Qiao Liu 146800980
Duc Nguyen 160246450
Rob Fedorowicz 170783690
Stevie Gallow 170405240
Simon Arcila 130804570

CP468 Term Project: Simple Genetic Algorithm

Algorithm Design

We used Python 3 to implement a Simple Genetic Algorithm (SGA), as we have easy access to many useful tools, such as the built-in random function, and data visualization (matplotlib & seaborn).

1. Input value

We store the inputs of the given objective function (OF) as a 10-bit binary number, which is one integer from one to 1023. The values must be reduced according to the domain of OF's input values. An example of this, is the Himmelblau function: The OF has two input values (x1 & x2), with value_num = 2. This means our string must have a length of 20 if we want to store the binary values. Since we calculate the OF on the domain of [-4 , -4] for both inputs, we must change the original range to get the approximate input domain (Changed range: (0 – 512) / 128, (1024 – 512) / 128).

2. Structure of chromosome

The class chromosome is used to store the information of one chromosome, and the values of the given OF:

```
class Chromosome:
    def __init__(self, string):
        self.string = string
        self.x = self.get_x()
        self.y = self.get_y()
        self.fitness = self.get_fitness()
        self.p = 0
    def get_x(self):
        if of == 'Himmelblau':
            # x1 x2 domain: -4 <= x1, x2 <= 4
```

```
                return [(int(self.string[i:i + value_len], 2) - 512) / 114
                        for i in range(0, len(self.string), value_len)]
        elif re.match('Test*', of):
                return [-1 if elem == '0' else 1 for elem in self.string]

    def get_y(self):
        if of == 'Himmelblau':
                return (self.x[0] ** 2 + self.x[1] - 11) ** 2 + (self.x[0] + self.x[1] ** 2 -
7) ** 2
        elif re.match('Test*', of):
                return abs(sum(elem[0] * elem[1] for elem in
list(combinations(self.x, 2))))

        def get_fitness(self):
                if is_max:
                        return self.y
                else:
                        return 1 / (1 + self.y)
```

We use a string variable to store the binary values of each input, randomly generated by the built-in random function. We put the decoded and scaled values in a list X, and we use Y as the output value of the OF. We use ‘itertools.combinations()’ to retrieve all pairs of the elements within X. This is where we introduce the fitness, which affects our Y value according to is_max. Our goal is to always maximize fitness, regardless of if we want to maximize/minimize Y. To check the maximum value of the OF, set is_max=True.

3. Roulette in reproduction

There are three steps in a Simple Genetic Algorithm: First, we’ll look at reproduction. This is where new generations are created by breeding a portion of the population. We use roulette for our fitness selection method. We start by getting the proportion of fitness (p) using the fitness / sum_fitness functions and place all values in a list. We then generate a new list, where each element is changed to be in the upper bound of the fitness proportion of the chromosome. When deciding which chromosome will be selected, we generate a random value between 0 and 1, and choose depending on that value.

4. Details in crossover & mutation

Crossover and mutation is used when creating the next generation of the next population. We treat the binary input as a string, which is a combination of values separated by j, which is the dimension of the objective function. An example of this, using the Himmelblau's function: Each chromosome has value_len = 20 (2-dimensional function). The next step is to generate a random value between 1 – 19 to separate the values for crossover. Now, each string would consist of j values. If they are split, you will see the values changed. Mutation follows the same idea, but with only 1 / j values go through a mutation. We found that as j increases to 7, crossover and mutation has little effect, and we find the simple genetic algorithm fails to find a global maximum of fitness, regardless of the probability of a crossover/mutation.

5. Elitist strategy

Mathematicians have proven that SGA does not converge in a probabilistic sense. In order to get it to converge on an optimal solution we can use an elitist strategy which puts the elite (one chromosome that has the largest fitness within the population) directly into the next generation while avoiding crossover and mutation. This can increase its computing speed on a large scale. Because the best solution is always sent to the next generation, the maximum fitness never decreases during its entire iteration.

In the program, two elites are chosen (ensure an even number of chromosomes remaining) at the beginning of the reproduce step and place them in the first two positions of the entire populations list, then these two elites are skipped in roulette, mating, crossover and mutation steps.

```
elite1 = population[elite1_index]
elite2 = population[elite2_index]
population[elite1_index] = population[0]
population[elite2_index] = population[1]
population[0] = elite1
population[1] = elite2
for i in range(2, n):
    # roulette steps
for i in range(2, n):
```

```
        # crossover
for i in range(2, n):
        # mutation
```

Part 2: Running & Parameter Controls

Our program has eight parameters and we tested the SGA using all the test OFs given by professor as well as seven benchmark OFs found online as well as with. Because different OFs require different parameters to find the optimal solution, we've included the parameters and OFs in a dictionary for convenience:

```
# key: objective function name

# value: [n, value_num, value_len, pc, pm, iteration, is_max]

param_dict = {
        'GramacyLee': [100, 1, 10, 0.5, 0.005, 20, False],
        'Beale': [100, 2, 10, 0.8, 0.01, 500, False],
        'GoldsteinPrice': [100, 2, 10, 0.8, 0.005, 100, False],
        'Himmelblau': [100, 2, 10, 0.8, 0.01, 20, False],
        'DeJong': [100, 10, 10, 1.0, 0.002, 1000, False],
        'Rosenbrock': [100, 7, 10, 1.0, 0.05, 1000, False],
        'Rastrigin': [100, 7, 10, 1.0, 0.02, 1000, False],
        'Test10': [100, 1, 10, 0.8, 0.01, 45, False],
        'Test11': [100, 1, 11, 0.8, 0.01, 55, False],


        'Test26': [100, 1, 26, 0.8, 0.01, 325, False],
        'Test27': [100, 1, 27, 0.8, 0.01, 351, False],

}
```

1. Running instructions

Our program used Python 3 along with two extended data visualization packages: matplotlib and seaborn. In order to run and test our program, all that is required is to assign the name of one OF at the start of the programs entrance. The program will then use default parameters found in param_dict. It changes them to a valid range to see different results.

```
if __name__ == '__main__':
    # choose an objective function from
    # 'GramacyLee' 'Beale' 'GoldsteinPrice' 'Himmelblau' 'DeJong' 'Rosenbrock'
    ‘Rastrigin'
    # or 'Test10' to 'Test27'
    of = ‘GramacyLee'
    # assign other values to parameters below if you want
    n = param_dict[of][0]
    value_num = param_dict[of][1]
    value_len = param_dict[of][2]
    pc = param_dict[of][3]
    pm = param_dict[of][4]
    iteration = param_dict[of][5]
    is_max = param_dict[of][6]
    # SGA iteration starts . . .
```

2. Parameter Controls

n is the population size that is a constant during iteration. The population size determines the density of the gene schemata. The normal range of n is 0 to 100. Having a large n value may be a burden on computation without yielding a better solution. In our programs default configuration, n's value is set to 100 for all OFs.

value_len and value_num, both of which have been mentioned previously, varies from different OFs. Since our program fixes the length of one encoded value to 10 in the seven OFs found online, value_len is 10 times the dimension of each OF. For three n-dimensional OFs, the value of value_num is set to 10 for DeJong's function and the other two are set to 7. With the OFs given by the professor, value_num is set to 1 and value_len is the same as the number of variables.

The performance of the SGA optimization problem depends on the balance of the depth and breadth of search space (the range of possible solutions) where mutation probability (pm) and crossover probability (PC) are important factors among them. The higher the pc is, the faster the SGA can converge to the optimal region, and at the same time is the higher the probability those highly adaptive schemata can be removed. Elitist strategy also deals with this problem as well. However, if the PC is too small, the searching progress can slow down or the algorithm might not even converge. Usually, the pc is chosen among [0.5, 0.9]. There is a similar situation for mutation probability: having a very small pm may not generate new schemata, leading to genetic drift and premature convergence while the large pm can turn SGA into a randomly searching algorithm. The recommended range for the optional pm is [0.001, 0.05], although there is no guidance for what the optimal choice may be.

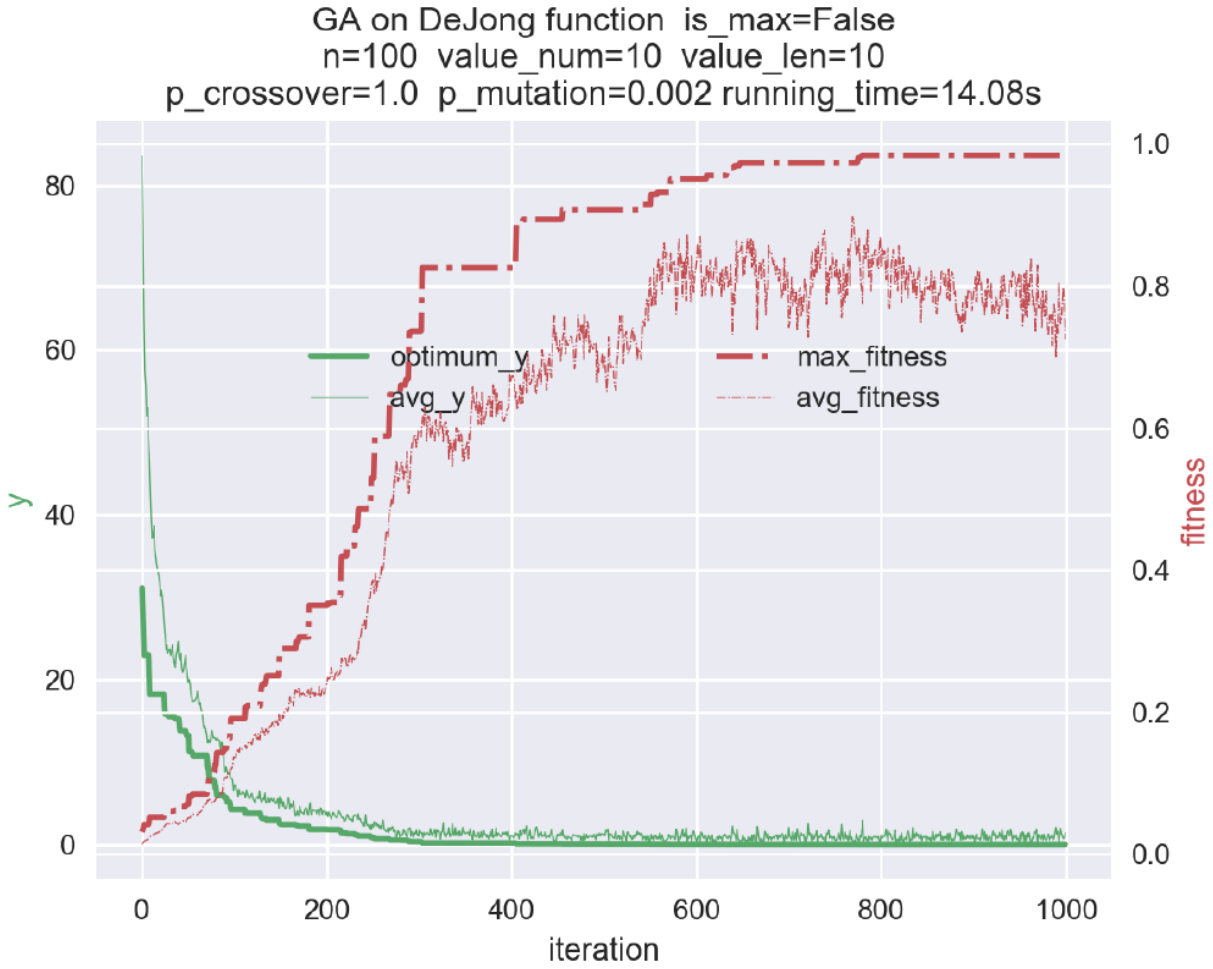
Iteration can be determined much easier than the pc and pm due to the fact that if the curve of fitness is still going up at the end of the iteration, the SGA does not get convergence yet. We already know that the global minimum for y is 0 and the global maximum for fitness is 1 for all six OFs within the programs. If the result is far from the target value, the iteration should continue. For SGA without a complex OF, the iteration is usually several hundred or less.

Part 3: Results on OFs

Results of SGA on all OFs is found within the Results folder. Each result has a figure showing how fitness and y change during iteration, as well as a text file showing the population at the end. The first two lines in the text file show the maximum of fitness and the optimum of y. Here we only show four of all OFs.

1. DeJong function

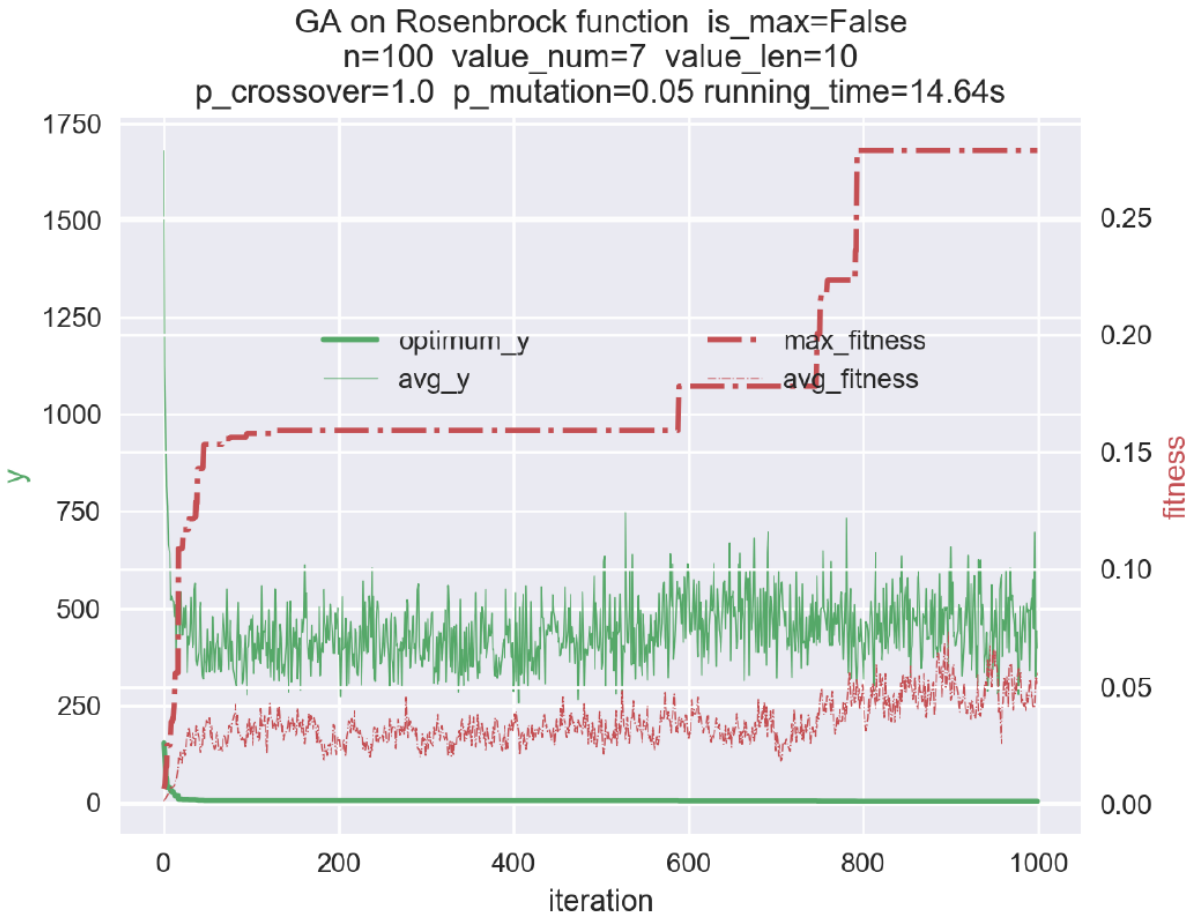
SGA converges at around iteration 800;
optimum_y decreases from around 32 to 0.02;
max_fitness increases from around 0.02 to 0.98.



2. Rosenbrock function

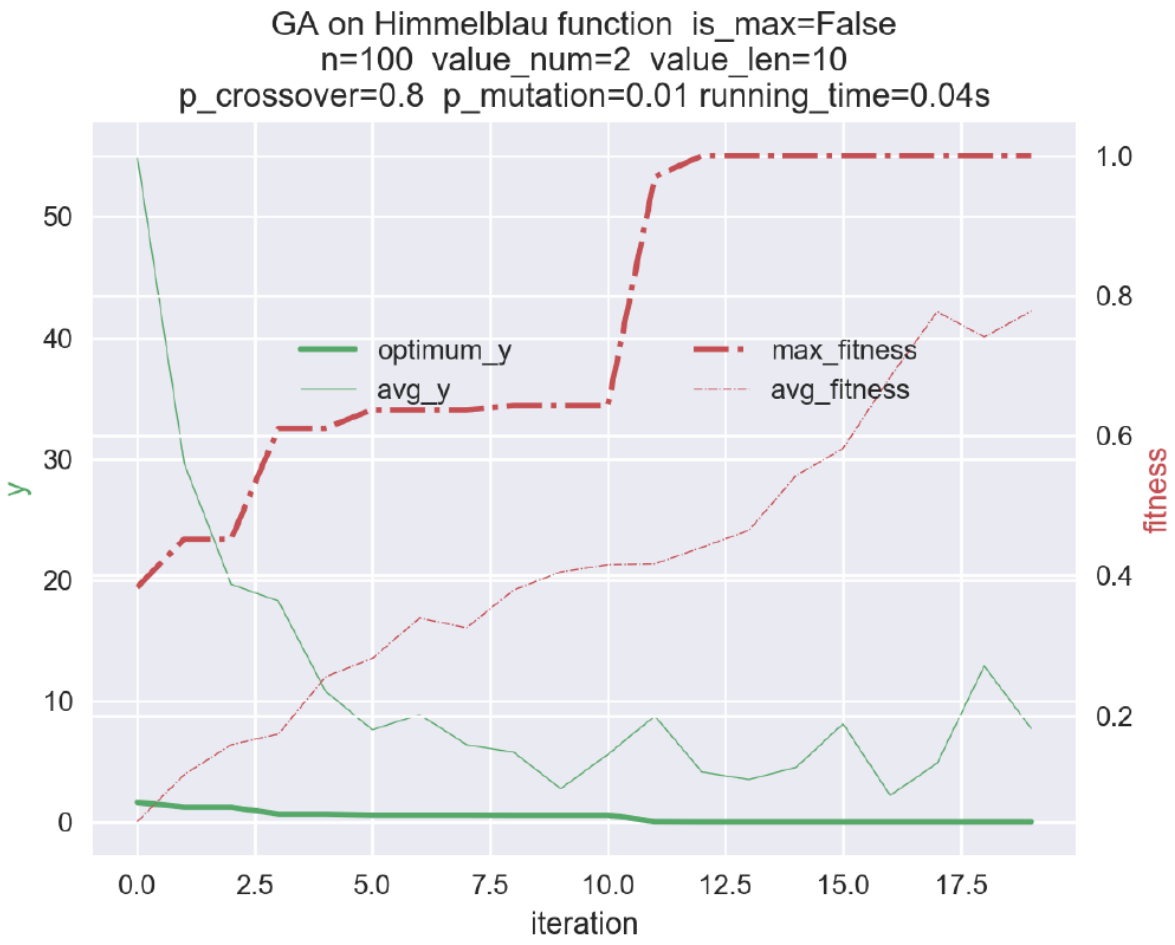
SGA converges at around iteration 800;
optimum_y decreases from 150 to 2.59;
max_fitness increases from 0.01 to 0.28, which is far from 1 in that due to the fitness

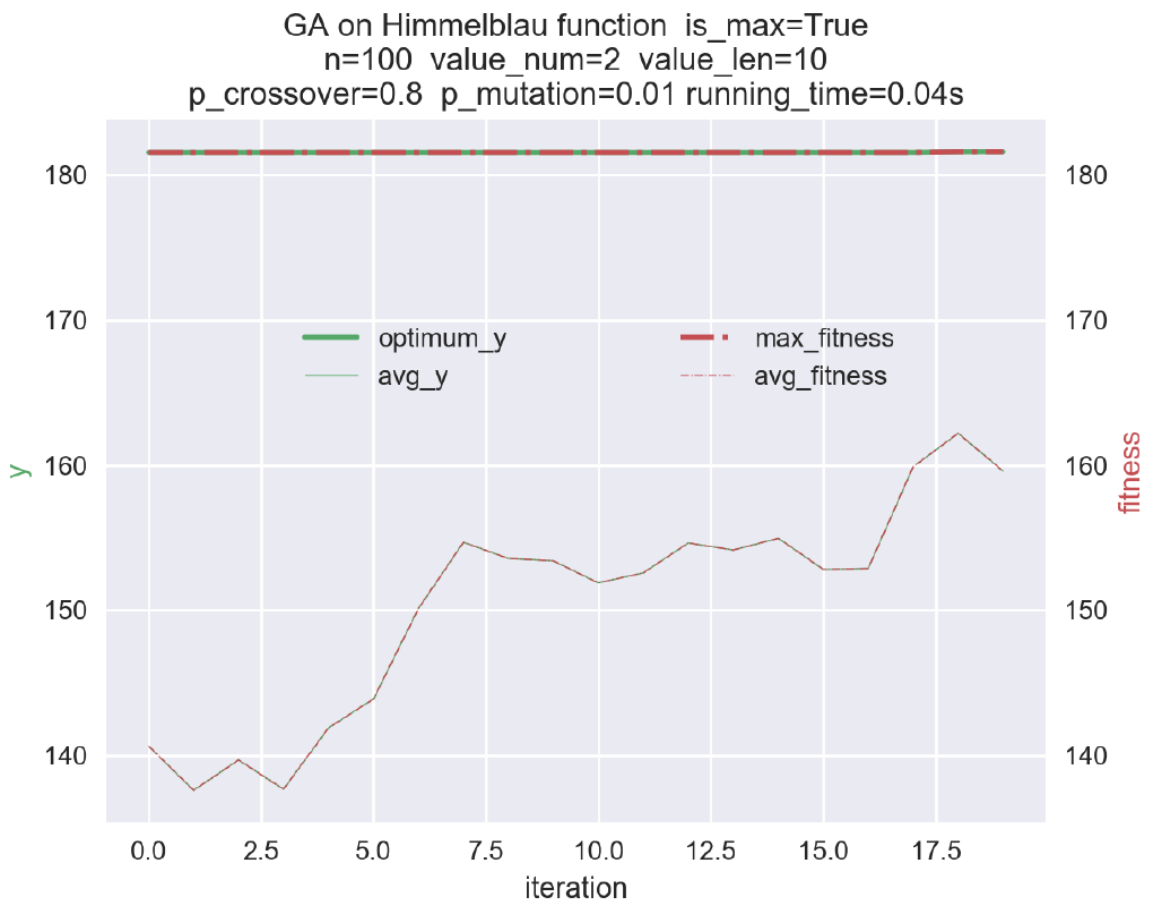
function 1/(1+y), fitness varies a lot when y is near to 0.



3. Himmelblau's function

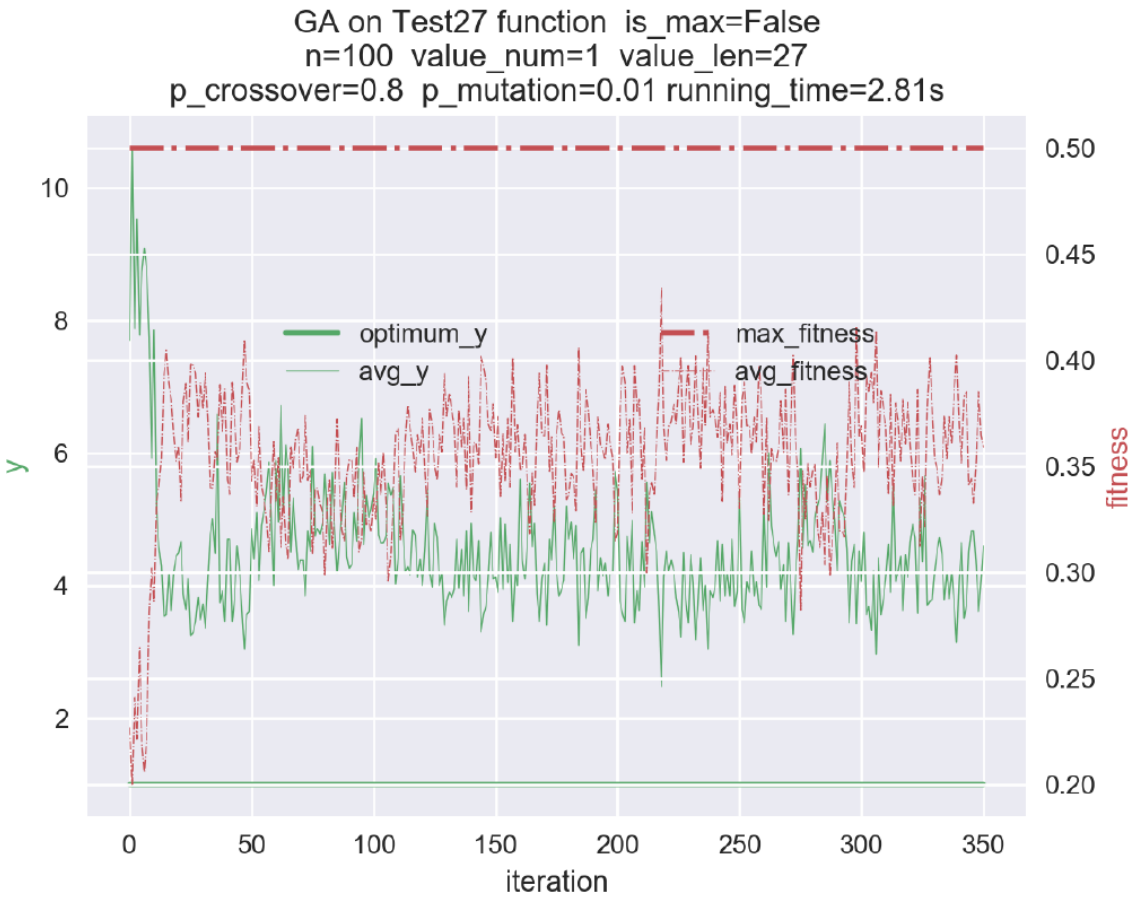
For finding minimum (figure on the top):
SGA converges at around iteration 13;
optimum_y decreases from 2 to 0.00025;
max_fitness increases from 0.4 to 0.9997.
For finding maximum (figure on the bottom):
SGA converges at iteration 1(kind of luck);
optimum_y is 181.57 ;
max_fitness is the same with optimum_y.


GA on Himmelblau function is_max=False
n=100 value_num=2 value_len=10
p_crossover=0.8 p_mutation=0.01 running_time=0.04s
optimum_y
avg_y
max_fitness
avg_fitness
y
fitness
iteration
0
10
20
30
40
50
0.2
0.4
0.6
0.8
1.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5



4. Test27 function

For finding minimum (figure on the top):
SGA converges at iteration 1(kind of luck);
optimum_y is 0
max_fitness is 0.5.
For finding maximum (figure on the bottom):
SGA converges at iteration 26;
optimum_y increases from 100 to 350 ;
max_fitness is the same with optimum_y.


GA on Test27 function is_max=False
n=100 value_num=1 value_len=27
p_crossover=0.8 p_mutation=0.01 running_time=2.81s
optimum_y
avg_y
max_fitness
avg_fitness
y
fitness
iteration
10
8
6
4
2
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0
50
100
150
200
250
300
350

Part 4: Source Code

```
import random
from random import randint
import matplotlib.pyplot as plt
import seaborn as sns
import math
import time
import re
from itertools import combinations

# key: objective function name
# value: [n, value_num, value_len, pc, pm, iteration, is_max]
param_dict = {
    'GramacyLee': [100, 1, 10, 0.5, 0.005, 100, False],
    'Beale': [100, 2, 10, 0.8, 0.01, 500, False],
    'GoldsteinPrice': [100, 2, 10, 0.8, 0.005, 100, False],
    'Himmelblau': [100, 2, 10, 0.8, 0.01, 20, False],
    'DeJong': [100, 10, 10, 1.0, 0.002, 1000, False],
```

```
    'Rosenbrock': [100, 7, 10, 1.0, 0.05, 1000, False],
    'Rastrigin': [100, 7, 10, 1.0, 0.02, 1000, False],
    'Test10': [100, 1, 10, 0.8, 0.01, 45, False],
    'Test11': [100, 1, 11, 0.8, 0.01, 55, False],
    'Test12': [100, 1, 12, 0.8, 0.01, 66, False],
    'Test13': [100, 1, 13, 0.8, 0.01, 78, False],
    'Test14': [100, 1, 14, 0.8, 0.01, 91, False],
    'Test15': [100, 1, 15, 0.8, 0.01, 105, False],
    'Test16': [100, 1, 16, 0.8, 0.01, 120, False],
    'Test17': [100, 1, 17, 0.8, 0.01, 136, False],
    'Test18': [100, 1, 18, 0.8, 0.01, 153, False],
    'Test19': [100, 1, 19, 0.8, 0.01, 171, False],
    'Test20': [100, 1, 20, 0.8, 0.01, 190, False],
    'Test21': [100, 1, 21, 0.8, 0.01, 210, False],
    'Test22': [100, 1, 22, 0.8, 0.01, 231, False],
    'Test23': [100, 1, 23, 0.8, 0.01, 253, False],
    'Test24': [100, 1, 24, 0.8, 0.01, 276, False],
    'Test25': [100, 1, 25, 0.8, 0.01, 300, False],
    'Test26': [100, 1, 26, 0.8, 0.01, 325, False],
    'Test27': [100, 1, 27, 0.8, 0.01, 351, False],
}


class Chromosome:
    def __init__(self, string):
        self.string = string
        self.x = self.get_x()
        self.y = self.get_y()
        self.fitness = self.get_fitness()
        self.p = 0

    # randomly generate input x of the objective function
    # each value takes value_len binary bits, thus can represent a integer among 0 -
2^value_len
    # scale the range 0 - 2^value_len to fit the given objective function's domain
    def get_x(self):
        if of == 'GramacyLee':
            # x         domain: 0.5 <= x <= 2.5, not starts from -0.5
            return (int(self.string, 2) + 256) / 512
```

```
        elif of == 'Beale':
            # x1 x2     domain: -4.5 <= x1, x2 <= 4.5
            return [(int(self.string[i:i + value_len], 2) - 512) / 114
                    for i in range(0, len(self.string), value_len)]
        elif of == 'GoldsteinPrice':
            # x1 x2     domain: -2 <= x1, x2 <= 2
            return [(int(self.string[i:i + value_len], 2) - 512) / 256 for i in range(0,
len(self.string), value_len)]
        elif of == 'Himmelblau':
            # x1 x2     domain: -4 <= x1, x2 <= 4
            return [(int(self.string[i:i + value_len], 2) - 512) / 128 for i in range(0,
len(self.string), value_len)]
        elif of in ['DeJong', 'Rastrigin']:
            # x[]       domain: -5.12 <= xi < 5.12
            return [(int(self.string[i:i + value_len], 2) - 512) / 100 for i in range(0,
len(self.string), value_len)]
        elif of in ['Rosenbrock']:
            # x[]       domain: -2.048 <= xi <= 2.048
            return [(int(self.string[i:i + value_len], 2) - 512) / 250 for i in range(0,
len(self.string), value_len)]
        elif re.match('Test*', of):
            return [-1 if elem == '0' else 1 for elem in self.string]

    # calculate output y of given objective function by input x
    def get_y(self):
        if of == 'GramacyLee':
            # subtract the minimum at the end so that the new minimum is 0
            return (self.x - 1) ** 4 + math.sin(10 * math.pi * self.x) / (2 * self.x) -
(-0.869011134989500)
        elif of == 'Beale':
            return (1.5 - self.x[0] + self.x[0] * self.x[1]) ** 2 \
                   + (2.25 - self.x[0] + self.x[0] * (self.x[1] ** 2)) ** 2 \
                   + (2.625 - self.x[0] + self.x[0] * (self.x[1] ** 3)) ** 2
        elif of == 'GoldsteinPrice':
            # subtract the minimum at the end so that the new minimum is 0
            return (1 + (self.x[0] + self.x[1] + 1) ** 2
                    * (19 - 14 * self.x[0] + 3 * self.x[0] ** 2
                       - 14 * self.x[1] + 6 * self.x[0] * self.x[1] + 3 * self.x[1] ** 2)) \
                   * (30 + (2 * self.x[0] - 3 * self.x[1]) ** 2
```

```
                * (18 - 32 * self.x[0] + 12 * self.x[0] ** 2 + 48 * self.x[1] - 36 * self.x[0] *
self.x[1] + 27 *
                   self.x[1] ** 2)) \
               - 3
        elif of == 'Himmelblau':
            return (self.x[0] ** 2 + self.x[1] - 11) ** 2 + (self.x[0] + self.x[1] ** 2 - 7) ** 2
        elif of == 'DeJong':
            return sum(elem ** 2 for elem in self.x)
        elif of == 'Rosenbrock':
            return sum(100 * ((self.x[i + 1] - (self.x[i] ** 2)) ** 2) + (1 - self.x[i]) ** 2
                       for i in range(1, len(self.x) - 1))
        elif of == 'Rastrigin':
            return 10 * len(self.x) + sum(elem ** 2 - 10 * math.cos(2 * math.pi * elem) for
elem in self.x)
        elif re.match('Test*', of):
            return abs(sum(elem[0] * elem[1] for elem in list(combinations(self.x, 2))))

    # generate fitness that we want to maximize
    # fitness == y if the aim is to maximize the objective function
    # fitness == 1 / (1 + y) if the aim is to minimize the objective function
    def get_fitness(self):
        if is_max:
            return self.y
        else:
            return 1 / (1 + self.y)

    def calc_p(self, sum_fitness):
        self.p = self.fitness / sum_fitness

    # customize printing format, called by print()
    def __repr__(self):
        return '%r\tx:%r\ty:%r\tfitness:%r\n' % (self.string, self.x, self.y, self.fitness)


# randomly generate a population of size n
def generate_population():
    new_population = list()
    for _ in range(n):
        str_list = [str(randint(0, 1)) for _ in range(value_num * value_len)]
```

```
            chromosome = Chromosome(''.join(str_list))
            new_population.append(chromosome)
        # print('population:\n%r\n' % new_population)
        return new_population


def reproduction():
    fitness_list = list()
    for i in range(n):
        fitness_list.append(population[i].fitness)

    # find two chromosomes with the largest fitness and put them at the beginning
    elite1_fitness = max(fitness_list)
    elite1_index = fitness_list.index(elite1_fitness)
    fitness_list.pop(elite1_index)
    elite2_fitness = max(fitness_list)
    elite2_index = fitness_list.index(elite2_fitness)

    elite1 = population[elite1_index]
    elite2 = population[elite2_index]
    population[elite1_index] = population[0]
    population[elite2_index] = population[1]
    population[0] = elite1
    population[1] = elite2

    # calculate p for non-elites chromosome
    sum_fitness = 0
    for i in range(2, n):
        sum_fitness += population[i].fitness
    for i in range(2, n):
        population[i].calc_p(sum_fitness)

    # use roulette to select string
    # roulette contains n-2 values
    p = 0
    roulette = list()
    for i in range(2, n):
        p += population[i].p
        roulette.append(p)
```

```
    # print('roulette:\n%r\n' % roulette)

    new_population = list()
    # 2 elites reproduce directly, do not go through roulette
    new_population.append(population[0])
    new_population.append(population[1])
    # n-2 left chromosomes go through roulette
    for i in range(n - 2):
        p = random.random()
        for j in range(n - 2):
            if p <= roulette[j]:
                new_population.append(population[j + 2])
                break
    # print('after reproduction:\n%r\n' % new_population)
    return new_population


def crossover():
    new_population = list()
    # 2 elites pass directly to next generation, do not go through crossover
    new_population.append(population[0])
    new_population.append(population[1])
    # n-2 left chromosomes go through crossover at probability pc
    for i in range(1, n // 2):
        father = population[i * 2]
        mother = population[i * 2 + 1]
        brother = father
        sister = mother
        for j in range(value_num):
            # has a probability of pc to do crossover
            if random.random() <= pc:
                pos = j * value_len + randint(1, value_len - 1)
                brother = Chromosome(father.string[:pos] + mother.string[pos:])
                sister = Chromosome(father.string[:pos] + mother.string[pos:])
                father = brother
                mother = sister
        new_population.append(brother)
        new_population.append(sister)
    # print('\nafter crossover:\n%r\n' % new_population)
```

```
    return new_population


def mutation():
    # n-2 left chromosomes go through mutation at probability pm
    for i in range(2, n):
        for j in range(value_num):
            for k in range(value_len):
                # has a probability of p to do mutation
                if random.random() <= pm:
                    string = population[i].string
                    flip = lambda x: '1' if x is '0' else '0'
                    char = string[j * value_len + k]
                    char = flip(char)
                    new_string = string[:j * value_len + k] + char + string[j * value_len + k + 1:]
                    # print('replace %r with %r' % (string, new_string))
                    population[i] = Chromosome(new_string)
    # print('\nafter mutation:\n%r\n' % population)
    return population


def get_record():
    index = -1
    max_fitness = -1
    sum_y = 0
    sum_fitness = 0
    for i in range(n):
        sum_y += population[i].y
        sum_fitness += population[i].fitness
        if population[i].fitness > max_fitness:
            max_fitness = population[i].fitness
            index = i
    return population[index].y, sum_y / n, population[index].fitness, sum_fitness / n


def draw_plot(data):
    sns.set()
    plt.title('GA on %s function  is_max=%r\nn=%r  value_num=%r  value_len=%r\n'
              'p_crossover=%r  p_mutation=%r running_time=%0.2fs'
```

```
            % (of, is_max, n, value_num, value_len, pc, pm, running_time))
    plt.plot(data['optimum_y'], color='C1', linestyle='-', linewidth=2, label='optimum_y')
    plt.plot(data['avg_y'], color='C1', linestyle='-', linewidth=0.5, label='avg_y')
    plt.xlabel('iteration')
    plt.ylabel('y', color='C1')
    plt.legend(loc=(.2, .6), frameon=False)
    right_axis = plt.twinx()
    right_axis.plot(data['max_fitness'], color='C2', alpha=1, linestyle='-.', linewidth=2,
label='max_fitness')
    right_axis.plot(data['avg_fitness'], color='C2', alpha=1, linestyle='-.', linewidth=0.5,
label='avg_fitness')
    right_axis.set_ylabel('fitness', color='C2')
    right_axis.legend(loc=(.6, .6), frameon=False)
    plt.show()


if __name__ == '__main__':
    start_time = time.time()
    # choose an objective function from
    # 'GramacyLee' 'Beale' 'GoldsteinPrice' 'Himmelblau' 'DeJong' 'Rosenbrock'
'Rastrigin' or 'Test10' to 'Test27'
    of = 'DeJong'
    n = param_dict[of][0]
    value_num = param_dict[of][1]
    value_len = param_dict[of][2]
    pc = param_dict[of][3]
    pm = param_dict[of][4]
    iteration = param_dict[of][5]
    is_max = param_dict[of][6]

    population = generate_population()

    keys = {'optimum_y', 'avg_y', 'max_fitness', 'avg_fitness'}
    record_dict = dict([(key, []) for key in keys])
    iterate = 0
    while iterate < iteration:
        population = reproduction()
        population = crossover()
        population = mutation()
```

```
        optimum_y, avg_y, max_fitness, avg_fitness = get_record()
        record_dict['optimum_y'].append(optimum_y)
        record_dict['avg_y'].append(avg_y)
        record_dict['max_fitness'].append(max_fitness)
        record_dict['avg_fitness'].append(avg_fitness)
        iterate += 1
    print('after genetic algoritm:\n%r\n' % population)
    running_time = time.time() - start_time
    draw_plot(record_dict)
```

Program Installations

Seaborn: https://seaborn.pydata.org/installing.html
MathPlotLib: https://matplotlib.org/3.1.1/users/installing.html