Simple Genetic Algorithm

CP468 Artificial Intelligence · Group 4 · Term Project

Tianran Wang 174178160

Yiyi Chen 174177150

Mingyue Li 174177600

Zhiyu Zhang 174178210

5th Dec 2017

Part 1: Algorithm Design

Our group uses Python3 to implement a simple genetic algorithm (SGA) for the convenience of built-in random function and extended data visualization tools like matplotlib and seaborn.

1. Input value

We use binary code to store input values of the given objective function (OF). Each input value is 10-bit long thus can represent one integer from 0 to 1023. We denote it as *value_len*=10. The range of input value needs to be narrowed according to OF's input domain. Take the 2-dimensional benchmark function Himmelblau's function as an example: this OF has two input values x1 and x2 (*value_num*=2), thus need a string of 20 in length to store all the binary codes. What's more, since we usually evaluate this OF on domain [-4, 4] for both x1 and x2, we need to transform the original range ([(0-512)/128, (1024-512)/128]) to get an approximate input domain for this OF.

However, for testing OFs given by professor, *value_len* ranges from 10 to 27 and *value_num* is fixed to 1, which means that one chromosome just contains one value of *value_len* bits.

2. Structure of chromosome

We define a class *Chromosome* to store information of one chromosome and its values of the given objective function:

```
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]
```

```
    . . .

# calculate output y of given objective function by input x
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)
```

string is used to store encoded binary codes of input values, randomly generated by python built-in function. *x* is a list which contains decoded and scaled input values and *y* is the output value of OF. Need to mention that all of the test OFs given by professor can be written in the same format, thus we use Python built-in function *itertools.combinations()* to get all pairs of elements in list *x*. Here we introduce *fitness* which is whether positively or negatively correlated with *y* depending on the boolean parameter *is_max*. If we want to get maximum of one OF, we can set *is_max*=True or vice versa. However, we always need to maximize *fitness* regardless of whether to maximize or minimize *y* for different OFs.

3. Roulette in reproduction

Among three normal steps in SGA, the first one is reproduction where a portion of the existing population is selected to breed a new generation. Roulette is one of the most-used fitness-based selection methods. We first get each chromosome's proportion of fitness *p* by doing *fitness/sum_fitness* where *sum_fitness* is the sum of all chromosomes' fitness in the population and put them in a list (for example: *[0.1, 0.4, 0.3, 0.2]*). Then we accumulate one proportion in each step to generate a new list in which each element is the upper bound of corresponding chromosome's fitness proportion(for example: *[0.1, 0.5, 0.8, 1.0]*). Later we stimulate spinning process by using function *random.random()* to generate a number between 0 and 1. The section the new number fall in decides the chromosome be selected.

4. Details in crossover & mutation

Crossover and mutation are used for generating the next generation of the population. We skip talking about the simple process in these two steps but focus on one idea: treating encoded binary codes *string* as a combination of *j* separated codes while *j* is the dimension of given OF. We still use Himmelblau's function as an example: each chromosome has a string of 20 in length since it is a 2-dimensional function and each input value consists of 10 bits. Suppose there are two strings:

1 0 0 1 0 0 1 0 0 1 0 0 1 0 1 1 1 0 0 1

0 0 1 1 0 1 1 0 0 0 0 1 0 1 0 1 0 0 1 1

In a normal case, we will generate a *k* randomly between 1 and 19 as a separation for crossover. Suppose that *k* is 5, we will get new strings after one crossover:

1 0 0 1 0 1 1 0 0 0 0 1 0 1 0 1 0 0 1 1

0 0 1 1 0 0 1 0 0 1 0 0 1 0 1 1 1 0 0 1

However, each of the string consists of *j* values. If we split them we will find that just one in *j* is changed during the crossover part:

1 0 0 1 0 1 1 0 0 0 0 1 0 1 0 1 0 0 1 1

0 0 1 1 0 0 1 0 0 1 0 0 1 0 1 1 1 0 0 1

The same thing happens in the mutation step: actually, only 1 over *j* part of values go through mutations. During the test of our program, we find that as j increasing to 7, functions of crossover and mutation work little and the SGA always fails to find a global maximum of fitness no matter what crossover probability and mutation probability we choose. The reason is simple: there are at most 1/7 part of values actually changed during each crossover and mutation, resulting in an unaffordable slowdown of searching speed to find the optimal solution. We notice this problem and fix it by applying crossover and mutation on *j* values in one string, thus getting a remarkable improvement of the program.

5. Elitist strategy

Mathematicians prove that SGA does not converge in a probabilistic sense. One way to make it converge towards the optimal solution is using an elitist strategy which means that put the elite (one chromosome with the largest fitness in the population) directly into the next generation without going through crossover and

mutation. This strategy may sound crucial in sociology terms but it can increase the computing speed on a large scale. What's more, since current the best solution is always sent to the next generation, the maximum of fitness will never decrease during the whole iteration.

In our program, we find two elites (ensure that remaining number of chromosomes is even) at the beginning of reproduce step and put them at the first two positions of the whole population's list, then we skip these two elites 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

# iterate in next generation, find two elites at the beginning
```

Part 2: Running & Parameter Controls

There are eight parameters in our program and we tested the SGA with seven benchmark OFs found online and all test OFs given by professor. Since different OFs need different params to get the optimal solution, we encapsulate these params 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 is implemented in Python 3 with two extended data visualization packages: matplotlib and seaborn.
Running and testing this program is very simple, you just need to assign the name of one OF at the beginning of the program's entrance. The program will use default parameters listed in *param_dict*. Change them within 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 size of a population which remains constant during iteration. The size of a population usually determines the density of gene schemata. The normal range of n is from 0 to 100. A very large *n* may bring a burden on computation without finding a better solution. In our program's default configuration, we set n=100 for all OFs.

value_len and *value_num*, which have been talked in the previous part, varies from different OFs. Since in our program, we fix the length of one encoded value to 10 in those seven OFs found online, *value_len* is 10 times the dimension of each OF. For three n-dimensional OFs, we set *value_num* as 10 for DeJong's function and 7 for

other two. While for those test OFs given by professor, *value_num* is fixed to 1 and *value_len* is the same as the number of variables.

Performance of SGA optimization problem mainly depends on the balance of the depth and breadth of search space (the entire range of possible solutions) where crossover probability (*pc*) and mutation probability (*pm*) are key factors among them. In theory, the higher the *pc* is, the quicker SGA could converge to the optimal region, while at the same time is the higher probability those highly adaptive schemata could be removed. Elitist strategy deals with this problem well. On the other hand, if *pc* is too small, the searching process may slow down or the algorithm may not even converge. Normally we pick *pc* among [0.5, 0.9]. The situation is similar for mutation probability: a very small *pm* may not be able to generate new schemata, thus lead to genetic drift and premature convergence while the large one may turn SGA into a randomly searching algorithm. Though there is no concrete guidance for choosing the optimal *pm*, the recommended range is [0.001, 0.05].

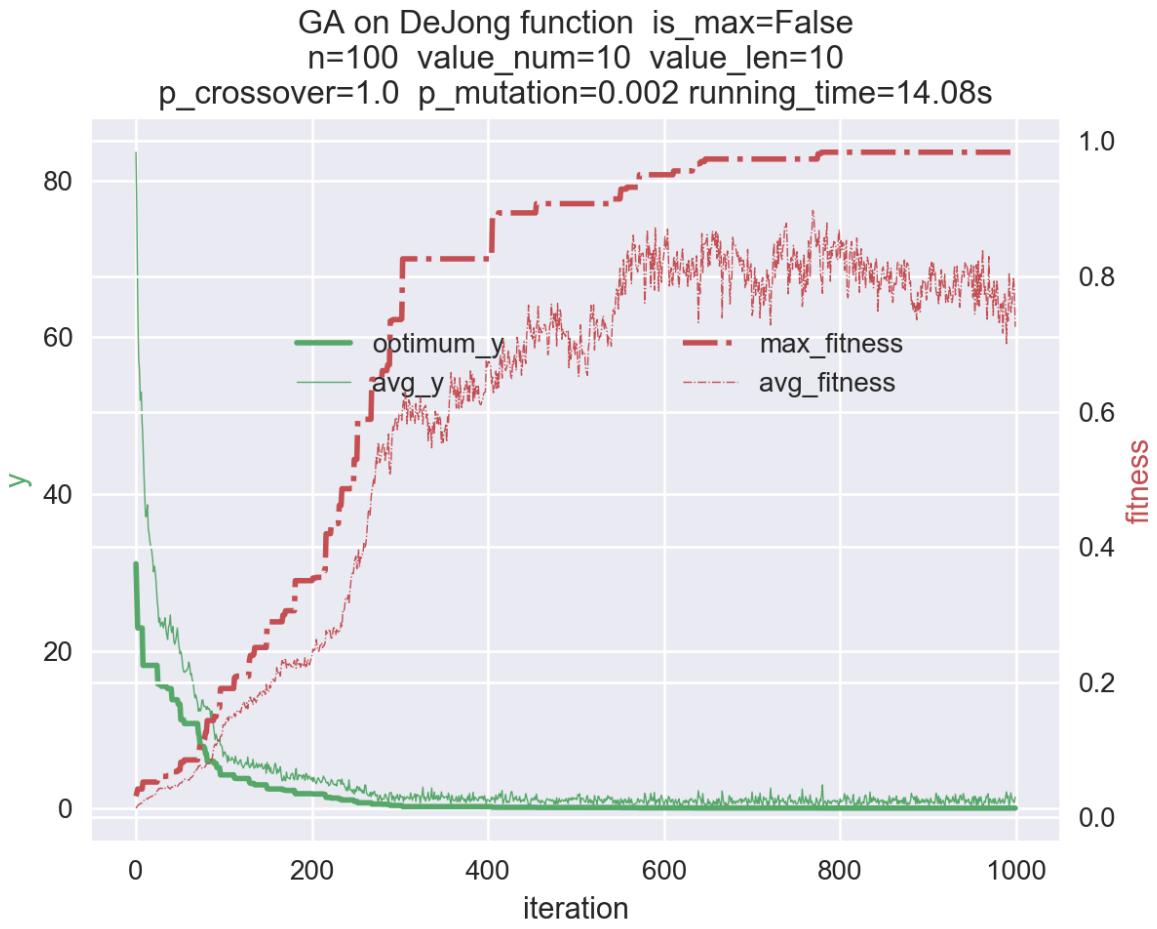
iteration is easy to determine compared to *pc* and *pm* because if the curve of fitness is still going up at the end of the iteration, it means that SGA does not get convergence yet. Moreover, we already know that the global minimum of *y* is 0 and the global maximum of *fitness* is 1 for all six OFs in our programs. If the result is still far from the target value, we can acknowledge that iteration should continue. For SGA with a not very complex OF, the iteration is usually around several hundred or even less.

Part 3: Results on OFs

Results of SGA on all OFs can be found on folder *Results*. Each result contains a figure showing how *y* and *fitness* change during iteration, and one text file showing the population at the end of iteration. The first two lines of the text file shows the optimum of *y* and the maximum of *fitness*. Here we only show four of all OFs.

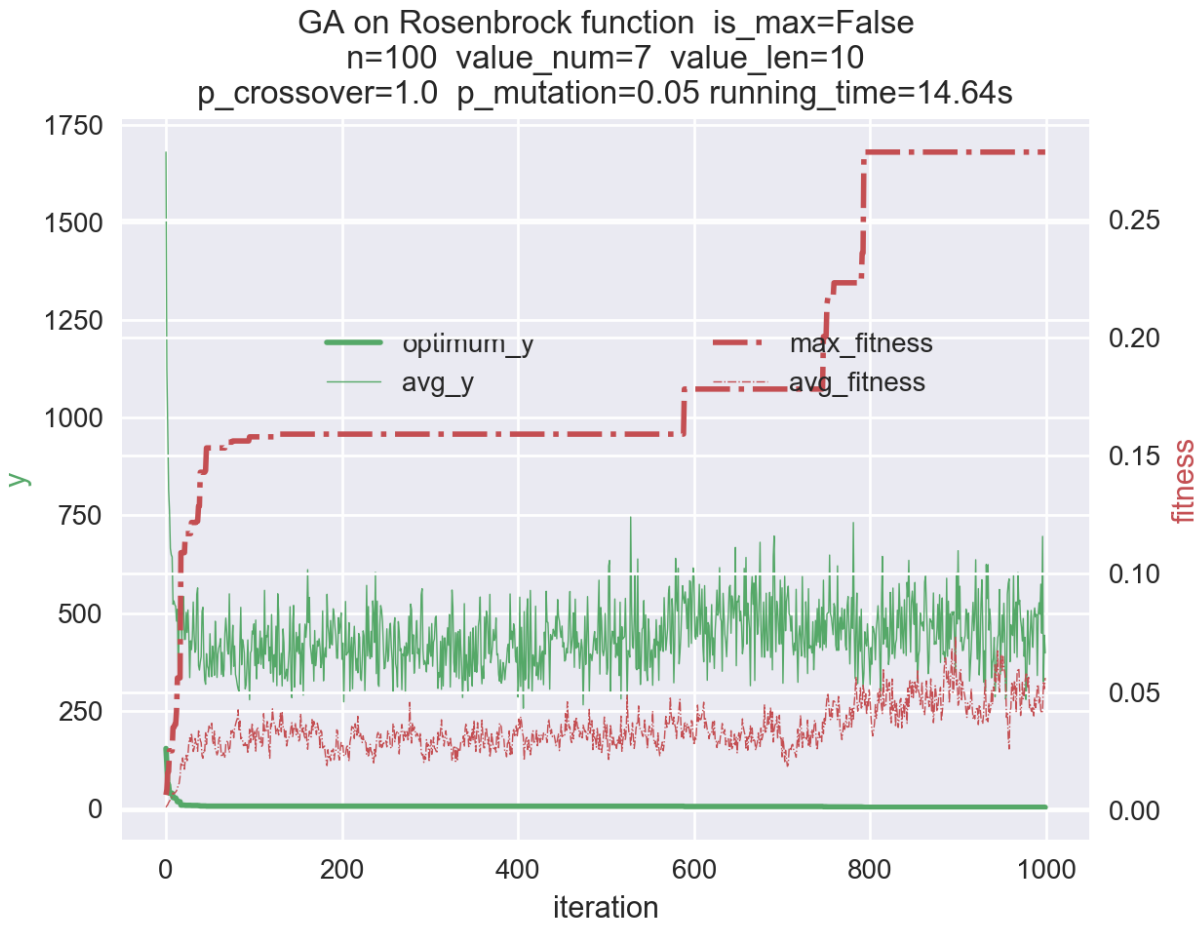
1. DeJong function

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

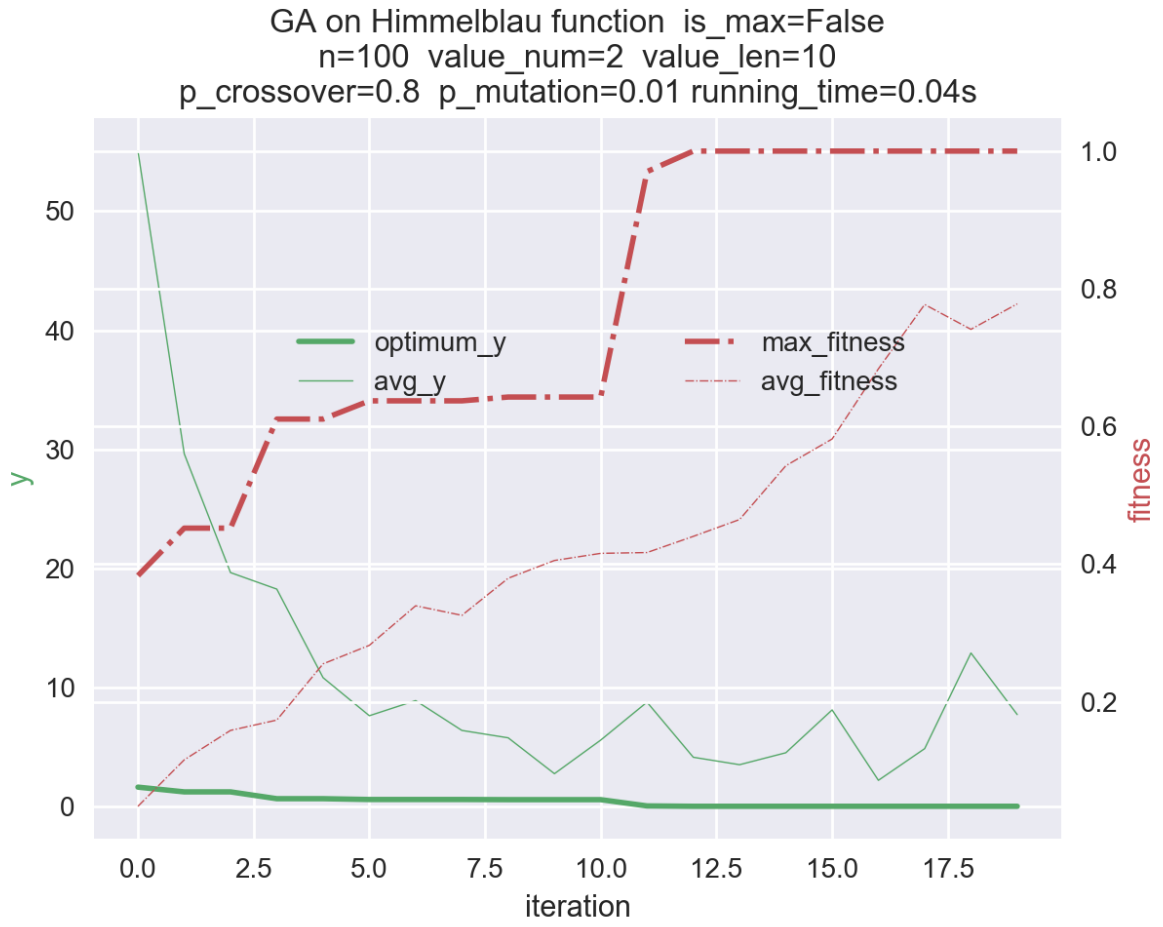
SGA converges at around iteration 800;
optimum_y decreases from 150 to 2.59;
max_fitness increases from 0.01 to 0.28, which is far from 1 in that due to the fitness function *1/(1+y)*, *fitness* varies a lot when y is near to 0.



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*.

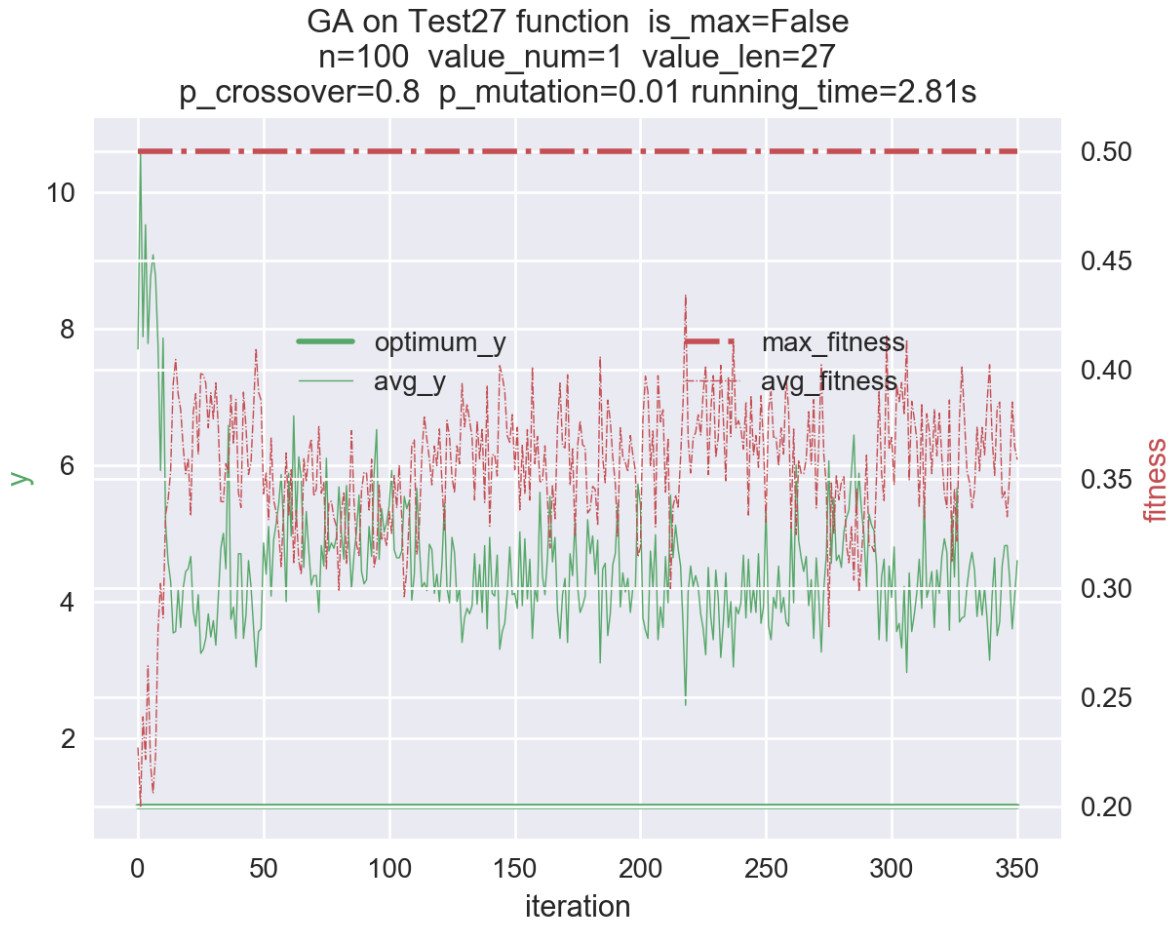

GA on Himmelblau function is_max=False
n=100 value_num=2 value_len=10
p_crossover=0.8 p_mutation=0.01 running_time=0.04s
optimum_y
avg_y
max_fitness
avg_fitness
y
fitness
iteration
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
10
20
30
40
50
0.2
0.4
0.6
0.8
1.0

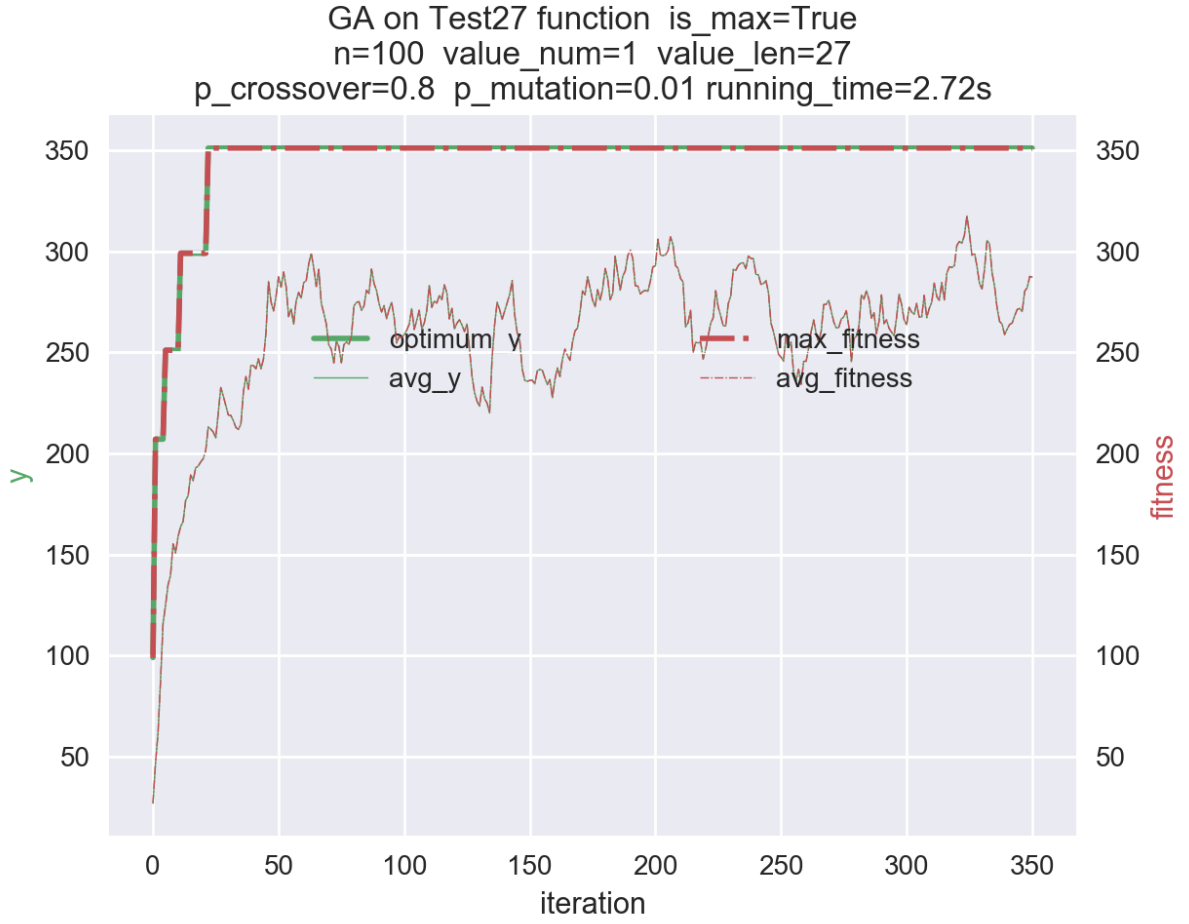

GA on Himmelblau function is_max=True
n=100 value_num=2 value_len=10
p_crossover=0.8 p_mutation=0.01 running_time=0.04s
optimum_y
avg_y
max_fitness
avg_fitness
y
fitness
iteration
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
140
150
160
170
180

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


GA on Test27 function is_max=True
n=100 value_num=1 value_len=27
p_crossover=0.8 p_mutation=0.01 running_time=2.72s
optimum_y
avg_y
max_fitness
avg_fitness
y
fitness
iteration