CSC3423 Bio-Computing

Coursework 1: Biologically-inspired computing for optimisation

B9052919 Ryota Fuwa

1 Genetic Algorithm (GA)

1.1 Description of GA

Genetic Algorithm is an algorithm inspired by the process of natural selection, where "individuals which are better adapted to the environment have more changes of reproducing" [1]. GA mainly consists of four parts: Evaluation, Selection, Crossover, Mutation. Figure1 below depicts the overview of the algorithm.

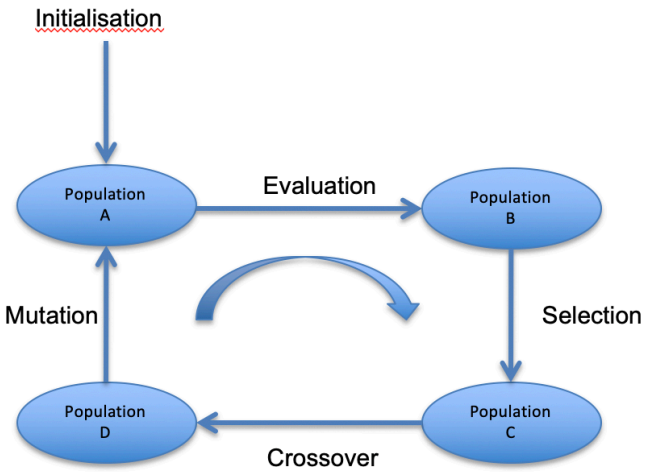

Figure1 Overview of GA [2]

First, all chromosomes in the population are fed to the fitness function, in our case, it is just a mathematical formula. Based on the fitness values, surviving chromosomes will be selected. After that, the survived chromosomes known as parents create new chromosomes known as children. There are a number of methods of selection and crossover operations, so the programmers have the responsibility for choosing appropriate operations. Then at the end of the cycle, a few, depending on the hyper parameter, of gene in some chromosomes will be changed as the mutation process. By repeating this cycle over and over, individuals the most adapted to the fitness function will survive at the end.

1.2 Calibration Process and Evaluation for GA

1.2.1 Scenario 1 (1M evaluation)

First, I simply run the program with default parameters to see how fitness values are moving through generations and drew a plot on Figure2 below.

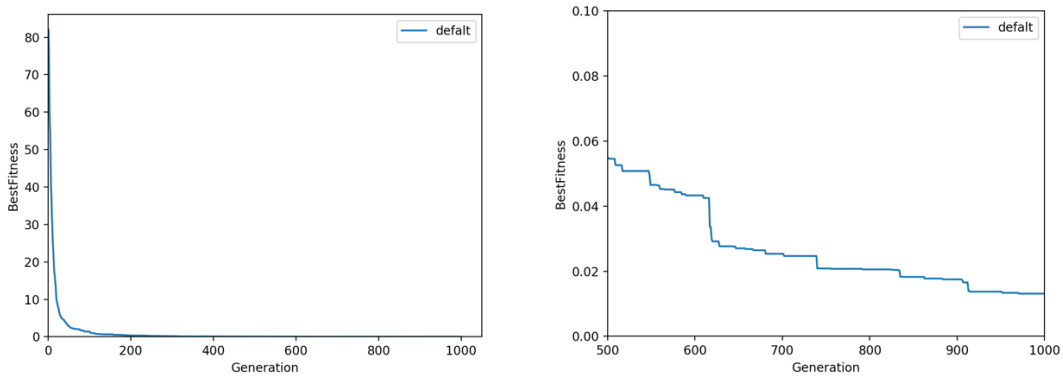

Figure2 Default Configuration Result (right: overall, left: at the end)

Next, I executed it with different types of Selectors. Figure3 shows results of executions with different type of selectors, but all other operations and parameters are the same as default.

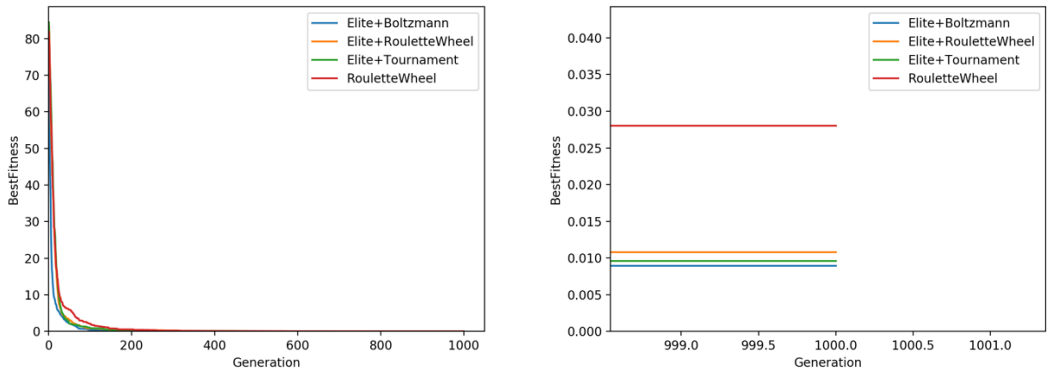

Figure3 Different Selector (R: overview, L: at the end)

The result without elitism (orange in Figure3) was much higher than the others with elitism. Boltzmann selector, which is similar to roulette wheel selector, but the probability for each chromosome are calculated using exponential. Boltzmann and Tournament selectors can be optimized with their parameters to make the algorithm exploitative or the other way.

As to Mutator, Swap mutator doesn't have much meaning itself since all it does is just to change genes' positions in chromosome, and this fitness function doesn't take geographical information in chromosome, in other words, the order of the genes in chromosome doesn't matter. However, When Swap mutator and MultiPoint crossover are put together, I thought it could have a positive effect on this GA process. I demonstrated the idea behind it with Figure4 below.

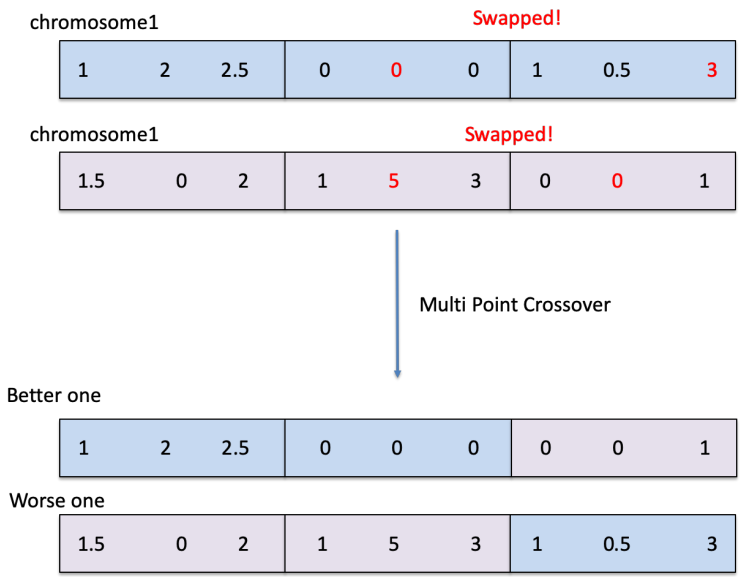

Figure4 Overview of Swap Mutator & Multi Point Crossover

After genes in each chromosome will be swapped, MultiPoint crossover possibly produces better chromosomes by switching parts of chromosomes. I drew plots for the execution with Swap Mutator and different numbers of crossover points in Figure5

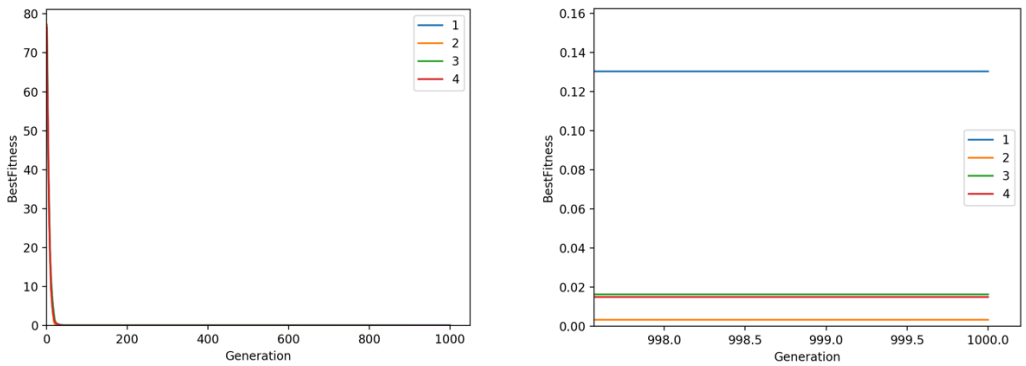

Figure5 Number of Crossover Point with Swap Mutator (R: whole, L: end)

Figure5 suggests that MultiPoint crossover is more efficient than SinglePoint crossover. Nevertheless, it got worse if it was more than 2 crossover points.

Next, I changed the combination of population size and iteration from (1000, 1000) to (200, 500) so that it has more chances of having swapping-and-crossover situation explained in Figure4. In addition to it, I added Gaussian mutation to make the GA process more active because only swapping and crossover don't change values itself, so it could be stuck with only the same values moving around inside. I plotted the result of this configuration in Figure6, and I call this configuration Swap and Gaussian from now on.

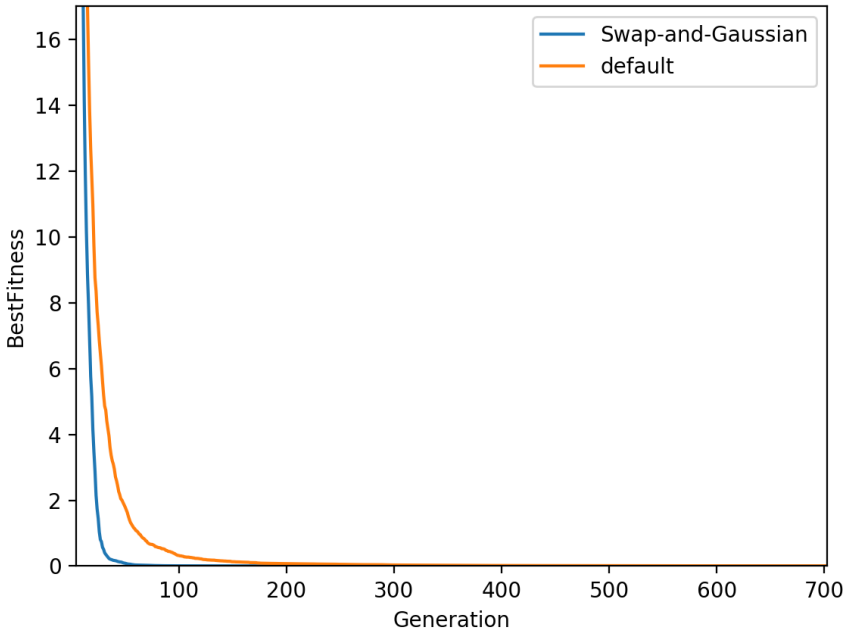

Figure6 Comparison to the Default Configuration

Performance with the new configuration is much higher than that with default configuration. The blue graph for the new configuration continues till 5000 generations and end up being much closer to 0.

Next, I optimized parameters. I made Swap Mutator Probability and MultiPoint Crossover Probability relatively higher because swapping and switching with crossover possibly create better chromosomes, and those operations won't change genes' values, so it can't be harmful with this fitness function. The parameters with which the program produced a great value (relatively low) is shown in Table1.

Table1 Parameter Details for Swap and Gaussian

Population Size	200
Generation (Iteration)	5000
Survivors Number	3
MultiPoint Crossover Points	2
b Constant Value of Boltzmann Selector	3
Probability of Gaussian Mutation	0.1
Probability of Swap Mutation	0.4
Probability of MultiPoint Crossover	0.4

I also tried with Intermediate crossover operation, which basically creates children who have values between the ones their parents have. I guessed all chromosomes lead to 0 or really close to 0 after a certain number of generations considering how the crossover works. Figure7 shows you the result, and compare it with the other configuration, and Table2 also shows you the parameter details I used. I call this configuration Gauss and Intermediate from now on.

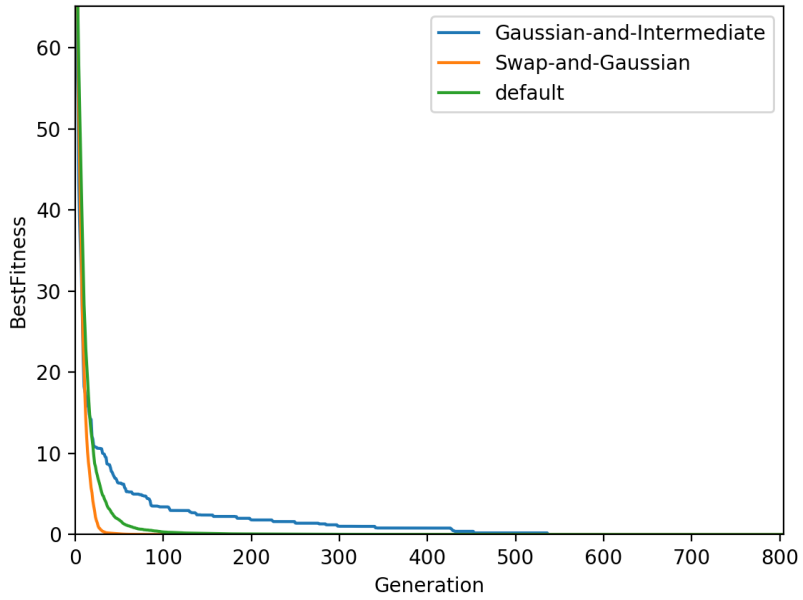

Figure7 Comparison of All Configurations

Table2 Parameter Details for Gauss and Intermediate

Population Size	200
Generation (Iteration)	5000
Survivors Number	5
b Constant Value of Boltzmann Selector	4
Probability of Gaussian Mutation	0.005
Probability of Intermediate Crossover	0.4

As you can see in Figure7, the fitness values of Gaussian and Intermediate didn't fall down as quickly as others; however, it keeps going down slowly, and it went to 0 at the end. I listed the statistics for the results of them in Table2 below.

Table3 Statistics for Each Configuration (10M)

Configuration	Default (Elite & Tournament Selector, SinglePoint Crossover, Gaussian Mutator,)	Swap and Gaussian	Gaussian and Intermediate
Population Size, Iteration	1000, 1000	200, 5000	200, 5000
Best Fitness Value	0.0022054	7.80034 x 10e-7	0
Mean Fitness Value	10.857	15.939	3.91416
Execution Time	1.7477	3.07563	2.23577
Beginning of Convergence	450	100	None

When it comes to execution time and mean fitness value, Swap and Gaussian configuration is not desirable since it uses two mutators, and it is explorative, so I think it also produces a number of "bad children" in each cycle. Gaussian and Intermediate configuration seems really great based on the best fitness value, mean fitness value and execution time, but its movement is unstable, so it needs a large number of iterations to make sure the fitness values fall down.

1.2.2 Scenario 1 (10k evaluation)

Assuming different evaluation numbers don't affect how each operation worked in 1M evaluation scenario so much, I tried with Swap and Gaussian configuration since Gaussian and Intermediate configuration might not be able to secure enough iterations in 10K scenario. I plotted the results with different parameters in Figure8 and statistics for each in Table4.

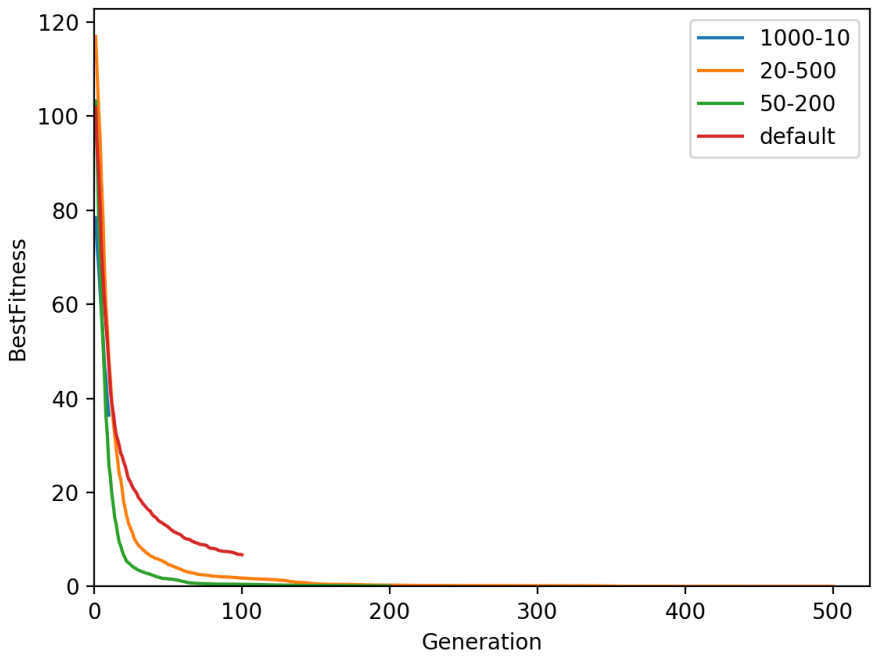

Figure8 10k Scenario Execution with Different Parameters

Table4 Statistics for Each Configuration (10K)

Configuration	Default (Elite & Tournament Selector, SinglePoint Crossover, Gaussian Mutator,)	1000-10 (Swap and Gaussian)	20-500 (Swap and Gaussian)	50-200 (Swap and Gaussian)
Population Size, Iteration	100, 100	1000, 10	20, 500	50, 200
Best Fitness Value	6.7387	36.435	0.043979	0.097950
Mean Fitness Value	37.85	31.858	30.477	36.174
Execution Time	1.7477	0.15767	0.19721	0.21342
Beginning of Convergence	None	None	160	100

According to Table4, 20-500 produced the best fitness value and improved from the default configuration by more than 10e-2. Moreover, unlike Gaussian and Intermediate configuration, Swap and Gaussian configuration decreased constantly, so it can be considered to be stable. I listed parameter details for the best configuration (20-500) in Table5.

Table5 Parameter Details for Gauss and Intermediate (20-500)

Population Size	20
Generation (Iteration)	500
Survivors Number	5
MultiPoint Crossover Points	2
b Constant Value of Boltzmann Selector	3
Probability of Gaussian Mutation	0.15
Probability of Swap Mutation	0.45
Probability of MultiPoint Crossover	0.45

I made each probability higher than those in 1M scenario because this time it only has 500 iteration, so I'd like it to be more active in cycles. To make sure if it's right, I executed the program with lower probability of gaussian mutation, and it only got

2 Particle Swarm Optimization (PSO)

2.1 Description of PSO

Particle Swarm Optimization is also non-deterministic algorithm like GA. In PSO, individuals called particles are placed in a field where each particle has its own position and velocity. Also, a fitness function is assigned on the field so that scores are calculated for each particle, and then each particle change its velocity based on their score and also on other particles. Particles look for the optimum over the iterations by associating themselves with others. There are a lot of different methods for how to make particles interact with each other. For example, which particle interact with which particles, and how much each particle gets affected by others, and so on. This method will be determined by the algorithm designers according to the sort of problems aiming for the efficiency of the algorithm. The diagram of the field is shown in Figure9 below.

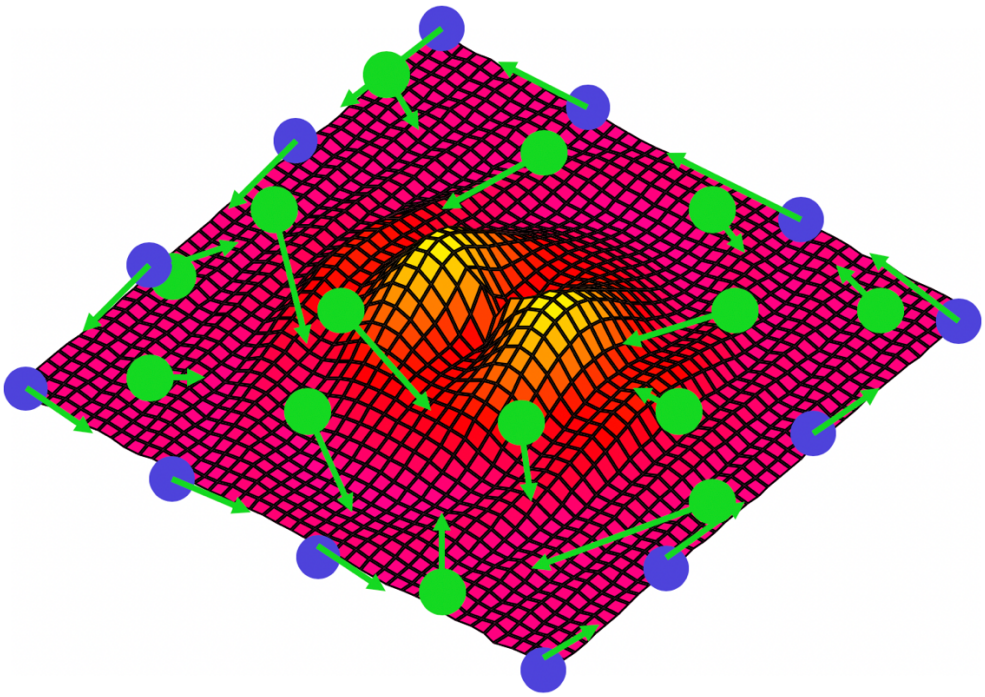

Figure9 Diagram of PSO Field [3]

2.2 Calibration Process and Evaluation for PSO

2.2.1 1M scenario

First, I just run the code with the default configuration, and plotted in Figure10.

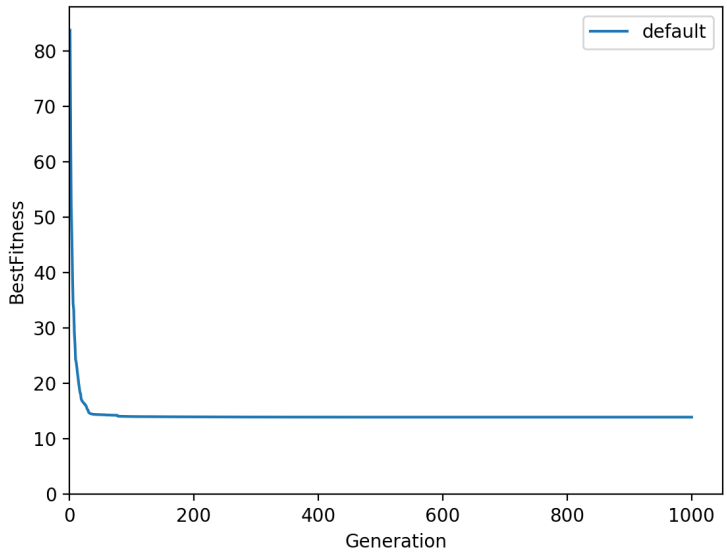

Figure10 Results for Default Configuration (1M)

From Figure10, we can see the best fitness values stopped moving from around 20 generation. I inferred all particles gathered at a local minimum. To avoid this, I made Global weight smaller, and plotted the results for different global weights in Figure11.

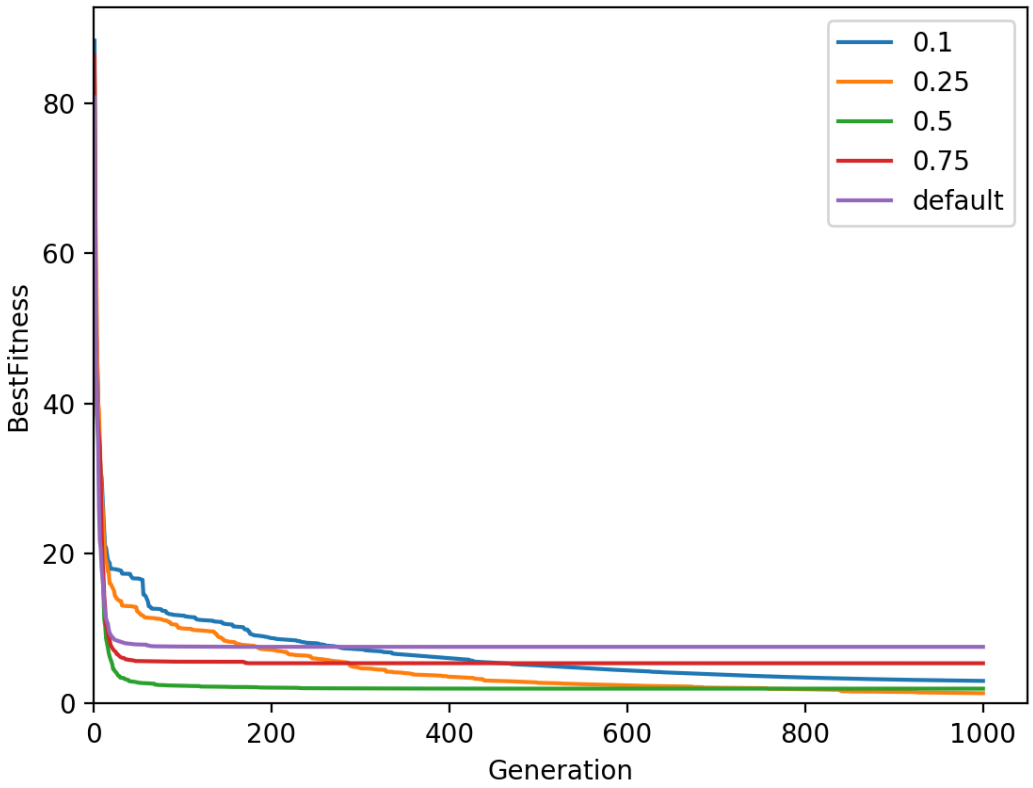

Figure11 Results with Different Global Weight

With global weights lower than 0.65, the fitness values keep decreasing, which means particles aren't stuck with a local minimum. However, if the weight is too small, particles become too slow to reach the lowest place.

Next, I changed Neighborhood weight to make particles active and explorative so that particles find the optimal place more quickly. Figure12 illustrates the outcomes for that.

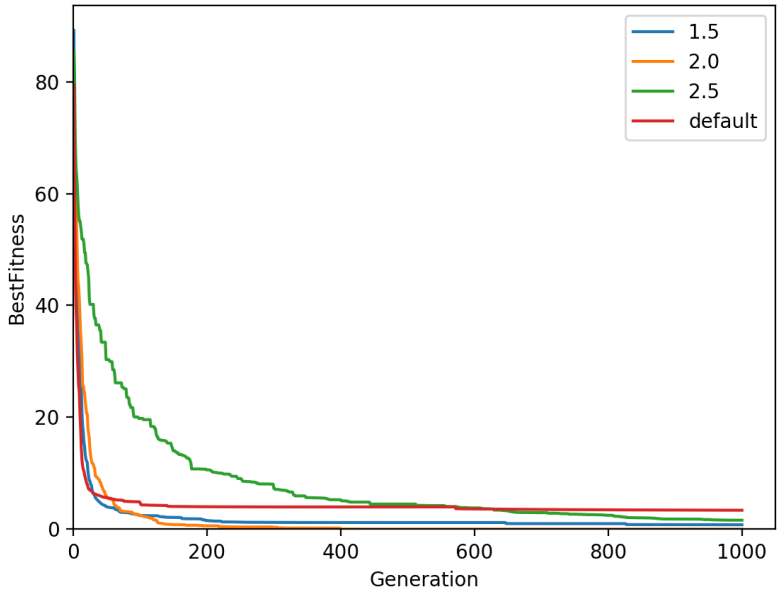

Figure12 Results with Different Neighborhood Weights

If neighborhood weight gets higher, the fitness value tends to become low over the iteration. However, when neighborhood weight is 2.5, it doesn't fall down quickly. I think It's because neighborhood particles are put together too quickly, so explorations don't happen a lot.

Now that I optimized the global weight and the neighborhood weight so that particles will not be stuck with a local minimum, next thing to do is to make the chances that particles find the optimal place instead of local minima as high as possible. First, I changed population size and generation from (1000, 1000) to (500, 2000) to get more generations. When it comes to Inertia weight, I thought the smaller it is, the more directions particles wander and explore. As a matter of fact, it was true based on Figure13 below.

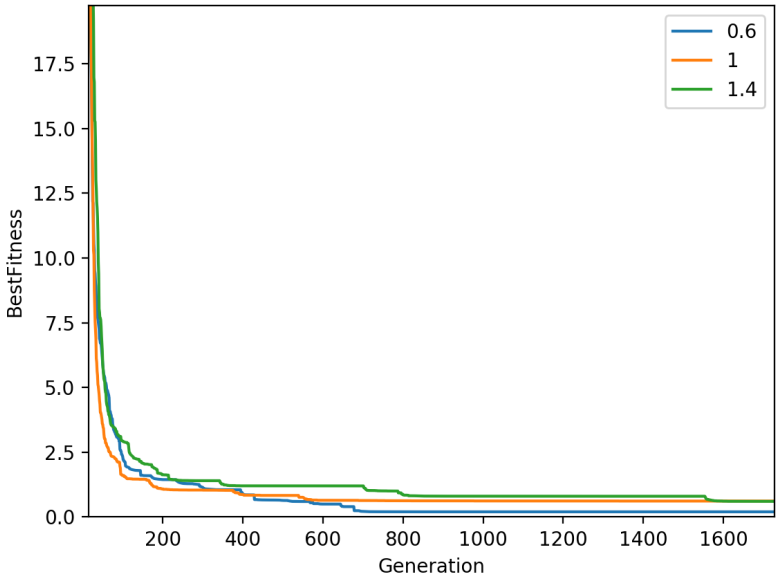

Figure13 Results with Different Inertia weights

Lastly, I changed Personal Weight empirically to seek a suitable configuration, and listed the detail of parameters in Table6.

Table6 Parameter Detail (1M scenario)

Number of Particles	500
Number of Iterations	2000
Neighborhood Weight	1.5
Inertia Weight	0.65
Personal Weight	1.35
Global Weight	0.65
Maximum Velocity	0.01
Number of Particles in One Neighborhood (Without itself)	2
Random Weight	0.00025

In Swarm.java and SwamUpdateSimple.java, I installed RandomIncrement, which adds random values to velocity in every cycle, so I aimd that all particles don't end up in a same local minimum by using this operation. In addition, I reduced the number of particles in a neighborhood to 2, so particles search the field freely from one another.

The best fitness value for this configuration is 3.7500 x 10e-10 (, which is an average of 30 results.)

2.2.2 10K scenario

I took the same steps with 1M scenario, and I listed only the final parameter details in Table7.

Table7 Parameter Detail (1M scenario)

Number of Particles	50
Number of Iterations	200
Neighborhood Weight	1.4
Inertia Weight	0.65
Personal Weight	1.35
Global Weight	0.65
Maximum Velocity	0.01
Number of Particles in One Neighborhood (Without itself)	2
Random Weight	0.00005

Coincidently, almost all parameters in the configuration that gives the best score are same as ones in 1M scenario. The average of the best fitness values of 30 executions was 8.8148. In 10K scenario, the outcomes are much more unstable, so if I take an average of 30 times, it doesn't produce good scores.

3 Comparison of GA and PSO

Chromosomes don't communicate with each other in the process whilst PSO particles does. Therefore, it was easier to find the best fitness value with PSO quicker than GA unless particles are stuck with a local minimum. In GA, I configurated operations and parameters to make fitness values close to 0, on the other hand, I configurated those to prevent all particles from falling into local minima in PSO. when it comes to the result, I've got 0.0 in best fitness value for both algorithms in 10M scenario, but PSO will still sometimes produce a high fitness value because of stagnations at local minima. In 10K scenario, it's much easier to get high performance constantly with GA since PSO in 10K scenario is unstable, or it wouldn't reach low places in time due to the limited number of iterations. I summarized the best fitness values from both algorithms in Table8.

Table8 Comparison of Both Algorithms

Algorithm	GA (1M)	PSO (1M)	GA (10K)	PSO (10K)

The best fitness value	0.0	is 3.7500 x 10e-10 0.0 (one time execution)	0.043979	8.8148 1.9899(one time execution)
State	stable	Fairly stable (about 1 in 40, it's stuck in 10e-2 ~ 10e10)	stable	Unstable (varies with in 12.0 ~ 1.0)

Reference:

[1] Darwin, C. (n.d.). *The origin of species*.

[2] Jaume, Bacardit. (2019). Biologically Inspired Computing Lecture 02 – Genetic Algorithms

[3] Jaume, Bacardit. (2019). Biologically Inspired Computing Lecture 04 - Swarm Intelligence.

[4] Jenetics.io. (2019). [online] Available at: http://jenetics.io/manual/manual-4.3.0.pdf [Accessed 15 Nov. 2019].

[5] En.wikipedia.org. (2019). *Crossover (genetic algorithm)*. [online] Available at: https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm) [Accessed 15 Nov. 2019].
[6] En.wikipedia.org. (2019). *Mutation (genetic algorithm)*. [online] Available at: https://en.wikipedia.org/wiki/Mutation_(genetic_algorithm) [Accessed 15 Nov. 2019].
[7] En.wikipedia.org. (2019). *Particle swarm optimization*. [online] Available at: https://en.wikipedia.org/wiki/Particle_swarm_optimization [Accessed 15 Nov. 2019].