Randomized Optimization

Georgia Institute of Technology CS 7641: Machine Learning Assignment Two

Xi Han

GT ID: xhan306

Introduction

This report explores 4 randomized optimization algorithms in machine learning by replacing backpropagation of neural networks which has been explored in the previous project with three randomized optimization algorithms, randomized hill climbing, simulated annealing, genetic algorithms, and investigating three optimization problems selected by ourselves with four randomized optimization algorithms, randomized hill climbing, simulated annealing, genetic algorithms, and MIMIC. Protein dataset which is mentioned in the last project was used for the first assignment in this report.

1. Randomized Optimization for Neural Network Weights

1.1 Introduction

In the last project, a neural network model was built for two datasets including protein dataset which was also used in this project. Protein solubility can be predicted by protein sequence and this dataset includes 3148 proteins in total. Protein solubility can represent the function of protein and activity to some extent. Improving protein solubility can reduce the cost significantly in biocatalyst industry where most biocatalysts are proteins. Some conventional strategies have also been studied such as metabolic engineering. However, wet-lab experiments are time-consuming and expensive. In addition, experiments *in vivo* always fails due to opaque reasons. Therefore, an accurate model for predicting protein solubility is required and can be constructed with the aids of machine learning algorithms. This dataset was selected to explore three optimization algorithms here.

Data pre-processing is a critical part in the data mining. The characters of protein sequence were converted into numerical values. 70% data of 3148 samples were training data and remaining were test data to measure the performance of models. The error in all the figures refers to the mean squared error. All algorithm implemented are from the ABAGAIL package in Java.

There are various methods to update the weights of neural networks. One of which is backpropagation. In our study, three randomized optimization algorithms were used to optimize the weights and the performance was compared for randomized hill climbing, simulated annealing, genetic algorithms by recording the error in each iteration.

1.2 Randomized Hill Climbing (RHC)

Hill climbing is a mathematical optimization technique for local search. It is an iterative algorithm that starts with a random point for a problem, then attempts to find a better solution by making an incremental change to the solution. If the change produces a better solution, another incremental change is made to the new solution, and so on until no further improvements can be found. Hill climbing is the simplest randomized optimization algorithm and easy for understanding and implementation. Hill climbing can find optimal solutions for convex problems. However, for other problems it will find only local optima.
We run RHC and the results are recorded in Figure 1. The training error and test error gradually decreased until about 250 iterations. Then the error became stable and continued to decrease slowly after that point. It can be observed that the final error was about 0.117 for test data. The implementation of this sample algorithm is the fastest in all algorithms.

1.3 Simulated Annealing (SA)

The simulated annealing algorithm was originally inspired from the process of annealing in metal work. Annealing involves heating and cooling a material to alter its physical properties due to the changes in its internal structure. As the metal cools its new structure becomes fixed, consequently causing the metal to retain its newly obtained properties. In simulated annealing, we keep a temperature variable to simulate this heating process. We initially set it high and then allow it to slowly 'cool' as the algorithm runs. While this temperature variable is high the algorithm will be allowed, with more frequency, to accept solutions that are worse than our current solution. This gives the algorithm the ability to jump out of any local optimums it finds itself in early on in execution. As the temperature is reduced so is the chance of accepting worse solutions, therefore allowing the algorithm to gradually focus in on an area of the search space in which hopefully, a close to optimum solution can be found. This gradual 'cooling' process is what makes the simulated annealing algorithm remarkably effective at finding a close to optimum solution when dealing with large problems which contain numerous local optimums. Simulated annealing is sometimes empirically much better at avoiding local minima than hill climbing.
There are two hyperparameters we can tune for SA. Just like the process of heating and cooling metals. Cooling rate (CR) and starting temperature (Temp) can influence the performance of this algorithm. At first, we changed the cooling rate and plotted the training error in Figure 2, test error in Figure 3. Cooling rate is a constant within 0 to 1, which represents the rate of temperature decreasing for each iteration and lower values mean it cools faster. In the process of implementation SA, cooling rate corresponds that how much the algorithm can jump between points. In other words, the lower the

temperature at one iteration, it is easier to move this algorithm towards the solution, and also higher chances of getting stuck in local optima. In our study, five different cooling rates were explored. It can be observed from both Figure 2 and 3, when the cooling rate is 0.95, the error for both training error and test error is large. Cooling rate 0.75 and 0.55 result in small error within 1000 iterations. The error is gradually decreasing and it may continue to reduce for more iterations. Not only a final small error is required, fast speed to achieve stable status and keeping small error are desired. Therefore, cooling rate 0.75 seems preferred for this aim. It keeps small error within 1000 iterations and smaller intervals can be explored in the future, such as cooling rate 0.65 between 0.55 and 0.75. A methodology is provided here and if better results are highly required, more searching work can be done.
In addition, another hyperparameter starting temperature was investigated in Figure 4 and 5. It can be seen that when the starting temperature is 1E12 and 1E13, there is a big peak at about 650 iterations. It means higher starting temperature causes worse performance of SA. It is same with cooling rate selection, we prefer to pick some values that converge quickly and surpass others consistently. IE11 was chosen according to this rule and best error is about 0.12 for test data.
Therefore, through our analysis above, cooling rate 0.75 and starting temperature 1E11 are suitable hyperparameters for our protein dataset.

1.4 Genetic Algorithms (GA)

Genetic algorithms are based on the process of evolution in natural selection which has been observed in nature. They essentially replicate the way in which life uses evolution to find solutions to real world problems. Strong individuals will dominate the generation through many iterations and there are mutation and crossover in the process. Fitness values were calculated to measure the performance of each generation. GA is pretty simple to use and understand but the speed of convergence is the slowest one among four randomized optimization algorithms.
Two hyperparameters population size (POP) and the number of individuals mutating with each generation (Mutate) were studied in Figure 6-9. For different population size, the error behaves erratically and only a vague trend can be seen. Moreover, the population size seems have little effect on the error. However, if we had very large population sizes, we have a very slow random search. On this basis, we pick 100 as a safe value for the population size.
For Mutate, we also pick five values from 0-100 to explore the effect for error. The figures show similar trend with POP. However, Mutate 25 seems have less peaks in the figures and the error is always low within 1000 iterations.
Therefore, population size 100 and the number of individuals mutating with each generation 25 were selected in GA for our dataset.

2. Three Optimization Problems

2.1 Traveling Salesman Problem

The traveling salesman problem (TSP) is a famous NP-complete combinatorial optimization problem that asks a very general question: given a list of cities, and the distance between each pair of cities, what is the shortest possible path a traveling salesman can take to visit each city exactly once which also takes him back to his city of origin. Four random optimization algorithms were explored in Figure 10. GA achieved best performance among four algorithms for this problem and Iteration 1000, POP 200, Mate 150, Mutate 20 were used here. GA is nondeterministic and modelled on biological processes, lending itself incredibly well to rapid heuristics, allowing it here to outperform the algorithms in fitness and in number of iterations. Therefore, GA is the ideal solution for the NP-complete TSP problem.

2.2 Flip Flop Problem

Flip Flop (FF) is a very simple problem where there is a fitness function that takes bit strings as input and returns the number of times that bits alternate in a bit string. In other words, going from a number to any other number in the next digit will add 1 to a count, beginning from 0. This count is what is eventually returned by the function. An example is that "1111" would return 0 while "1010" would return 4. A maximum fitness bit string would be one that when passed into the function, returned its original length; by this logic, the ideal bit string would consist entirely of alternating digits. Among four algorithms, MIMIC achieved best performance with Iterations 1000, CR 0.95, Temp 100. This is may be caused by that MIMC can convey structure, whereas all the others do not have structures. In addition, MIMC can directly model probability distribution and successfully refine model. Therefore, MIMIC beats other algorithms within 1000 iterations.

2.3 Max K-Colouring Problem

A graph is defined as K-colorable if each node of the graph can be colored with one of the K colors and have no nodes of the same color connected to each other. Much like the traveling salesman problem, the maximum K-coloring problem is known to be NP-complete, and is defined as finding some K-coloring of a graph that minimizes the number of pairs of identically colored and connected nodes. The graphs used are of size 50, and 8 colors, and for each graph, there exists a solution. The fitness function is based off of whether an algorithm can find the solution, and how quickly the algorithm can find it. In Figure 12, it can be demonstrated that MIMIC with Iterations 1000, POP 200, KEEP 100 achieves the best fitness value within 50 iterations. It was obvious that MIMIC rapidly reached the global optimum sooner than the genetic algorithm and outperformed GA for all 50 iterations. It may because a structure is needed for lots of data and MIMIC is the algorithm which can build a structure among all four algorithms. Although GA was able to build a structure based on its knowledge slowly, MIMIC beats GA successful in the speed and solves the problem better.

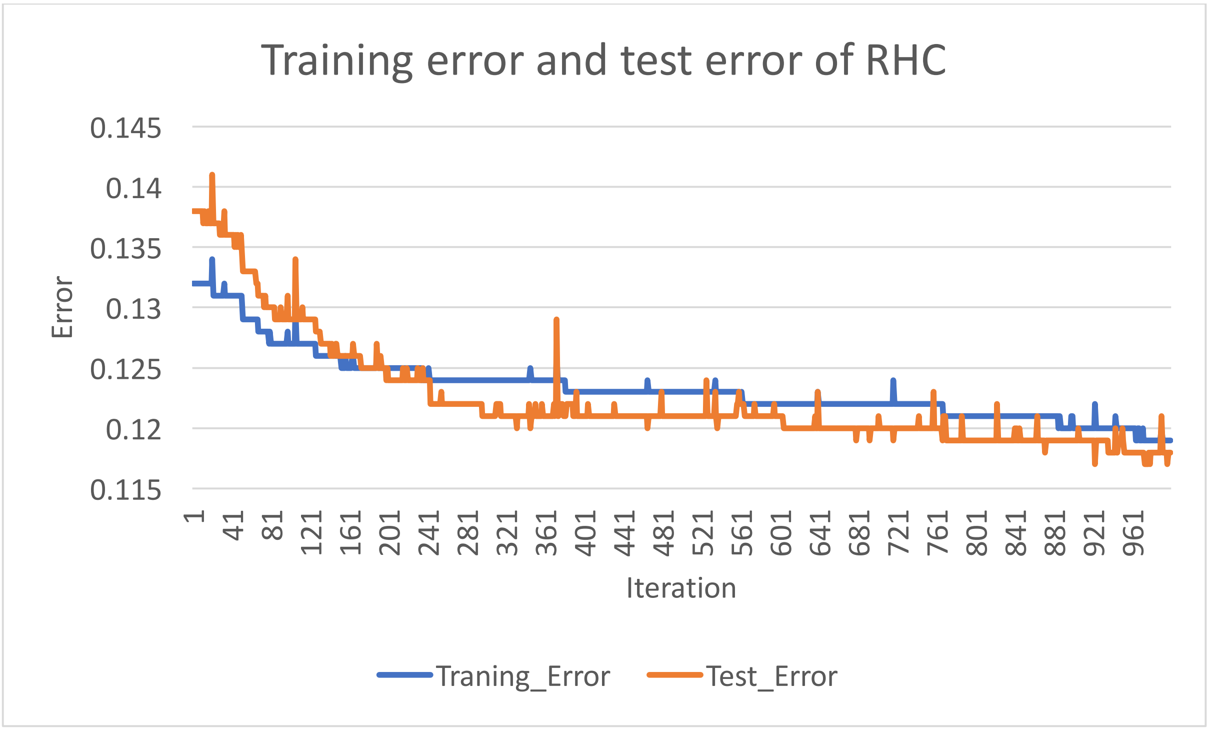

Figure 1. The training error and test error of RHC

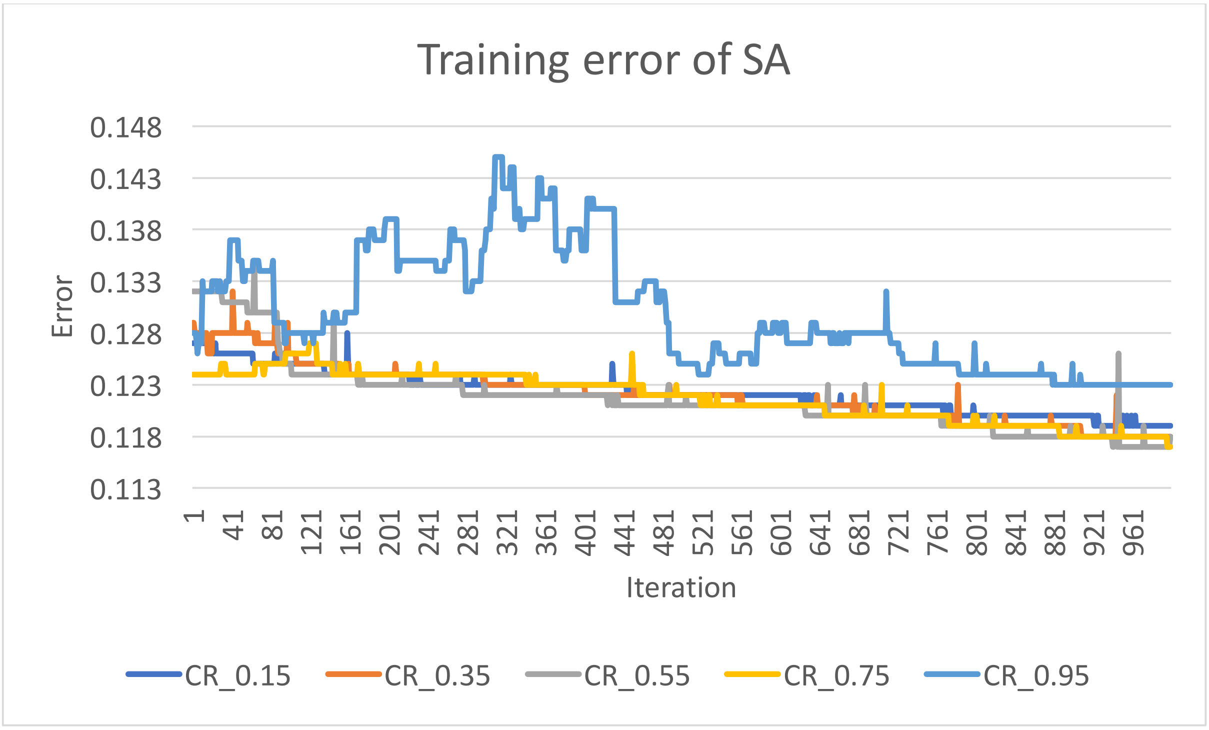

Figure 2. The training error of SA for different cooling rates

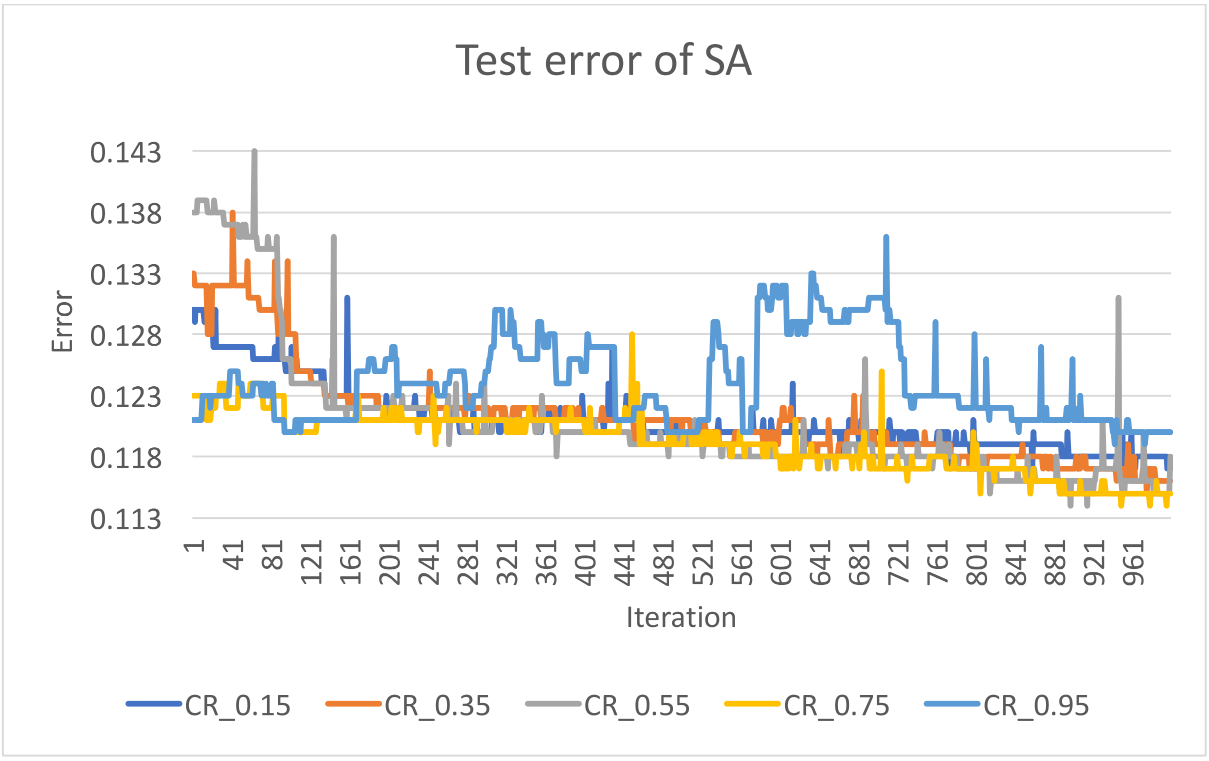

Figure 3. The test error of SA for different cooling rates

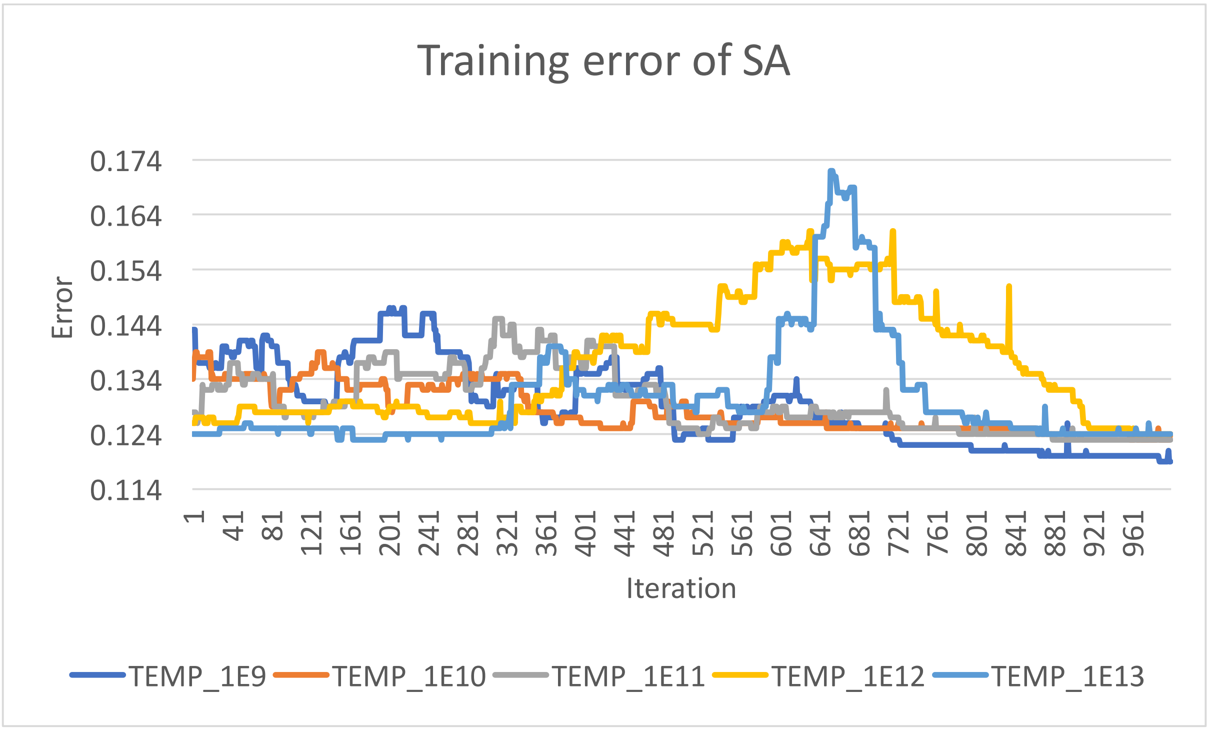

Figure 4. The training error of SA for different starting temperatures

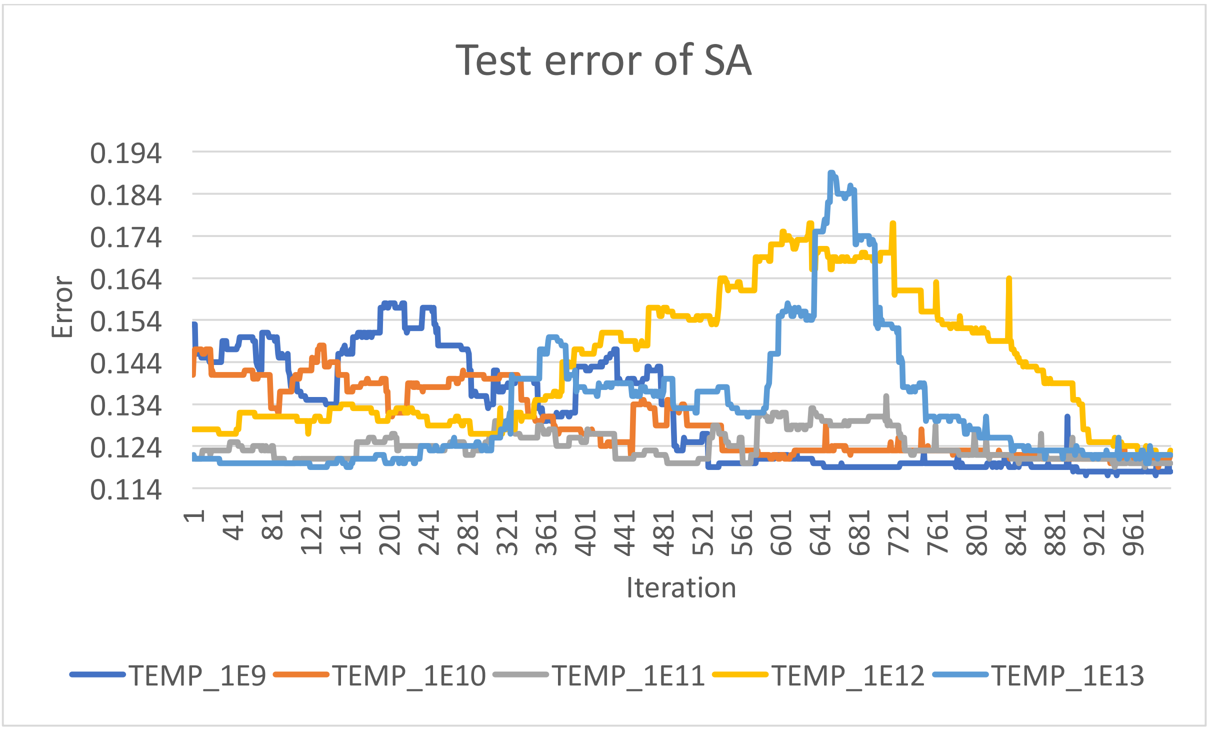

Figure 5. The test error of SA for different starting temperatures

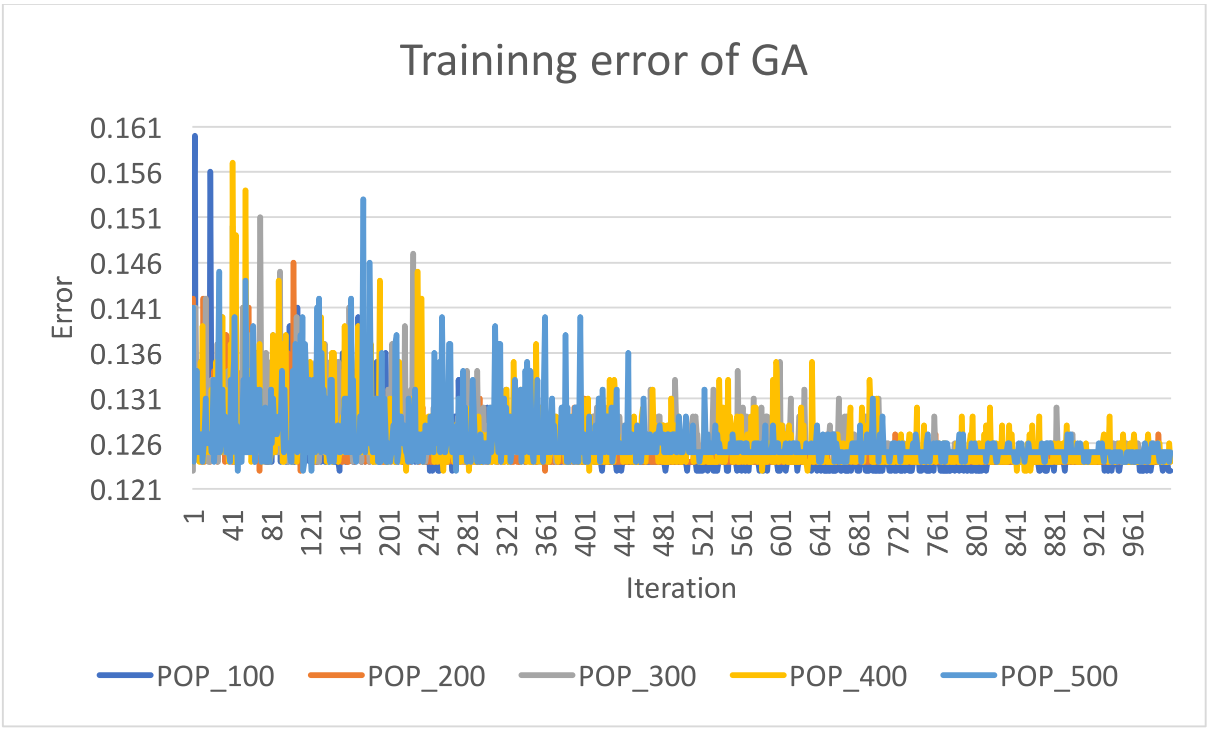

Figure 6. The training error of GA for different population sizes

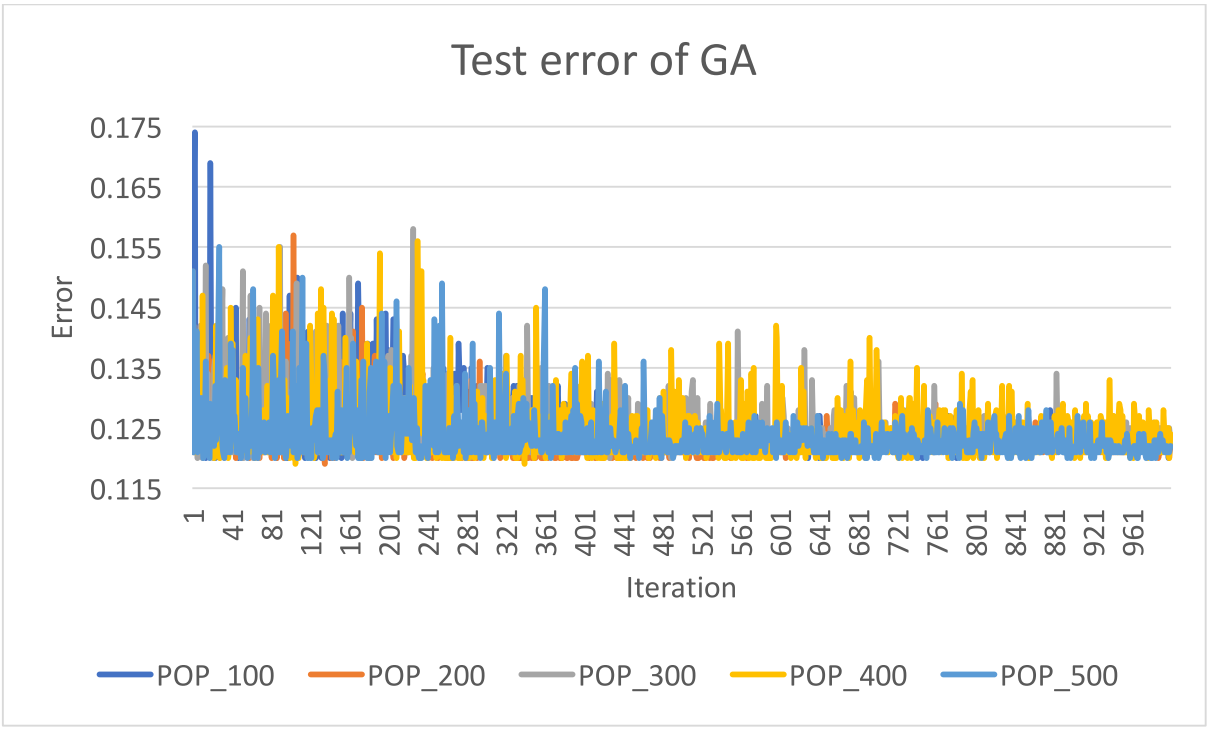

Figure 7. The test error of GA for different population sizes

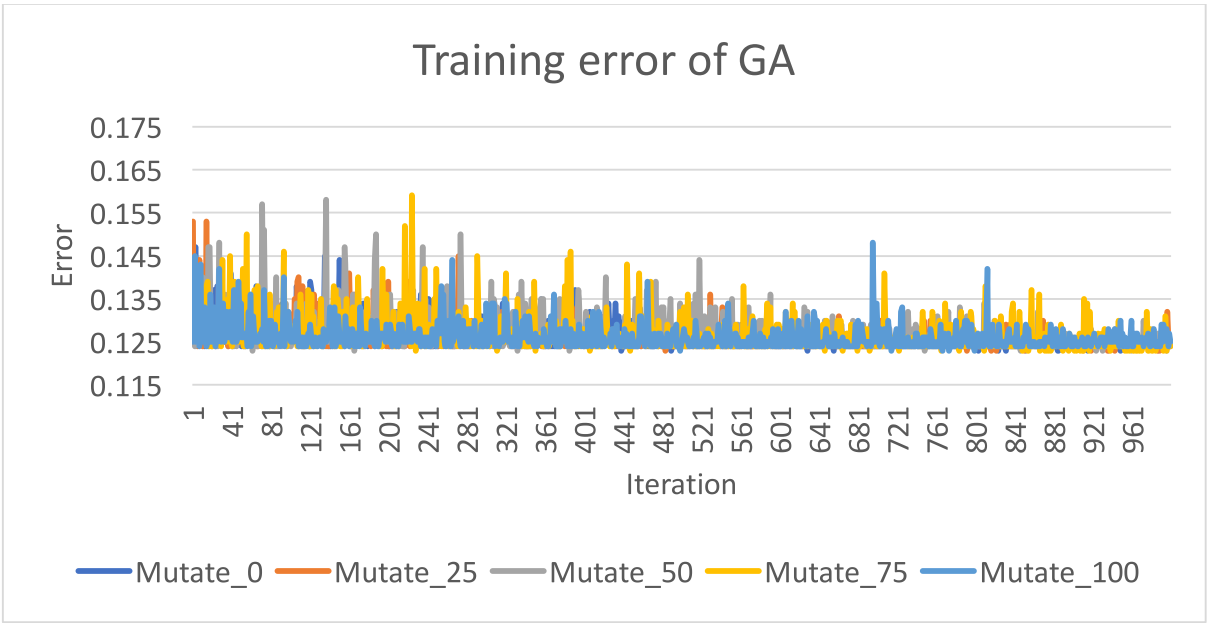

Figure 8. The training error of GA for different mutations

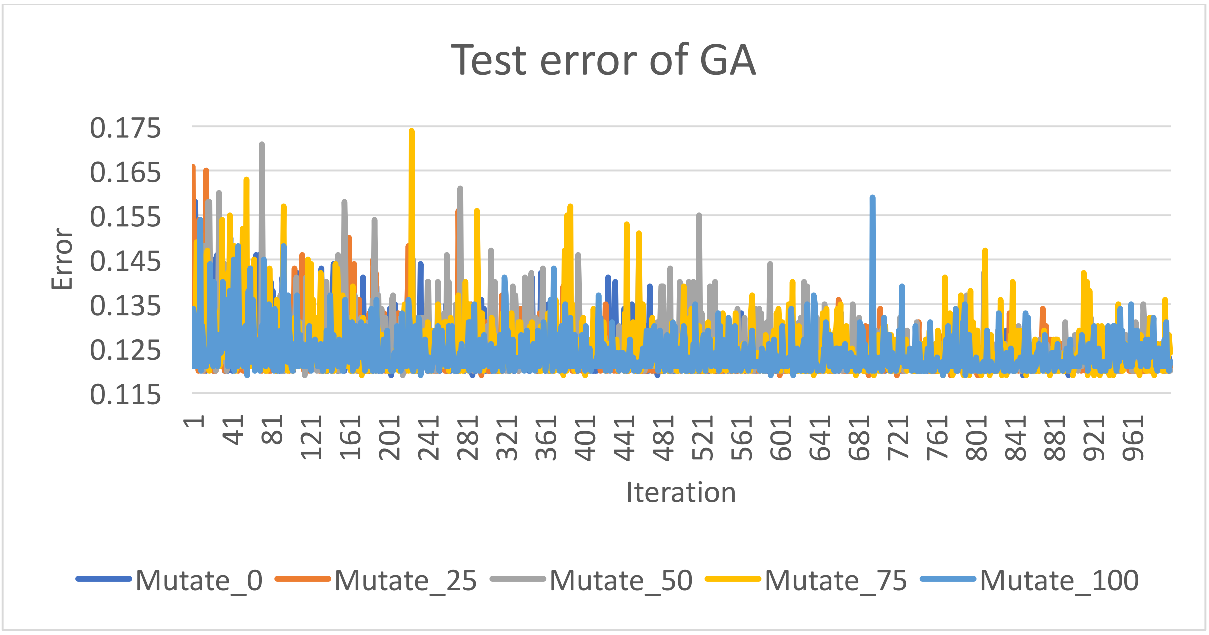

Figure 9. The test error of GA for different mutations

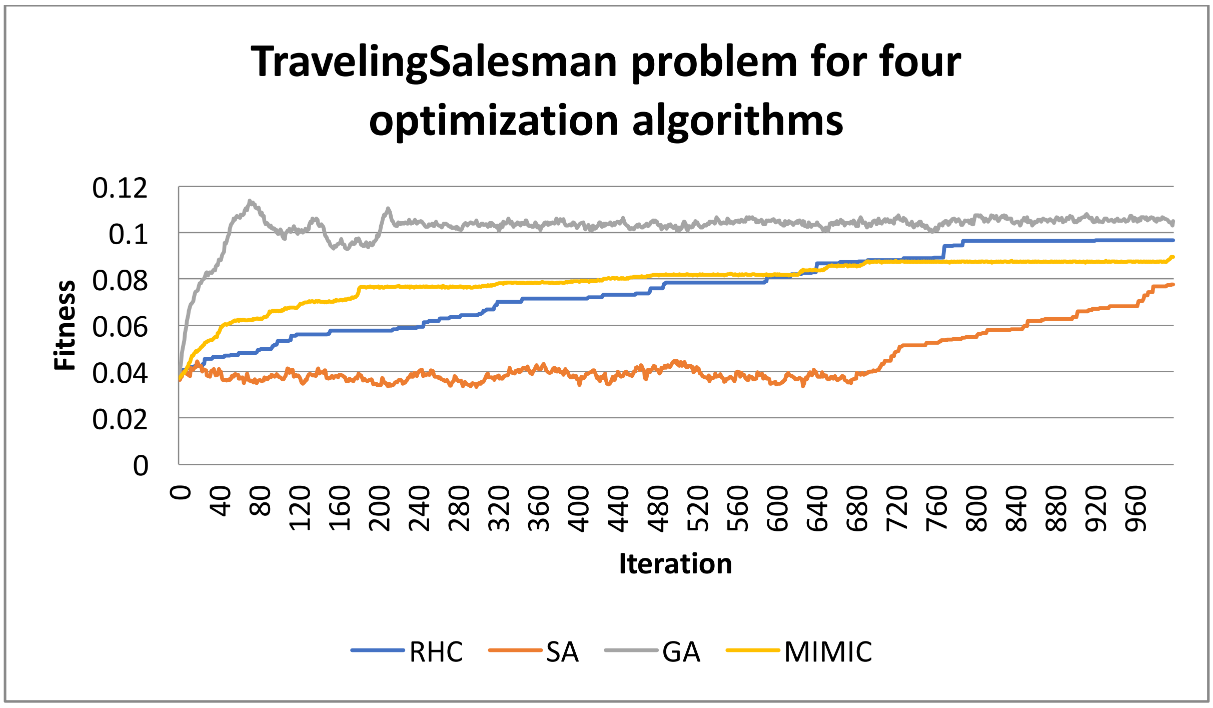

Figure 10. Traveling Salesman Problem for four algorithms

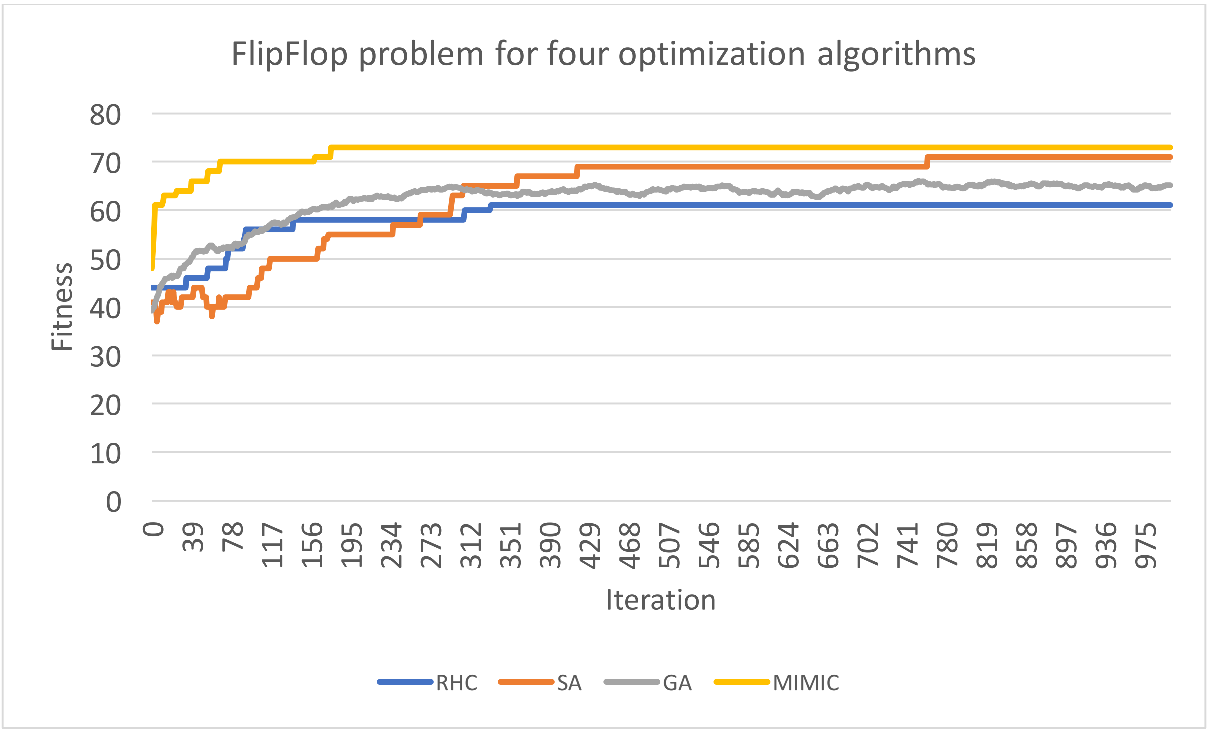

Figure 11. Flip Flop Problem for four algorithms

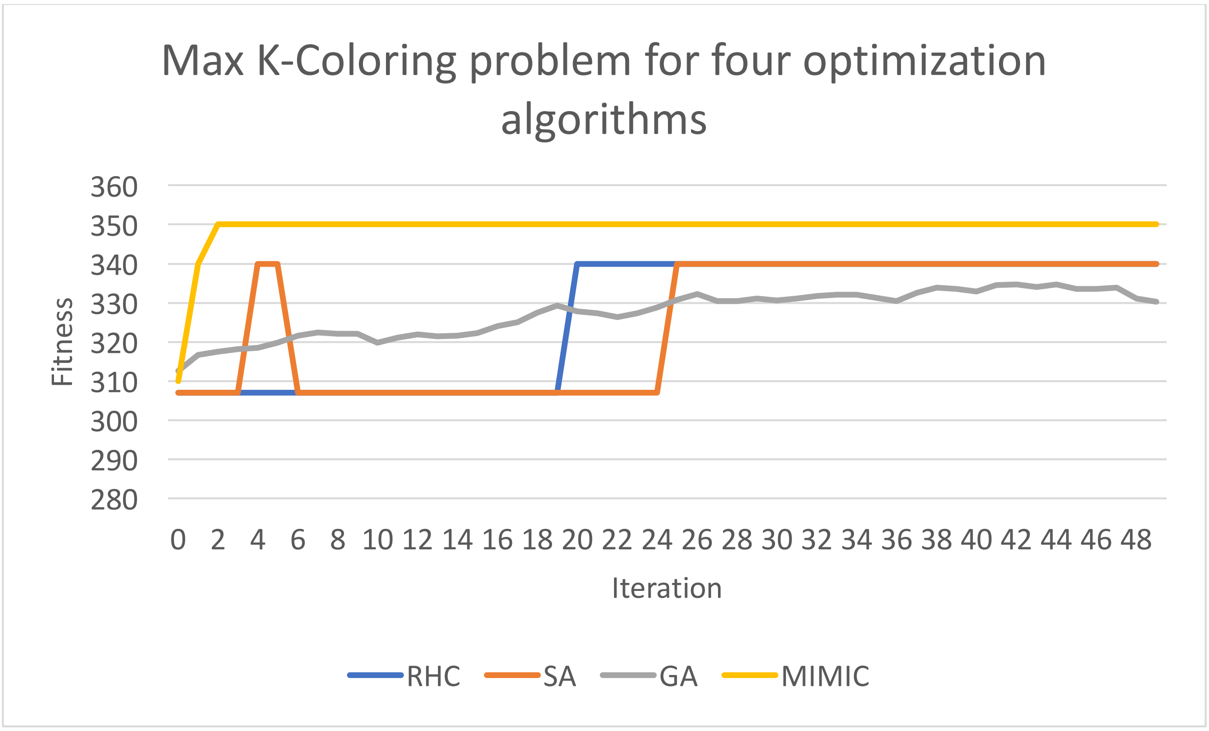

Figure 12. Max K-Coloring Problem for four algorithms