")
        fitness = np.sum(np.array(population) * self.inputs)
        # print(f"fitness of the population {fitness}")
        return fitness

    # Function to perform crossover
    def crossover(self, population1, population2):
        if random.random() < self.crossover_probability:
            population1_lefthalf = population1[:2]
            population1_righthalf = population1[2:]

            population2_lefthalf = population2[:2]
            population2_righthalf = population2[2:]

            newpop1 = list(population1_lefthalf) +
```

```
list(population2_righthalf)
            newpop2 = list(population2_lefthalf) +
list(population1_righthalf)

            return newpop1, newpop2
        else:
            return population1, population2

    # function to perform mutation
    def mutation(self, population):
        if random.random() < self.mutation_probability:
            mutation_point = randint(0, 5)
            population[mutation_point] = random.randint(-4, 4)
            return population

        return population

    # function for stop criteria
    def stop(self):

        # stop program if the best fitness doesn't improve for
200 rounds on row
        if len(self.fitness_history) < self.stoprounds:
            return False
        best_fitness = self.fitness_history[-self.stoprounds][1]
        for fitness_history_ in self.fitness_history[-
(self.stoprounds-1):]:
            fitness = fitness_history_[1]
            if fitness > best_fitness:
                return False
        return True

    # function to create graph
    def plotgraph(self, title):
        history = []
        for ele in self.fitness_history:
            history.append(ele[1])
        plt.plot(range(len(self.fitness_history)), history,
'g.')
        plt.title(title)
        plt.xlabel('$Iterations$', fontsize=18, color= "BLUE")
        plt.ylabel("$Best fitness$", rotation=90, fontsize=18,
color= "ORANGE")
        plt.show()

    def run(self):
```

```
        # create initial populations
        self.generatepopulation()

        # check the stop criterion
        while not self.stop():

            newpop = []

            # calculate each item's fitness in new population
            fitness = []
            for population in self.populations:

fitness.append(self.calculatefitness(population))
            # print(f"last 6 fitness {fitness}")
            # keep the best two items
            best_two_population_index =
sorted(range(len(fitness)), key=lambda i: fitness[i],
reverse=True)[:2]

            best_first_population =
self.populations[best_two_population_index[0]]
            best_second_population =
self.populations[best_two_population_index[1]]

            newpop.append(best_first_population)
            newpop.append(best_second_population)

            # put the best one in the fitness_history list
            best_first_fitness =
fitness[best_two_population_index[0]]
            self.fitness_history.append((best_first_population,
best_first_fitness))

            # list of populations for crossover and mutation
            crossovermutateindex = [elem for elem in range(6) if
elem not in best_two_population_index]
            crossovermutatelist =
list(np.array(self.populations)[crossovermutateindex])

            # do the crossover and mutation
            newpop1, newpop2 =
self.crossover(crossovermutatelist[0], crossovermutatelist[1])
            newpop1 = self.mutation(newpop1)
            newpop2 = self.mutation(newpop2)

            newpop3, newpop4 =
self.crossover(crossovermutatelist[2], crossovermutatelist[3])
```

```
            newpop3 = self.mutation(newpop3)
            newpop4 = self.mutation(newpop4)

            newpop.append(newpop1)
            newpop.append(newpop2)
            newpop.append(newpop3)
            newpop.append(newpop4)

            # update the populations
            self.populations = newpop

        # print out the final result
        best_population = self.fitness_history[-
self.stoprounds][0]
        best_fitness = self.fitness_history[-self.stoprounds][1]

        print('Best Population: ')
        print(best_population)

        print('Fitness of the best population: ')
        print(best_fitness)

        # draw the update history
        self.plotgraph("Maximization Fitness Update")


class knapsack_genticalgorithm:
    def __init__(self):
        # initialize crossover probability
        self.crossover_probability = 0.6

        # initialize mutation probability
        self.mutation_probability = 0.1

        #initialize populations
        self.populations = []

        # define number of populations for each iterations
        self.num_population = 6

        #initialize weights and benefits as per the problem
given in assignment
        self.weights = [7, 8, 4, 10, 4, 6, 4]
        self.benefits = [5, 8, 3, 2, 7, 9, 4]

        # initialize fitness history
        self.fitness_history = []
```

```
        # define stop criterian
        self.stoprounds = 150

    # function to generate the initial population
    def generatepopulation(self):
        for _ in range(self.num_population):
            population_ = []
            for _ in range(7):
                population_.append(randint(0, 1))
            self.populations.append(population_)

    # function to calculate the fitness
    def calculatefitness(self, population):
        total_weight = sum([item*weight for item, weight in
zip(population, self.weights)])
        if total_weight > 22:
            fitness = -1
        else:
            fitness = sum([item*benefit for item, benefit in
zip(population, self.benefits)])
        return fitness

    # function to perform crossover
    def crossover(self, population1, population2):
        if random.random() < self.crossover_probability:
            population1_lefthalf = population1[:2]
            population1_righthalf = population1[2:]

            population2_lefthalf = population2[:2]
            population2_righthalf = population2[2:]

            newpop1 = list(population1_lefthalf) +
list(population2_righthalf)
            newpop2 = list(population2_lefthalf) +
list(population1_righthalf)

            return newpop1, newpop2
        else:
            return population1, population2

    # Function to perform mutation
    def mutation(self, population):
        if random.random() < self.mutation_probability:
            mutation_point = randint(0, 6)
            if population[mutation_point] == 1:
                population[mutation_point] = 0
```

```
            else:
                population[mutation_point] = 1
            return population

        return population

    # function to define stop criteria
    def stop(self):
        # stop program if the best fitness doesn't improve for
200 rounds on row
        if len(self.fitness_history) < self.stoprounds:
            return False
        best_fitness = self.fitness_history[-self.stoprounds][1]
        for fitness_history_ in self.fitness_history[-
(self.stoprounds-1):]:
            fitness = fitness_history_[1]
            if fitness > best_fitness:
                return False
        return True

    # function to create graph
    def plotgraph(self, title):
        history = []
        for ele in self.fitness_history:
            history.append(ele[1])
        plt.plot(range(len(self.fitness_history)), history,
'g.')
        plt.title(title)
        plt.xlabel('$Iterations$', fontsize=18, color = "BLUE")
        plt.ylabel("$Best Fitness", rotation=90, fontsize=18,
color= "ORANGE")
        plt.show()

    def run(self):
        # create initial populations
        self.generatepopulation()

        # check if the stop criterion
        while not self.stop():

            newpop = []
            # calculate each item's fitness in new population
            fitness = []
            for population in self.populations:

fitness.append(self.calculatefitness(population))
```

```
            # keep the best two items
            best_two_population_index =
sorted(range(len(fitness)), key=lambda i: fitness[i],
reverse=True)[:2]

            best_first_population =
self.populations[best_two_population_index[0]]
            best_second_population =
self.populations[best_two_population_index[1]]

            newpop.append(best_first_population)
            newpop.append(best_second_population)

            # put the best one in the fitness_history list
            best_first_fitness =
fitness[best_two_population_index[0]]
            self.fitness_history.append((best_first_population,
best_first_fitness))

            # list of populations for crossover and mutation
            crossovermutateindex = [elem for elem in range(6) if
elem not in best_two_population_index]
            crossovermutatelist =
list(np.array(self.populations)[crossovermutateindex])

            # do the crossover and mutation
            newpop1, newpop2 =
self.crossover(crossovermutatelist[0], crossovermutatelist[1])
            newpop1 = self.mutation(newpop1)
            newpop2 = self.mutation(newpop2)

            newpop3, newpop4 =
self.crossover(crossovermutatelist[2], crossovermutatelist[3])
            newpop3 = self.mutation(newpop3)
            newpop4 = self.mutation(newpop4)

            newpop.append(newpop1)
            newpop.append(newpop2)
            newpop.append(newpop3)
            newpop.append(newpop4)

            # update the populations
            self.populations = newpop

        # print out the final result
        best_population = self.fitness_history[-
self.stoprounds][0]
```

```python
        best_fitness = self.fitness_history[-self.stoprounds][1]

        print('Best Population: ')
        print(best_population)

        print('Fitness of the best population: ')
        print(best_fitness)

        # draw the update history
        self.plotgraph('Knapsack Fitness Updates')



if __name__ == '__main__':


    # Run knapsack problem
    print("=================knapsack problem============" )
    geneticalgorithm = knapsack_genticalgorithm()
    geneticalgorithm.run()

    # Function maximization problem
    print("=================Maximization problem=========")
    geneticalgorithm = maximization_geneticalgorithm()
    geneticalgorithm.run()
```