# Part1 Report

# Genetic Algorithm Implementation

## Overview

The `myGA` class in `ga_lib.py` provides a comprehensive implementation of a genetic algorithm for solving optimization problems. It includes methods for population initialization, selection, crossover, mutation, and evolution over multiple generations.

## Class: `myGA`

### Methods

**`__init__(self, problem, n_max_iterations=100, minimize=True)`**

Initializes the genetic algorithm with the given optimization problem, maximum number of iterations, and whether to minimize the objective function.

#### Sampling Methods

- `permutation_sampling(self, n_samples)`: Performs permutation random sampling to generate initial population individuals.

  This method generates individuals with random permutations of values within the specified range for each parameter of the optimization problem.

- `float_sampling(self, n_samples)`: Performs float random sampling to create initial population individuals.

  This method creates individuals with random float values within the specified range for each parameter of the optimization problem.

- `binary_sampling(self, n_samples)`: Performs binary random sampling to produce initial population individuals.

  This method generates individuals with random binary strings representing the parameters of the optimization problem.

#### Selection Methods

- `tournament_selection(self, pop, n_selects, n_parents=2, tournament_size=2)`: Selects individuals using tournament selection based on their fitness.

  Tournament selection is used to select the fittest individuals from the population to act as parents for producing offspring in the next generation.

**Crossover Methods**

- `uniform_crossover(self, X, crossover_rate=0.7)`: Performs uniform crossover to create offspring from selected parent individuals.

  Uniform crossover swaps genes between pairs of parents at randomly chosen positions based on a specified crossover rate.

- `order_crossover(self, X, crossover_rate=0.7, shift=False)`: Applies order crossover to generate offspring from selected parent individuals.

  Order crossover recombines genes of parent individuals based on specific rules, controlled by the crossover rate and shift flag.

- `point_crossover(self, X, crossover_rate=0.7, n_points=2)`: Utilizes point crossover to create offspring from selected parent individuals.

  Point crossover combines genetic information from parent individuals at specific crossover points, with the number of points determined by the parameter n_points.

**Mutation Methods**

- `gaussian_mutation(self, pop, mutation_rate=0.5, sigma=0.2)`: Applies Gaussian mutation to introduce random noise to the gene values of individuals in the population.

  Gaussian mutation perturbs gene values by adding random values drawn from a Gaussian distribution with a specified standard deviation (sigma).

- `inversion_mutation(self, pop, mutation_rate=0.5)`: Utilizes inversion mutation to reverse a segment of genes within individuals in the population.

  Inversion mutation reverses the order of a subset of genes within an individual's chromosome with a certain probability.

- `bit_flip_mutation(self, pop, mutation_rate=0.5)`: Performs bit flip mutation to flip the bits of selected genes within individuals in the population.

  Bit flip mutation flips the bits of randomly selected genes within an individual's chromosome with a specified probability.

**Other Utility Methods**

- `eliminate_duplicates(self, pop, epsilon=1e-16)`: Eliminates duplicate individuals in the population based on a specified threshold value.

  This method removes duplicate individuals from the population to maintain diversity and avoid redundant evaluations.

- `survival(self, pop, n_survival)`: Selects the top individuals to survive and continue to the next generation based on their constraint values and fitness.

The survival method selects the less constraint values or the fittest individuals from the population to form the next generation, ensuring that the best individuals are retained, where other worse individuals are going to death.

- `generate_offsprings(self, pop, num_offsprings, selection_method, crossover_method, crossover_rate, mutation_method, mutation_rate)`: Generates offspring individuals by performing selection, crossover, mutation, and eliminating duplicates iteratively.

  This method orchestrates the process of generating offspring individuals from the current population using the specified selection, crossover, and mutation methods, while ensuring diversity through eliminating duplicates.

- `initialize_population(self, pop_size, sampling_method)`: Initializes the population by creating individuals through random sampling using the specified method.

  The initialize_population method creates the initial population of individuals by performing random sampling according to the specified method (e.g., permutation, float, binary).

**Evolution Method**

- `evolve(self, n_pop, n_gen, n_off, sampling_method, crossover_method, crossover_rate, mutation_method, mutation_rate, selection_method="tournament_selection")`: Evolves the population over a specified number of generations by generating offspring and selecting the fittest individuals to form the next generation.

  The evolve method coordinates the evolutionary process by repeatedly applying selection, crossover, mutation, and survival strategies to generate and retain the fittest individuals in the population over multiple generations.

---

## Code Document:

### 1.Problem1

The problem 1 aims to find the best solution for the given target ASCII art matrix. The fitness function is designed to measure the similarity between the target and the output ASCII art matrix. The higher the fitness score, the closer the output ASCII art matrix is to the target.

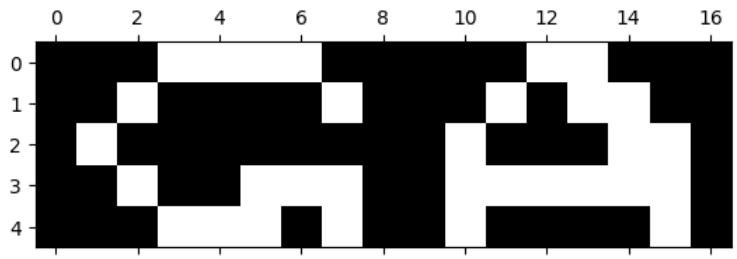
The genetic algorithm is used to evolve a population of potential solutions, where each solution is represented by a vector with the length equal to the multiplication between row and column of target ASCII art matrix. The calculation of fitness is shown below:

```
score = self.sign*np.bitwise_not(np.bitwise_xor(x, self.target)).sum()
```

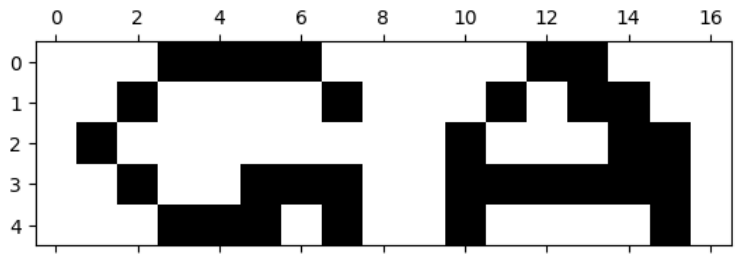
where `x` is the output ASCII art matrix, `self.target` is the target ASCII art matrix, and `xor` operation is used to calculate the number of different bits between the target and the output ASCII art matrix. By then using bitwise `not` operation and `sum` operation, we can obtain the total number of same bits between the target and the output ASCII art matrix. Besides, `self.sign` is a variable for user to choose whether maximizing the similarity or minimizing the similarity between the target and the output ASCII art matrix. Here we use the default value of `self.sign = -1` since our GA is to minimize the fitness score.

The fitness function is then applied to evaluate the fitness of each solution, selecting the fittest individuals for the next generation based on their fitness scores. The process continues until a solution with a high fitness score is found or a maximum number of generations is reached.

The figures shown below are the maximization and minimization solutions of the ASCII art matrix generated by the GA.




Maximization solution result




Minimization solution result

It can be observed that the maximization solution show the same bit matrix to the target ASCII art matrix, while the minimization solution show the filpped bit matrix from the target ASCII art.

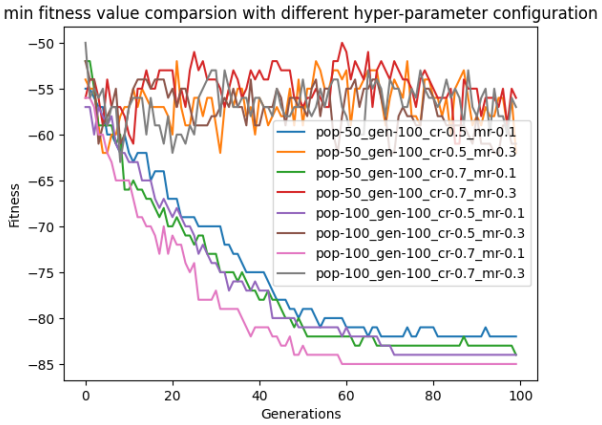
**2.Problem 2**

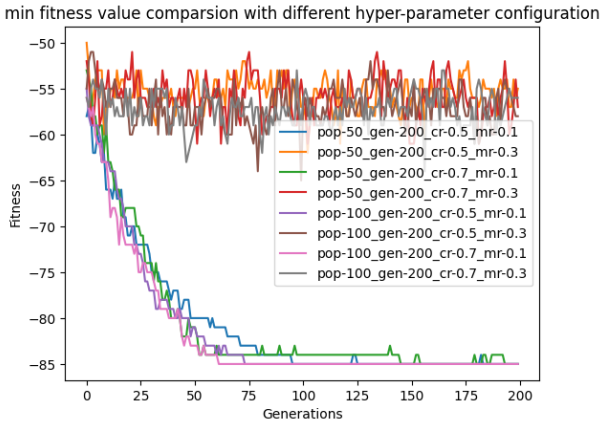
The evolution of the population is guided by a set of hyper-parameters, including population size, mutation rate, and the number of generations. By tuning these hyper-parameters, we can control the trade-off between exploration and exploitation, and achieve better performance in the target problem.

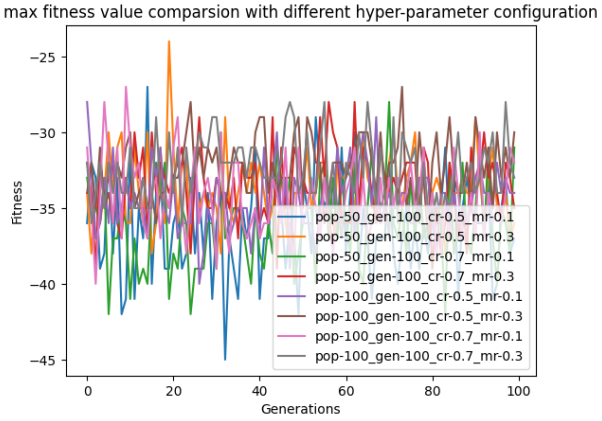
The default hyper-parameters I choose for a GA are:

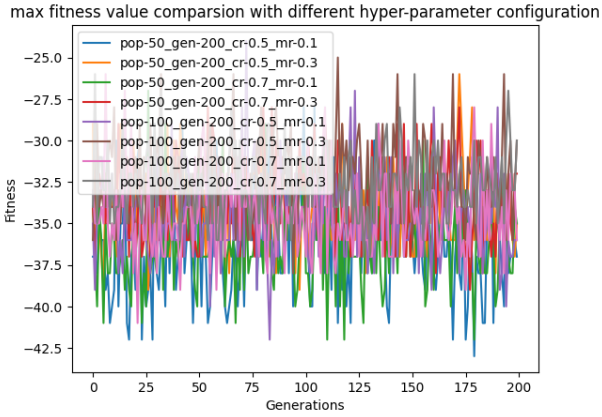
- Population size: 200
- Mutation rate: 1/85
- Crossover rate: 0.7
- Tournament size: 2
- Number of generations: 200

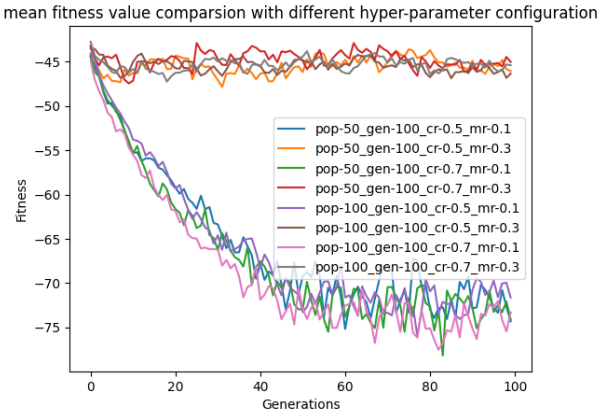
To observe the behaviour of the GA with different hyper-parameters, I picked serveral parameters configurations and plot the result figure below:

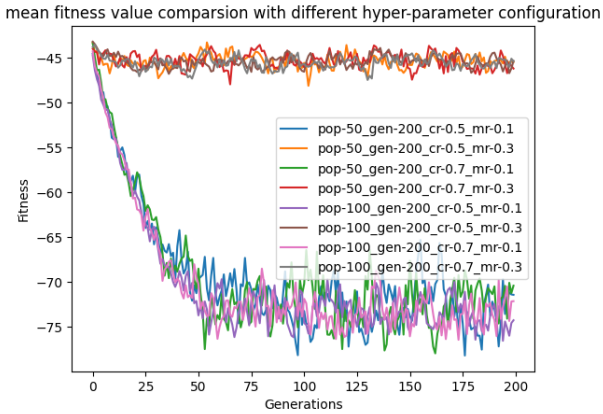


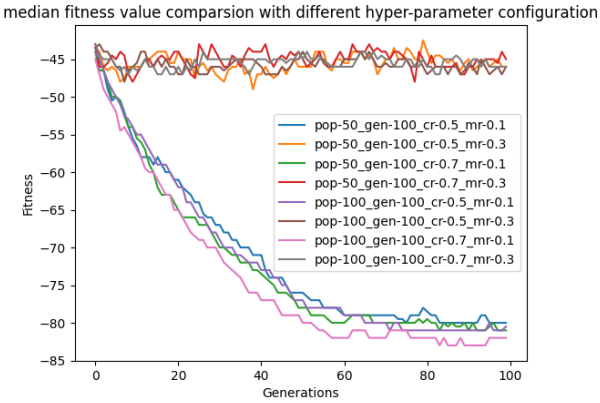



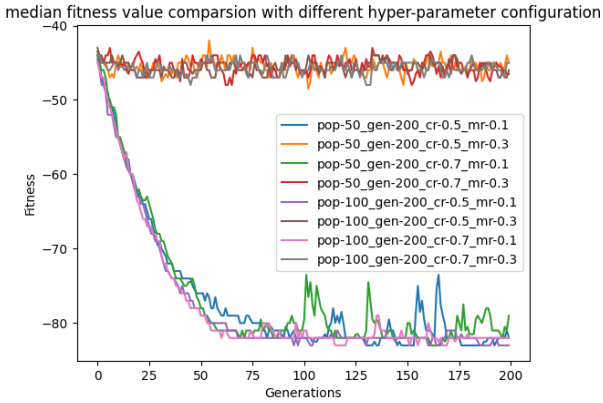



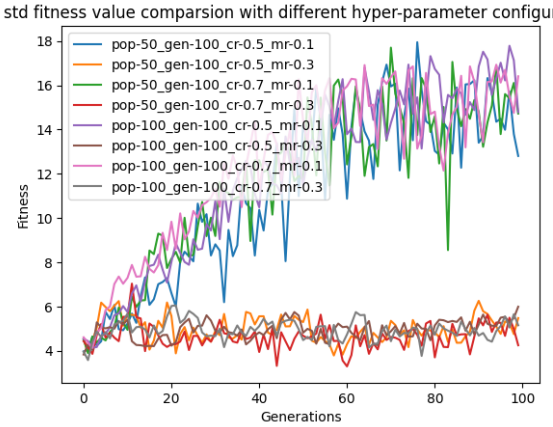



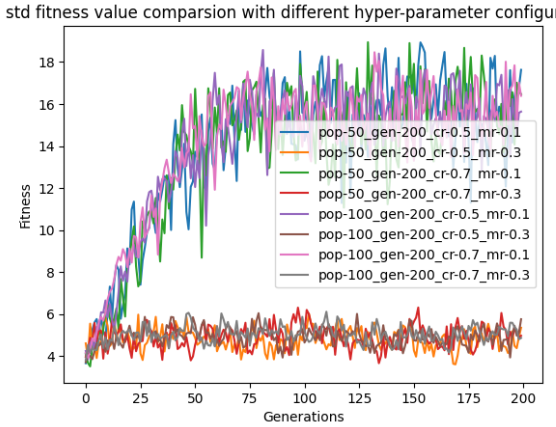



In the figures above, the legends represent different hyper-parameters configurations. The x-axis represents the number of generations. The y-axis represents the fitness value. There are five figures which represent the minimum, maximum, mean, median, and standard deviation of the fitness values.

It is clear to see that configuration with population size 100, crossover rate 0.7, and mutation rate 0.1 has the best performance. The successful reason I think is that it has more population that with more divergence and crossover rate is high enough that the population can inherit their parents' advantage gene. It is worth noting that high mutation rate will cause the population to be more diverse and the convergence rate will be slower.

I also tried to extend generation times to 200. Sadly, some parameters configure still could not convergence.

Furthermore, it can be predicted that using this identified parameter combination may not be universally effective, as different problems may have varying sizes of search space, leading to potential limitations in applicability.

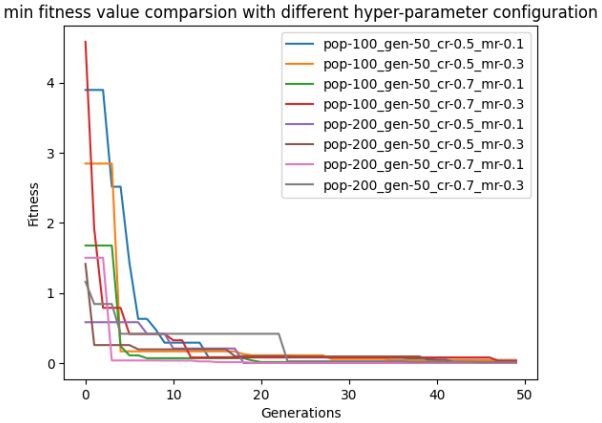
### 3.Problem 3

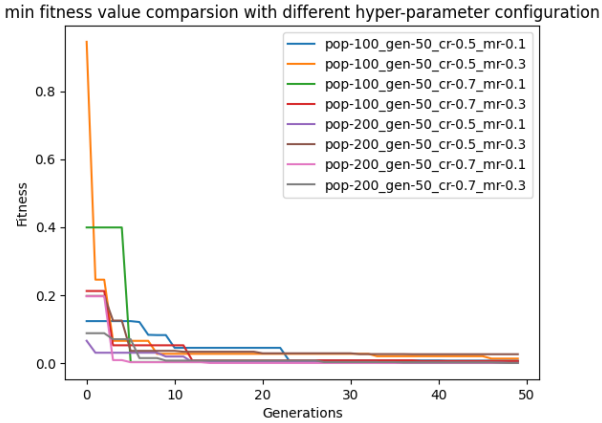
For the Problem 3, I choose to optimize `Booth function` as the single-objective problem and choose `Rosenbrock function` that constrained to a disk as the constrained optimisation problem.

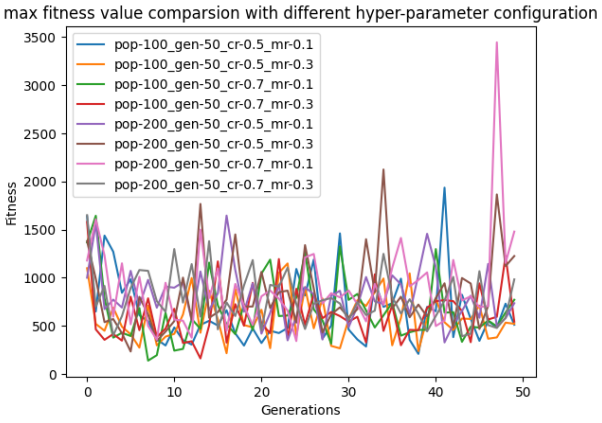
For both of them, I use realed value of x and y to represent the solution. The fitness is calculated by use `(x[0]+2*x[1]-7)**2+(2*x[0]+x[1]-5)**2` and `(1-x[0])**2+100*(x[1]-x[0]**2)**2` function respectively, where `x` is an array of `x` and `y`.

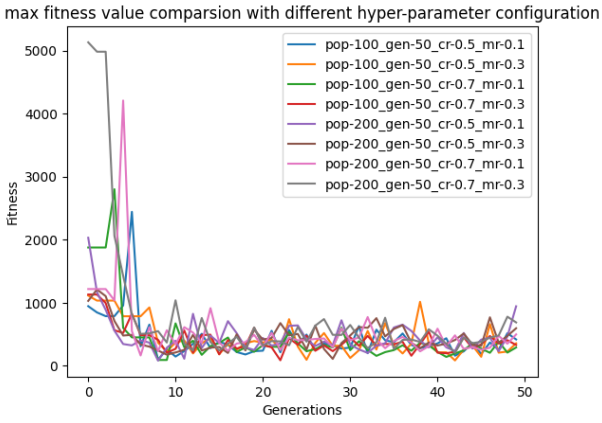
For this Problem, I use `float sampling` to generate the initialized population, since here is real-value. Then I use `uniform_crossover` to do uniform exchange between the parent and the offspring. And I use `gaussian_mutation` to do mutation for each individual of offsprings, that the mutated point is going to change depends on the sigma value. Here the sigma value is 0.2. Most importantly, since those two functions have bounds, I carefully set the bounds of the individuals that make gene keep in the range of the bounds. Moreover, for inequality constraints, I calculated the value beyond constraint, that make those as the selection element, tending to minimize them less and equal to zero.

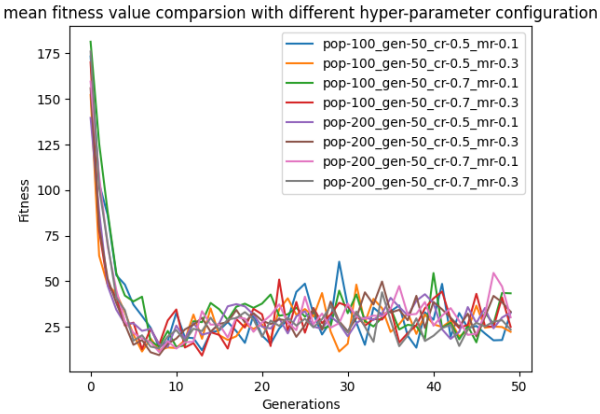
I tried with different parameter configurations, and the statistic figures are shown below

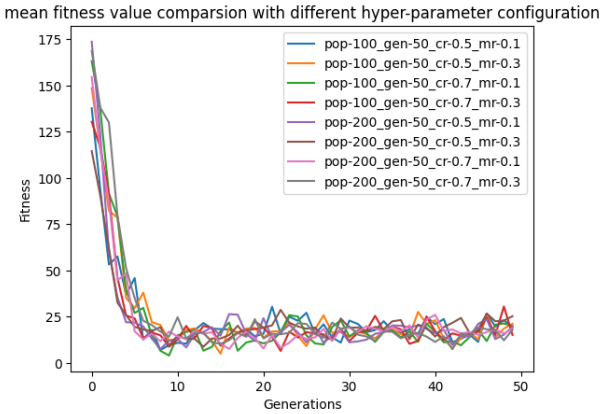


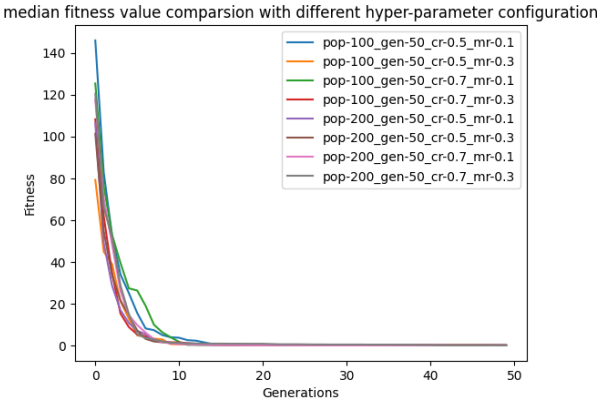



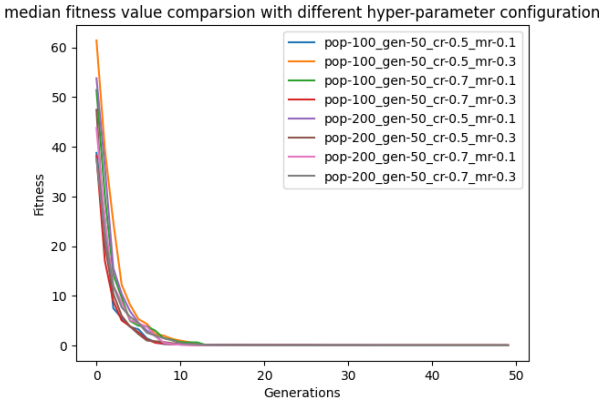



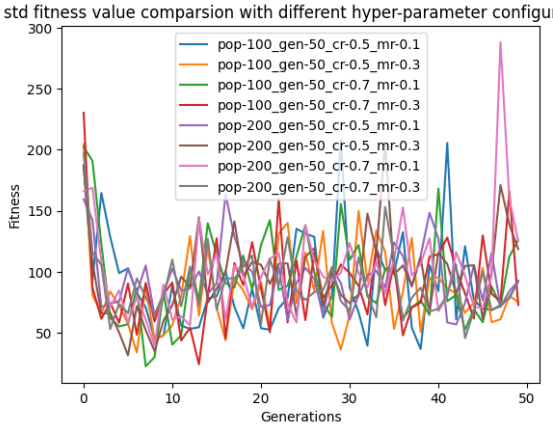



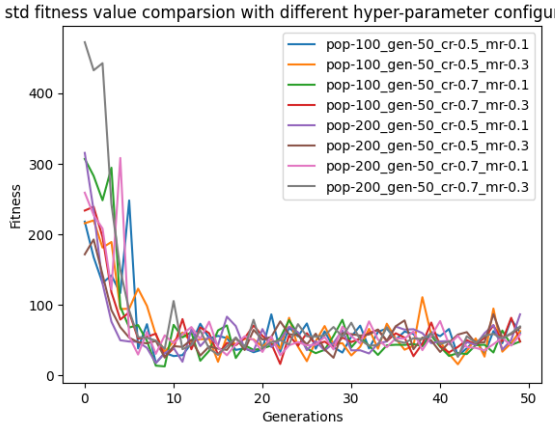



The left column pictures are the statistic figures of single objective optimization, and the right column pictures are the statistic figures of constraint optimization. Just as the same as the problem 2, I set different tries by adjusting hyper-parameters. Even though both of them are not arrive to the global optimal solution, the results are super near the global minimum.

The fitness error of single objective optimization is `min:0.010715169866754191 max:981.6644088910125 mean:23.44821230547529 median:0.01801048593527593 std:92.4774022188559`, where the fitness error of constraint optimization is `min:8.138668064486233e-05 max:687.4661555088726 mean:17.89533431298032 median:0.0026709492816758846 std:69.52604138474383`

**4.Problem 4**

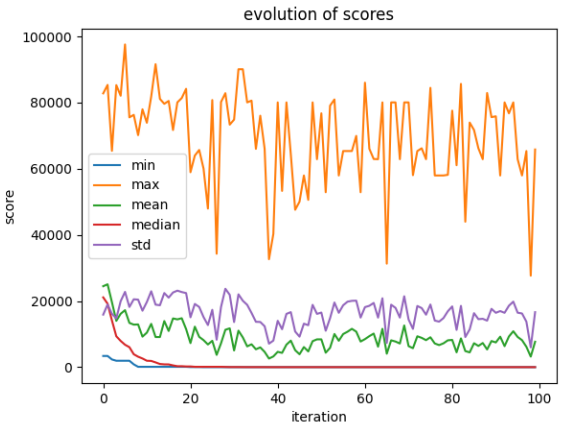
In this problem, there is a mapping `L->D` from letter to digit, that can make `SANTA - CLAUS = XMAX`.

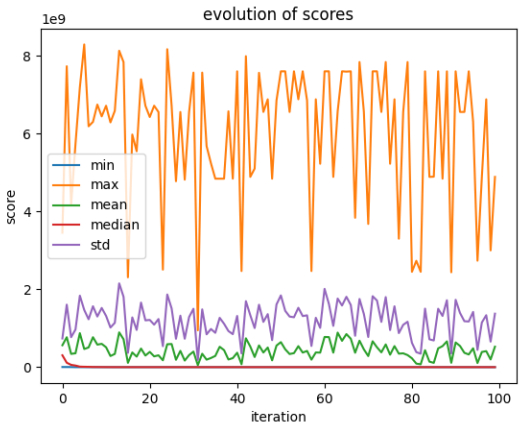
I ensure uniqueness by generating a population using the permutation method with a range list from 1 to 9. Additionally, I utilize `order crossover` and `inversion mutation` techniques to maintain individual

uniqueness in the letter-to-digit mappings within your representation. The order crossover will pick a random crossover point and copy the elements between those two points from one parent to the other where the other element will be set orderly and avoid duplication. The inversion mutation will pick a random inversion point and reverse the elements between those two points. Hence, the offspring will have unique letter-to-digit mappings.

Since this problem has small search space, I set the population size to 50 and the number of generations to 100. The global minimum is reached fast and one of the best mapping solution is [4 0 9 2 6 1 8 3 5 7].

The default fitness function is effective as our goal is to achieve 'SANTA−CLAUS=XMAS', aiming to minimize the absolute error ('SANTA−CLAUS-XMAS') to zero. Moreover, I changed the fitness function by incorporating squared error, which yield better behaviour of convergence. The figures shown below compare the fitness function with and without squared error.




The left figure above is using abstract error as fitness value, where the right one is using square error as fitness value. According to the mean and median line in figures, it can be observed that the fitness function with squared error is more effective in minimizing the absolute error.