PRATICAL ASSIGNMENT: SOLVING PBO PROBLEMS WITH EVOLUTIONARY ALGORITHMS

Xiang He
s3627136
1234@umail.leidenuniv.nl

Shupei Li
s3430863
s.li.18@umail.leidenuniv.nl

November 27, 2023

1 Introduction

The evolutionary algorithm is one type of heuristic algorithm. It is inspired by the evolutionary process in nature and can be applied in various fields, such as engineering electrical electronics, automation control systems, etc [1]. Generally, evolutionary algorithms begin with population initialization, which generates a set of individuals randomly. Then, the population is updated during the iteration until termination conditions are satisfied. Evolutionary algorithms guide the optimization direction via two basic operators, namely variation and selection [2]. Variation includes recombination and mutation. It encourages the diversity of the population and a greater coverage of the search space. Selection consists of a series of evaluation strategies determining survivors during each epoch. It controls the overall quality of solution candidates and ensures the optimization does not deviate from the correct direction. The alternation between variation and selection simulates the gene inheritance and natural selection that balances between exploration and exploitation. In this assignment, we investigate two representative evolutionary algorithms — the genetic algorithm and the evolution strategy. Specifically, we apply these two algorithms to solve two pseudo-boolean optimization (PBO) problems.

The domain of PBO problems is usually defined on (or can be easily transformed into) $\{0,1\}^n$, where n represents the problem dimension. Meanwhile, the range of the problem is often limited on the field of real numbers $\mathbb{R}$. Given some constraints, our goal is finding a model satisfied $\{0,1\}^n \to \mathbb{R}$ that maximizes (or minimizes) the objective function F. The first problem we aim to solve is the low autocorrelation binary sequences (LABS) problem. The input $\mathbf{s}$ of the LABS problem is usually defined on $\{-1,1\}^n$. The objective of the LABS problem is:

$$\min\ E(\mathbf{s}) = \sum_{k=1}^{n-1} \left(\sum_{i=1}^{n-k} s_i s_{i+k} \right)^2 .$$

Or we can also maximize the merit factor equivalently, which is defined as:

$$\max\ F(\mathbf{s}) = \frac{n^2}{2E(\mathbf{s})}.$$

The term $\frac{E(\mathbf{s})}{n}$ is the total energy of n Ising spins $s_i \in \{-1,1\}, i = 1,\ldots,n$ in statistical physics [3]. Variants of the LABS problem have application value in fields like communication engineering and applied mathematics. However, our implementation transforms the domain into $\{0,1\}^n$ to ensure the compatibility of the software environment. The final objective function can be written as:

$$\max F(\mathbf{x}) = \frac{n^2}{2\sum_{k=1}^{n-1}\left(\sum_{i=1}^{n-k} s_i \cdot s_{i+k}\right)^2}$$
$$s.t. \quad s_i = 2x_i - 1,\ x_i \in \{0,1\}, i = 1,\ldots,n.$$

Figure 1 shows the largest known merit factors. Factors are confirmed to be optimal solutions when $n \leq 66$.

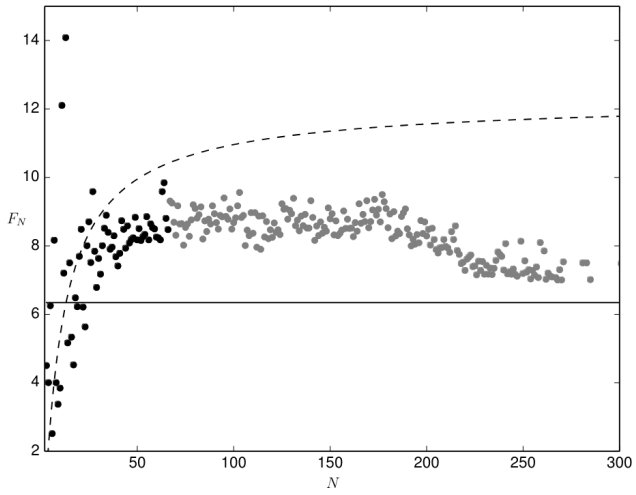


Figure 1: Largest known merit factors (optimal when $n \leq 66$). This figure is directly adapted from [3].

The second problem we focus on in this assignment is the Ising model. The Ising model is an NP-hard problem originating from the ferromagnetism theory. Its general form is defined on an undirected graph $G = (V, E)$. Each vertex in the graph corresponds to an Ising spin $s_i \in \{-1, 1\}$. Given a weight function w, the objective is maximizing $\sum_i \sum_j s_i \cdot s_j \cdot w(e_{ij})$, where $s_i, s_j \in V$ and $e_{ij} \in E$ [4]. In this assignment, we only consider the one-dimensional case, where the structure of the Ising model degenerates to a ring. With assumptions of zero external magnetic fields and a unit interaction strength, the objective function of the Ising model is:

$$\max\ F(\mathbf{x}) = \sum_{i,j,e_{ij}\in E} \left[x_i x_j + (1-x_i)(1-x_j)\right].$$

Notice that the similar transformation trick is applied: $x_i = \frac{s_i+1}{2}, i = 1,\ldots,n$. Consequently, the search space is mapped onto $\{0,1\}^n$.

Our algorithm implementation uses the IOHprofiler framework as the backbone. IOHprofiler integrates many benchmark problems and supports three types of analysis — fixed-target, fixed-budget, and algorithm parameters [5]. Two problems we study in this assignment have been added to IOHprofiler's built-in benchmark problems, denoted as F18 and F19 of the PBO class. With the help of the IOHprofiler, it is easy to build pipelines that automatically evaluate evolutionary algorithms' performance. We first record experimental logs via the IOHexperimenter module. After that, we upload logs to a free server running the IOHanalyzer service to obtain the statistical summary as well as results presented by images. All of results in this report are outputs of the IOHanalyzer.

The rest of this report is organized as follows. Section 2 describes the genetic algorithm and the evolution strategy in detail. It also provides information about hyperparameter settings. We report the metrics and experimental results in Section 3. This report ends with a brief conclusion and discussion.

2 Algorithms

2.1 Genetic Algorithm

2.1.1 Overview

The genetic algorithm is designed for the discrete search space, which usually appears in problems related to combinatorial optimization. The binary encoding is a common choice, i.e., $\mathbf{x} \in \{0,1\}^n$. Given an objective function F, the outline of the genetic algorithm used in this project is shown in Algorithm 1.

Algorithm 1: Overview of Genetic Algorithm

Input : Budget B,
Problem dimension n,
Population size μ,
Mutation rate p_m,
Update ratio θ,
Objective function F,
Tournament size t_k,
Tournament probability p_k.

Termination : The algorithm terminates when $B \leq 0$.

Initialize the population $\mathcal{D}^{(0)} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_\mu\}$ by random sampling, where $\mathbf{x}_i = [x_1, x_2, \ldots, x_n]^T, i = 1, \ldots, \mu$ and $x_j \sim$ iid `Bernoulli` $(0.5), j = 1, \ldots, n$;
`Evaluation(`$\mathcal{D}^{(0)}$, F, B`)`;
B_{copy} = B;
while $B > 0$ **do**
  updateNumber = $\mu \cdot \theta$;
  `/* Update the population partially. */`
  while *updateNumber* > 0 **do**
    p_1, p_2 = `Selection(`$\mathcal{D}^{(t)}$, B, B_{copy}, t_k, p_k`)`;
    c = `Crossover(`p_1, p_2`)`;
    `Evaluation(`c, F, B`)`;
    if *c.fitness* > $\mathbf{x}_1$.*fitness* **then**
      Delete $\mathbf{x}_1$ from $\mathcal{D}^{(t)}$;
      $\mathcal{D}^{(t)}$ = $\mathcal{D}^{(t)} \cup c$;
      Sort $\mathcal{D}^{(t)}$ in ascending order according to fitness values;
    end
  end
  `Mutation(`$\mathcal{D}^{(t)}$, p_m`)`;
  `Evaluation(`$\mathcal{D}^{(t)}$, F, B`)`;
  $\mathcal{D}^{(t+1)} = \mathcal{D}^{(t)}$
end
Function `Evaluation(`$\mathcal{D}$, F, B`)`**:**
  for $i = 1$ **to** $\mathcal{D}$.*size* **do**
    $\mathbf{x}_i$.fitness = $F(\mathbf{x}_i)$;
    B = B - 1;
  end
  Sort $\mathcal{D}$ in ascending order according to fitness values;

In the vanilla genetic algorithm, offspring are generated by applying the selection, crossover, and mutation operators sequentially at time t. Then, the population at time $t+1$ is a copy of these offspring. However, we found that updating the whole population at each time step often leads to an unstable performance in pilot experiments. The reason is that the overall population quality may decrease during the optimization since there is no guarantee that offspring have higher fitness values than parents. A decline in the population quality caused by offspring generation is not a big problem with a sufficient budget and a relatively small problem dimension. The decrease in quality at the early stage usually means a broader exploration area in search space and can be compensated easily later. But given $B = 5,000$

and $n = 50$, the convergence rate of the vanilla genetic algorithm is too slow to find a satisifying solution when the budget runs out. To overcome this weakness, we improve the vanilla genetic algorithm in several aspects:

- Introduce a hyperparameter update ratio θ to control the proportion of updating population at each time step. $\theta = 1$ corresponds to the vanilla genetic algorithm, while $\theta = \frac{1}{\mu}$ means only update one individual.
- Maintain the ascending order of the population based on the fitness value. An offspring will only be by added to the population if its fitness value is better than the first individual's.
- Design a hybrid selection strategy that will be described in Section 2.1.2. Outputs of the selection operator are two individuals set as parents.
- Use an inner loop to generate the offspring one by one until the update ratio is satisfied at each time step, aiming at saving budget and encouraging diversity.

The following three sections introduce selection, crossover, and mutation operators used in Algorithm 1 in detail.

2.1.2 Selection

There are various selection strategies for the genetic algorithm. The different strategies put different selection pressures on the population during optimization. Here, we mainly consider three representative selection strategies and develop a hybrid selection operator. Algorithm 2 illustrates the idea of combining different selection strategies.

Algorithm 2: Genetic Algorithm: Hybrid Selection Operator

Function `Selection(`$\mathcal{D}$, B, $B_{copy}i$, t_k, p_k`):`
  $B_{current}$ = $B_{copy} - B$;
  if $B_{current} < (0.4 * B_{copy})$ **then**
    `ProportionalSelection(`$\mathcal{D}$`);` `// Early stage: proportional selection.`
  else if $B_{current} < (0.6 * B_{copy})$ **then**
    `RankSelection(`$\mathcal{D}$`);` `// Middle stage: rank selection.`
  else
    `TournamentSelection(`$\mathcal{D}$, t_k, p_k`);` `// Late stage: tournament selection.`
  end
  Sample two individuals p_1, p_2 without replacement from $\mathcal{D}$ by simple random sampling, where $P(\mathbf{x} \text{ is selected}) = \mathbf{x}$.p_selection;
  return p_1, p_2

Function `ProportionalSelection(`$\mathcal{D}$`):`
  sum = 0;
  foreach $\mathbf{x} \in \mathcal{D}$ **do**
    sum = sum + $\mathbf{x}$.fitness;
  end
  foreach $\mathbf{x} \in \mathcal{D}$ **do**
    $\mathbf{x}$.p_selection = $\mathbf{x}$.fitness / sum;
  end

Function `RankSelection(`$\mathcal{D}$`):`
  `/* `$\mathcal{D}$` is sorted based on fitness. */`
  for $i = 1$ **to** $\mathcal{D}$.*size* **do**
    $\mathbf{x}_i$.p_selection = $(2 * i)/(\mathcal{D}.\text{size} * (\mathcal{D}.\text{size} + 1))$;
  end

Function `TournamentSelection(`$\mathcal{D}$, t_k, p_k`):`
  Sample t_k individuals from $\mathcal{D}$ to constitute $\mathcal{D}_{tour}$. For other non-selected individuals, $\mathbf{x}$.p_selection = 0;
  Sort $\mathcal{D}_{tour}$ in descending order according to fitness values;
  for $i = 1$ **to** $\mathcal{D}_{tour}$.*size* **do**
    $\mathbf{x}_i$.p_selection = $p_k(1 - p_k)^i$;
  end
  Normalize the selection probabilities;

Proportional selection assigns a selection probability to each individual that is proportional to the fitness. The intuition behind this strategy is the survival of the fittest. However, the proportional selection is likely affected by significant

variations in F, which may lead to the local optimum trap. Rank selection is a selection strategy that is more robust to the variance. It decides selection probabilities based on a function of the ranking. Here, we adopt the ranking function suggested by [6]:

$$P\left(\mathbf{x}_i \text{ is selected}\right) = \frac{2r_i}{\mu(\mu+1)}.$$

where r_i is the rank of the individual x_i based on its fitness value in ascending order, and μ is the population size. Tournament selection is another commonly used strategy. Given a pre-defined tournament size t_k and the tournament probability p_k, it firstly samples t_k individuals from the population. Then, the selected individuals compete for the parent qualification following the Geometric distribution with the success probability equal to p_k. In other words, the i-th fittest individual is assigned the selection probability $p_k\left(1-p_k\right)^i$. We apply the normalization to all individuals at the end of the tournament to ensure the sum of selection probabilities equals 1.

The selection pressure of proportional selection, rank selection, and tournament selection increases sequentially. High selection pressure encourages exploitation and reduces exploration, and vice versa. Therefore, we divide the optimization into three stages to balance between exploitation and exploration. The hybrid selection operator can be summarized as follows:

1. Early stage: current budget $>0.6\times$ total budget. Use the proportional selection to explore the search space.
2. Middle stage: $0.4\times$ total budget $\leq$ current budget $<0.6\times$ total budget. Use the rank selection to shift from exploration to exploitation.
3. Late stage: current budget $\leq 0.4\times$ total budget. Use the tournament selection to exploit current optimal solutions.

2.1.3 Crossover

We choose the one-point crossover operator in the genetic algorithm. Algorithm 3 and Figure 2 show the logic of this operator.

Algorithm 3: Genetic Algorithm: One-point Crossover Operator

Function `Crossover`(p_1, p_2)**:**
  Generate a random integer $m \in \{1, 2, \ldots, p_1.\text{length}\}$;
  $c = p_1[:m] + p_2[m+1:]$;
  return c

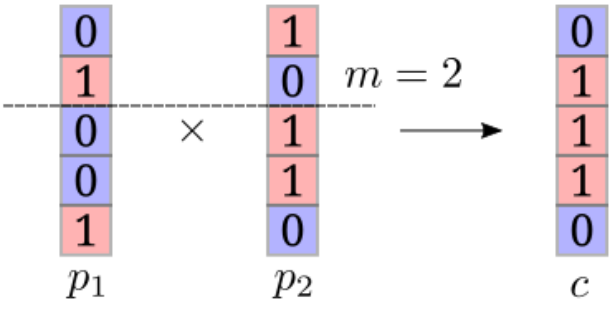


Figure 2: One-point crossover operator. Assume $m = 2$.

Inputs of crossover operator are two parents $p_1 = [x_1, x_2, \ldots, x_n]^T$ and $p_2 = [y_1, y_2, \ldots, y_n]^T$. The crossover point is generated by sampling an integer $m \in \{1, 2, \ldots, n\}$. The child c is produced by swapping bits between parents based on the crossover point, that is, $c = [x_1, \ldots, x_m, y_{m+1}, \ldots, y_n]^T$. Traditionally, the crossover operator will output two children. However, we only retain the first child because we update the population partially in the genetic algorithm. Redundant offspring will waste valuable budgets.

2.1.4 Mutation

The mutation operator introduces variations into solutions to broaden the search area in the feasible region. We adopt the bit flip mutation operator because of the usage of binary encoding. The bit flip mutation operator is illustrated in Algorithm 4 and Figure 3.

Algorithm 4: Genetic Algorithm: Bit Flip Mutation Operator

Function `Mutation`($\mathcal{D}$, p_m)**:**
  foreach $\mathbf{x} \in \mathcal{D}$ **do**
    for *i = 1* **to** $\mathbf{x}$.*length* **do**
      Sample $d \sim \texttt{Uniform}(0, 1)$;
      if $d < p_m$ **then**
        $x_i = |x_i - 1|$;
      end
    end
  end

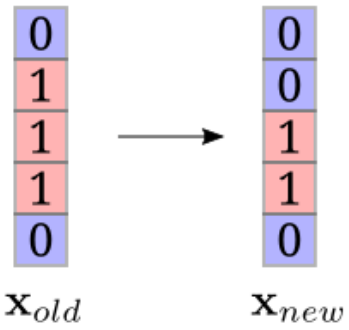


Figure 3: Bit flip mutation operator. Assume the second location of the individual is flipped.

The mutation operator is applied to the whole population. For each location of the individual, the bit flipping happens with probability p_m, i.e., $0 \rightarrow 1, 1 \rightarrow 0$. With mutation, we can increase the diversity of solutions.

2.1.5 Hyperparameter Settings

There are three major hyperparameters in the genetic algorithm — population size μ, mutation rate p_m, and update ratio θ. We perform hyperparameter fine-tuning on these parameters with the grid search method. The fine-tuning results are discussed in Section 3. As for other hyperparameters, we determine their values according to assignment requirements and the results of pilot experiments. Table 1 summarizes the hyperparameter settings of the genetic algorithm for reproducibility.

Table 1: Hyperparameter settings of the genetic algorithm.

Hyperparameter	**Meaning**	**Value**
B	budget	5,000
n	problem dimension	50
t_k	tournament size	Equal to the population size
p_k	tournament probability	0.8

We set the random seed to 42 when running the genetic algorithm.

2.2 Evolution Strategy

3 Experimental Results

3.1 Metrics

In order to evaluate the models' performance, we choose the following metrics:

Statistics. We report the mean, median, and max of the best-found fitness $f(x)$. Because the objective of both F18 and F19 is maximization, a larger statistic means the better performance.

ECDF curves. We plot the empirical cumulative distribution function of the running time (ECDF). The ECDF curve reflects the proportion of solutions at least as good as a given target ϕ_i within the budget b.

AUC. We calculate the area under the ECDF curve. AUC measures the overall performance of the algorithm. A higher AUC represents the algorithm performs better on exploration and converges faster.

ERT curves. The expected running time (ERT) curve is widely used in fixed-target analysis. If the algorithm successfully hits the given target ϕ_i, ERT measures the average time the algorithm needs to find the qualified solution. Otherwise, ERT equals the positive infinity.

We run each experiment 20 times and report the average values of metrics to alleviate the impact of randomness.

3.2 Genetic Algorithm

This section first discusses the hyperparameter settings based on fine-tuning results. After that, we report the best result of our implementation of the genetic algorithm with the recommended parameter settings.

We use the grid search method to find suitable parameter settings. If we consider all possible combinations of hyperparameters, the search space is so large that we do not have enough computational resources to exhaust all possibilities. Therefore, our approach is exploring one hyperparameter setting while fixing other parameters.

3.2.1 Fine-tuning: Population Size

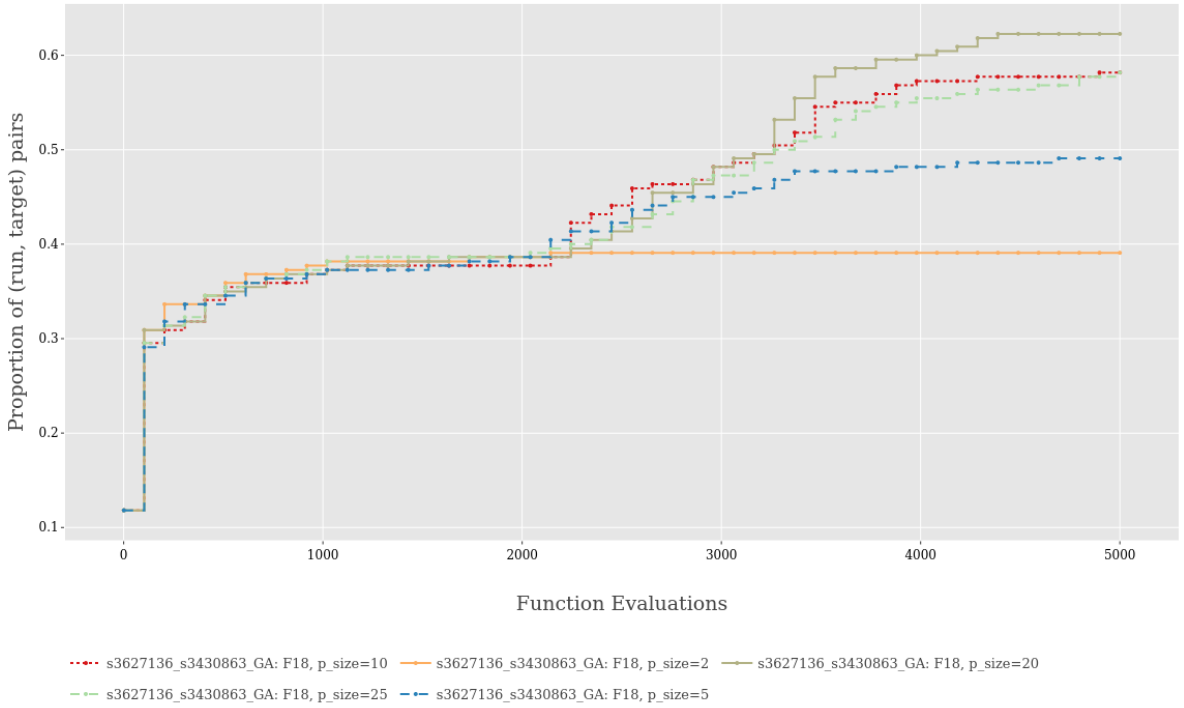
The population size decides the scale of solutions we keep track of at a specific moment. A large population size increases the diversity of solutions but decreases the number of updates because of the fixed budget. In experiments, we set the population size to 2, 5, 10, 20, and 25. Table 2 summarizes statistics and AUC values of F18 and F19.

Table 2: Summary of statistics and AUC in population size fine-tuning.

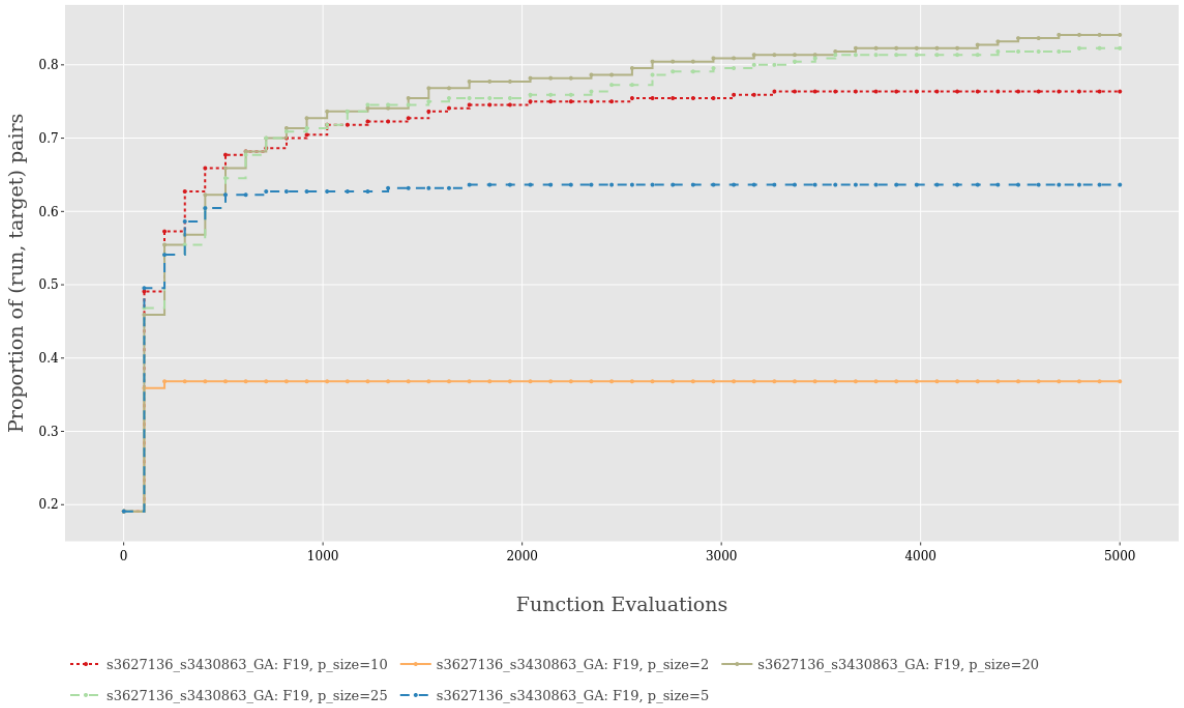
Population Size μ	**F18**				**F19**			
	Mean	Max	Median	AUC	Mean	Max	Median	AUC
2	2.62	3.12	2.62	0.379	31.1	38.0	30.0	0.366
5	3.23	3.71	3.28	0.419	39.8	46.0	41.0	0.623
10	3.70	**5.36**	3.71	0.451	44.1	48.0	44.0	0.727
20	**3.90**	4.72	**3.89**	**0.464**	**46.5**	48.0	**46.0**	**0.763**
25	3.72	4.45	3.71	0.446	46.0	**50.0**	**46.0**	0.750

Figure 4 and Figure 5 show ECDF and ERT curves, respectively.

According to Table 2, setting the population size to 20 achieves the best performance on the mean, median, and AUC in F18 and F19. Although it does not perform best on the max value, its overall performance exceeds other settings. ECDF curves and ERT curves support the superiority of $\mu = 20$. ECDF curves of two problems show that the small population size, like 2 or 5, prevents the algorithm from exploring a broader area in the search space. We can see that the algorithm tends to get stuck in the local optimum after the first few epochs. ERT curves also support the argument that a small population requires more budgets to reach the same level of targets. However, a larger population does not equal good performance. A population greater than 20 actually slows the convergence rate, which can be concluded from ECDF and ERT curves. Therefore, our recommendation setting of the population size is 20 for both problems.

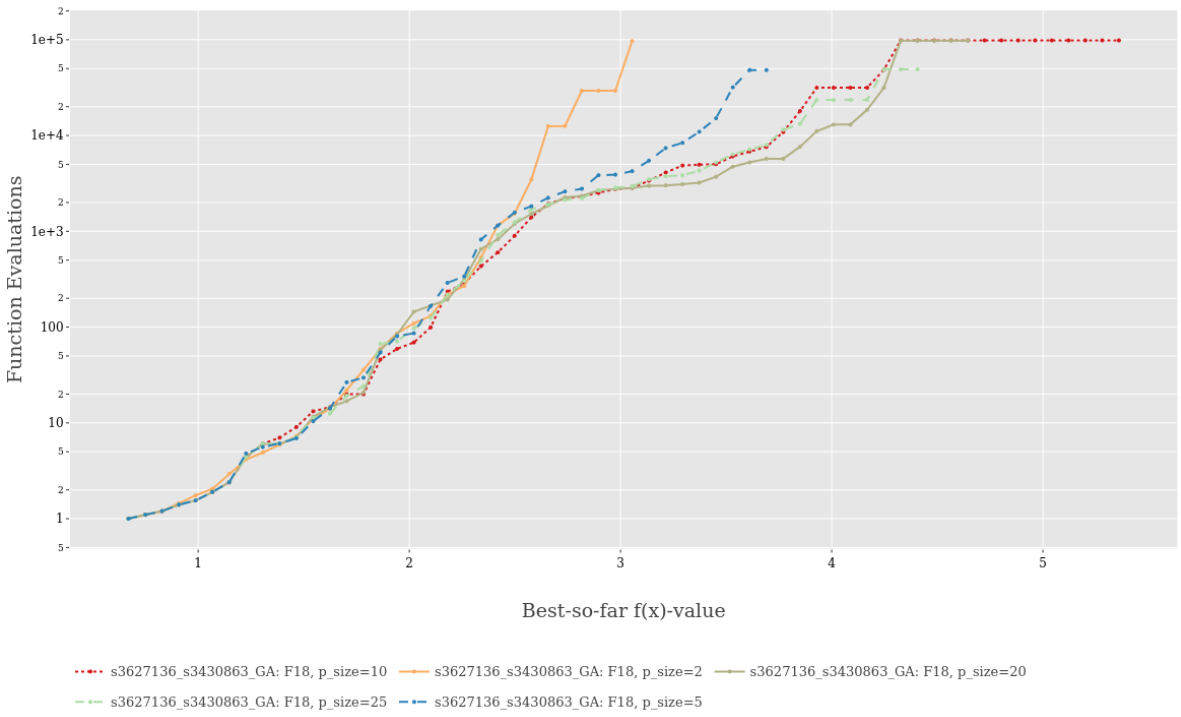


(a) F18 population size fine-tuning: ECDF curves. $f_{\min} = 0.67, f_{\max} = 5.36, \Delta f = 0.52$.

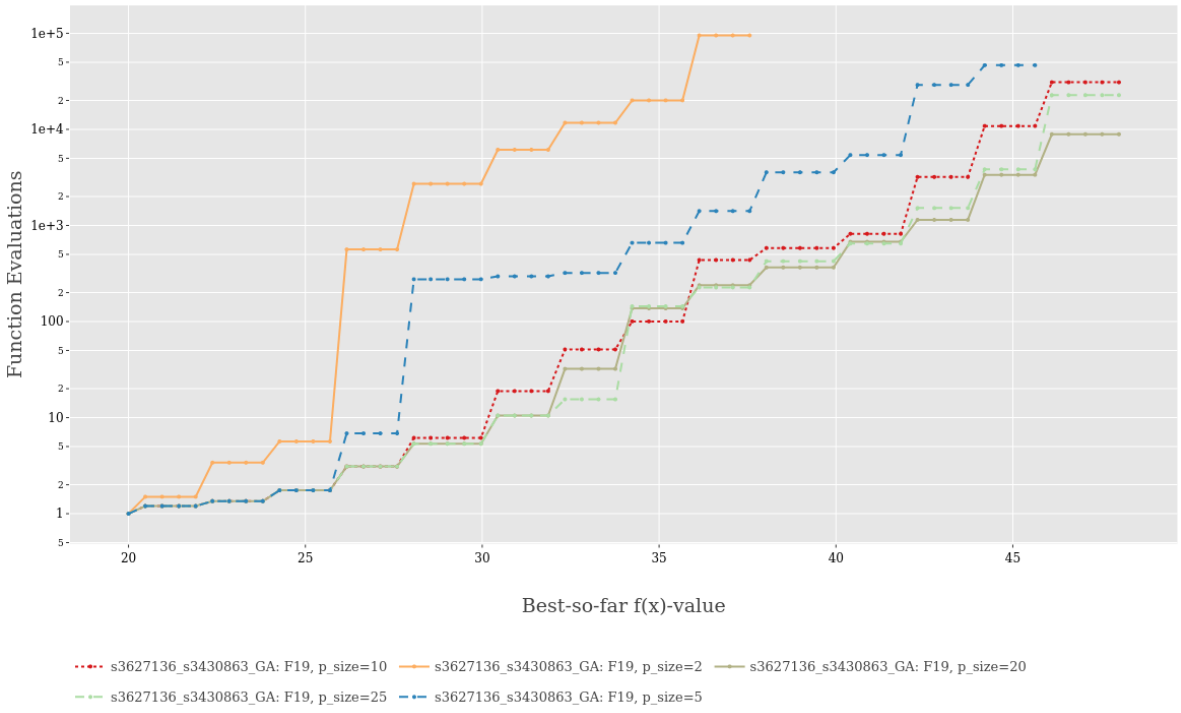


(b) F19 population size fine-tuning: ECDF curves. $f_{\min} = 20, f_{\max} = 48, \Delta f = 3.11$.

Figure 4: ECDF curves of the genetic algorithm with various population size settings. Figures are downloaded from the IOHanalyzer.



(a) F18 population size fine-tuning: ERT curves. We scale y axis $\log_{10}$.



(b) F19 population size fine-tuning: ERT curves. We scale y axis $\log_{10}$.

Figure 5: ERT curves of the genetic algorithm with various population size settings. Figures are downloaded from the IOHanalyzer.

3.2.2 Fine-tuning: Mutation Rate

The mutation rate controls the degree of variations the genetic algorithm introduces in each time step. According to the algorithm description in Section 2.1.4, the bit flip mutation operator accepts a constant rate at each time step. However, we think an adaptive mutation rate may be more suitable in some scenarios. Our attempts to search the appropriate mutation rate are twofold. One is trying some constant mutation rates (0.1, 0.05, $1/n = 0.02$), and the other uses a piecewise function to adjust the mutation rate over time. Based on pilot experiments, we design a piecewise function given the budget $B = 5,000$ as follows:

$$p_m = \begin{cases} 0.02, & B < 3000, \\ 0.1, & \text{Otherwise.} \end{cases} \tag{1}$$

The intuition behind Equation 1 is the variation brought by the mutation should be smaller in the late stage to encourage the exploitation behavior. We refer to the usage of Equation 1 as '`piecewise`' in tables and figures.

Table 3 represents the results of the mutation fine-tuning in F18 and F19. ECDF and ERT curves are shown in Figure 6 and Figure 7.

Table 3: Summary of statistics and AUC in mutation rate fine-tuning.

Mutation Rate p_m	F18				F19			
	Mean	Max	Median	AUC	Mean	Max	Median	AUC
0.1	3.01	3.71	2.94	0.391	41.9	44.0	42.0	0.614
0.05	3.36	4.21	3.35	0.417	45.3	**48.0**	**46.0**	0.691
0.02	3.81	**5.36**	3.71	**0.496**	**46.5**	**48.0**	**46.0**	**0.763**
`piecewise`	**3.90**	4.72	**3.89**	0.464	45.0	**48.0**	**46.0**	0.669

In F18, setting $p_m = 0.02$ or using the piecewise function seems reasonable choices based on statistics and AUC values. The piecewise function performs best on the mean and median values, while $p_m = 0.02$ achieves a higher max value as well as the AUC value. However, ECDF curves suggest that the genetic algorithm is more robust with the piecewise function. Besides, ERT curves show that the piecewise function slightly accelerates the convergence rate. Hence, we conclude using the piecewise function defined by Equation 1 is the optimal setting of the mutation rate.

Unfortunately, the same conclusion does not hold in F19. In fact, $p_m = 0.02$ outperforms all the other settings on metrics listed in Table 3. ECDF and ERT curves show that $p_m = 0.02$ converges much faster than other mutation rates and achieves higher fitness values. We think the scenario defined in F19 prefers a conservative mutation rate, which means choosing $p_m = 1/n$ can guarantee a satisfying model performance. In our case, problem dimension n is 50 and p_m is 0.02.

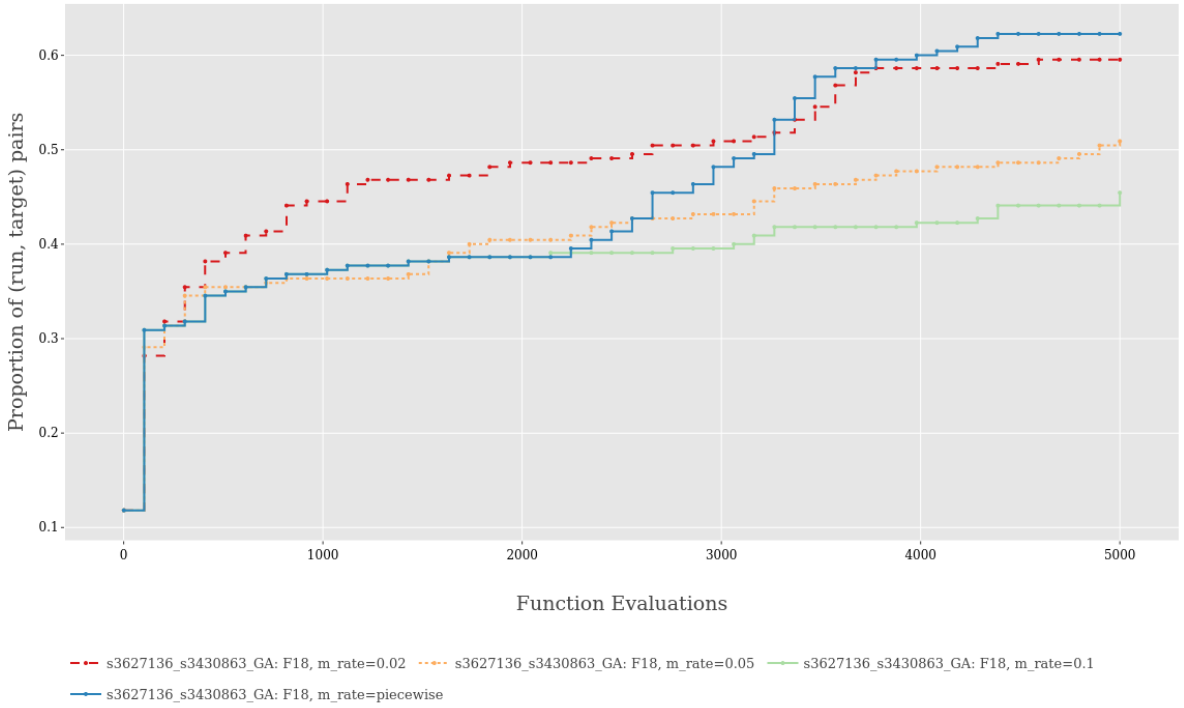
3.2.3 Fine-tuning: Update Ratio

Remember that our genetic algorithm implementation only partially updates the population at each time step, as mentioned in Section 2.1.1. We introduce a hyperparameter, update ratio θ, to control the intensity of the update. We set the update ratio to 0.2, 0.4, 0.8, and 1.0 to find a suitable number. Notice that $\theta = 1.0$ is equivalent to the vanilla genetic algorithm. Results of the update ratio fine-tuning are summarized in Table 4, Figure 8, and Figure 9.

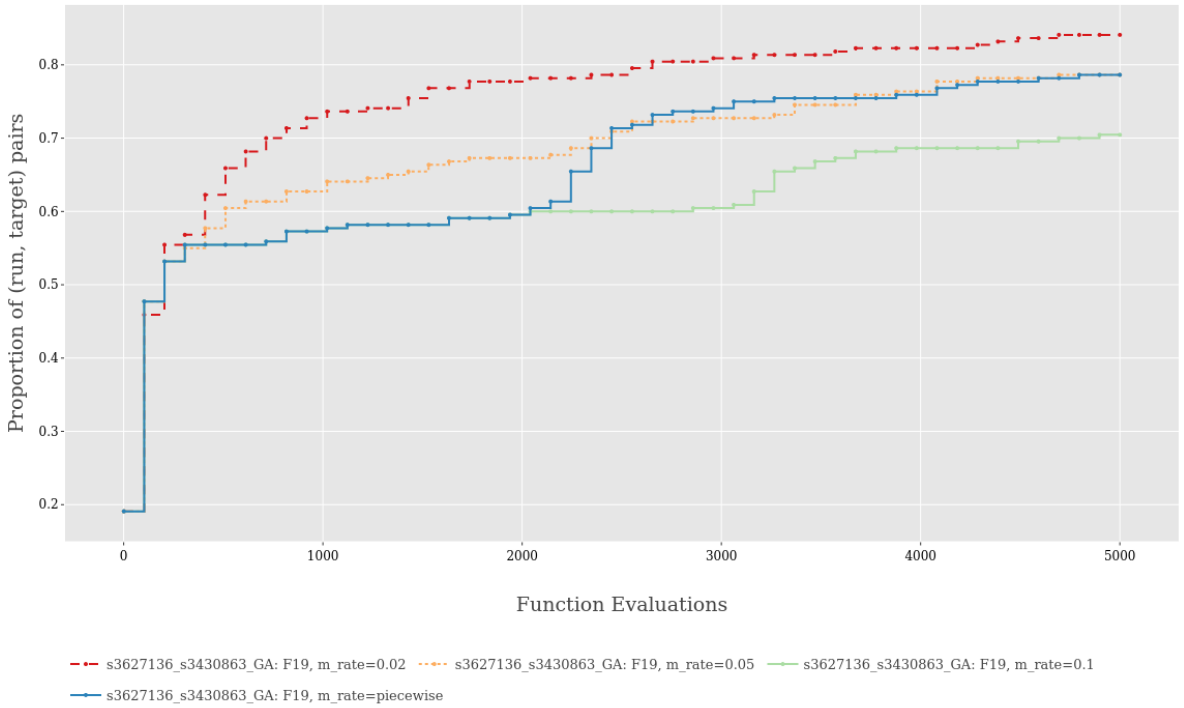
Table 4: Summary of statistics and AUC in update ratio fine-tuning.

Update Ratio θ	F18				F19			
	Mean	Max	Median	AUC	Mean	Max	Median	AUC
0.2	3.53	4.58	3.43	0.418	44.9	48.0	45.0	0.686
0.4	**3.90**	**4.72**	**3.89**	**0.464**	**46.5**	**48.0**	**46.0**	**0.763**
0.8	3.35	4.21	3.28	0.443	42.6	46.0	42.0	0.698
1.0	2.99	3.71	2.94	0.413	42.3	46.0	42.0	0.689

Results validate that the update ratio is an important influencing factor of algorithm performance. Using the setting $\theta = 0.4$ obtains the best performance on all metrics in both F18 and F19. ECDF and ERT curves also support

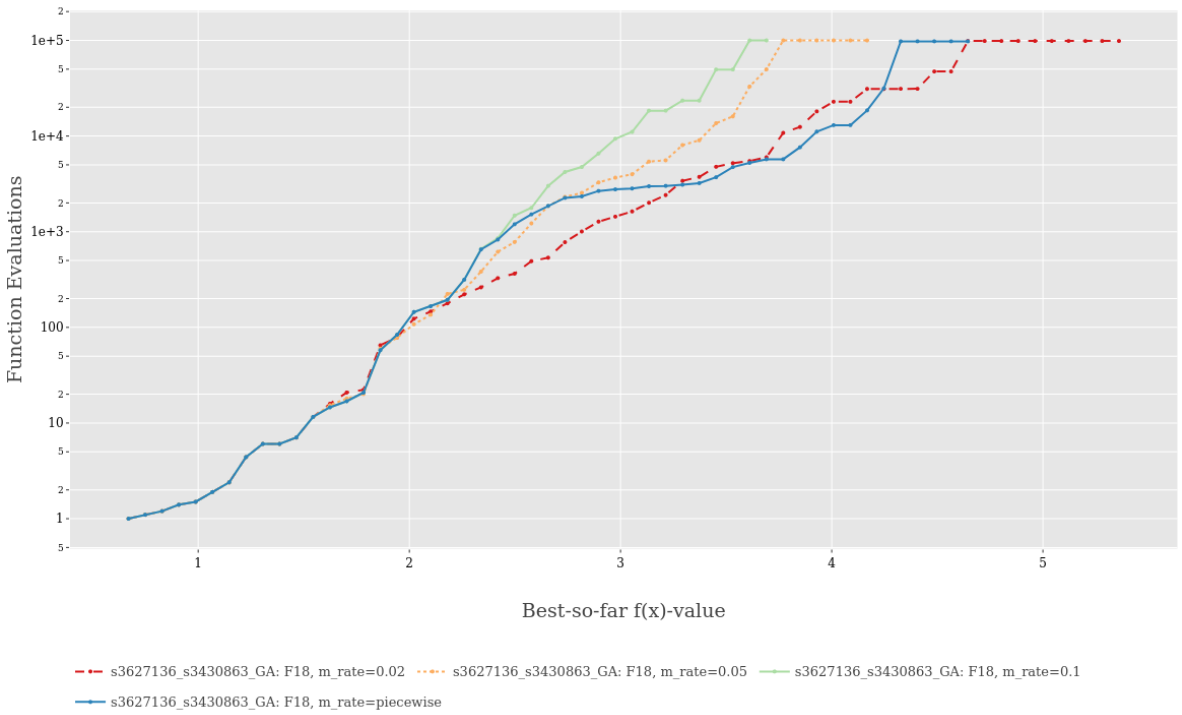


(a) F18 mutation rate fine-tuning: ECDF curves. $f_{\min} = 0.67, f_{\max} = 5.36, \Delta f = 0.52$.

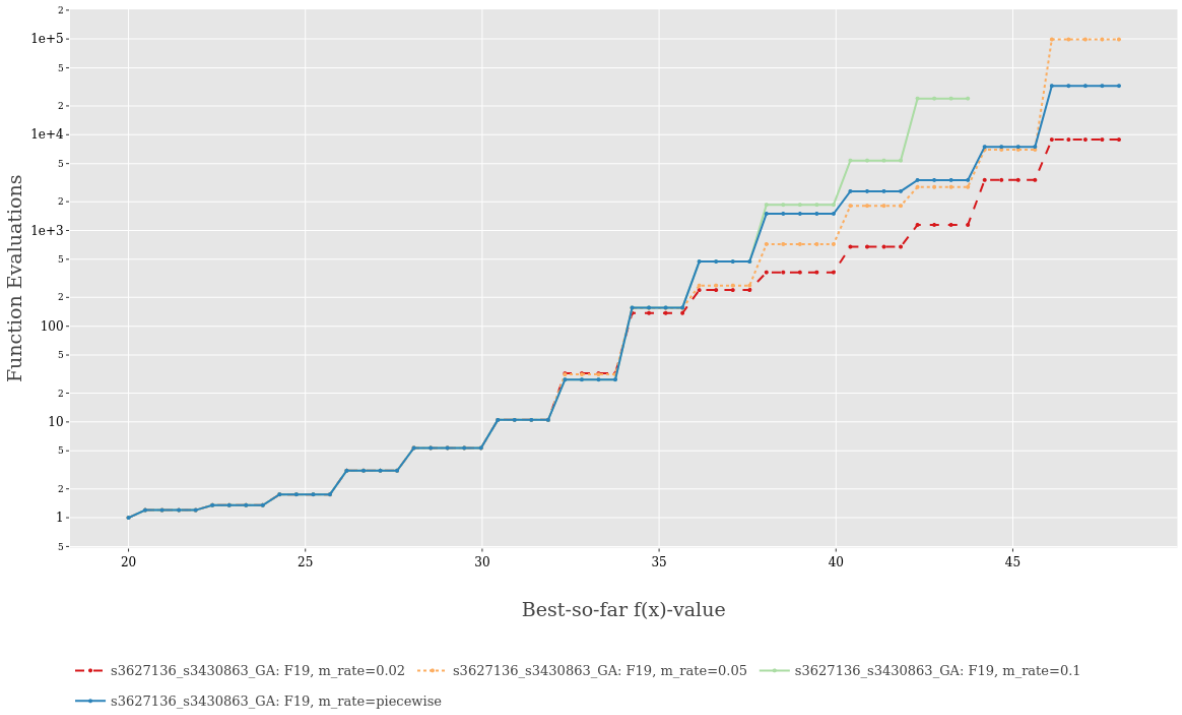


(b) F19 mutation rate fine-tuning: ECDF curves. $f_{\min} = 20, f_{\max} = 48, \Delta f = 3.11$.

Figure 6: ECDF curves of the genetic algorithm with various mutation rate settings. Figures are downloaded from the IOHanalyzer.

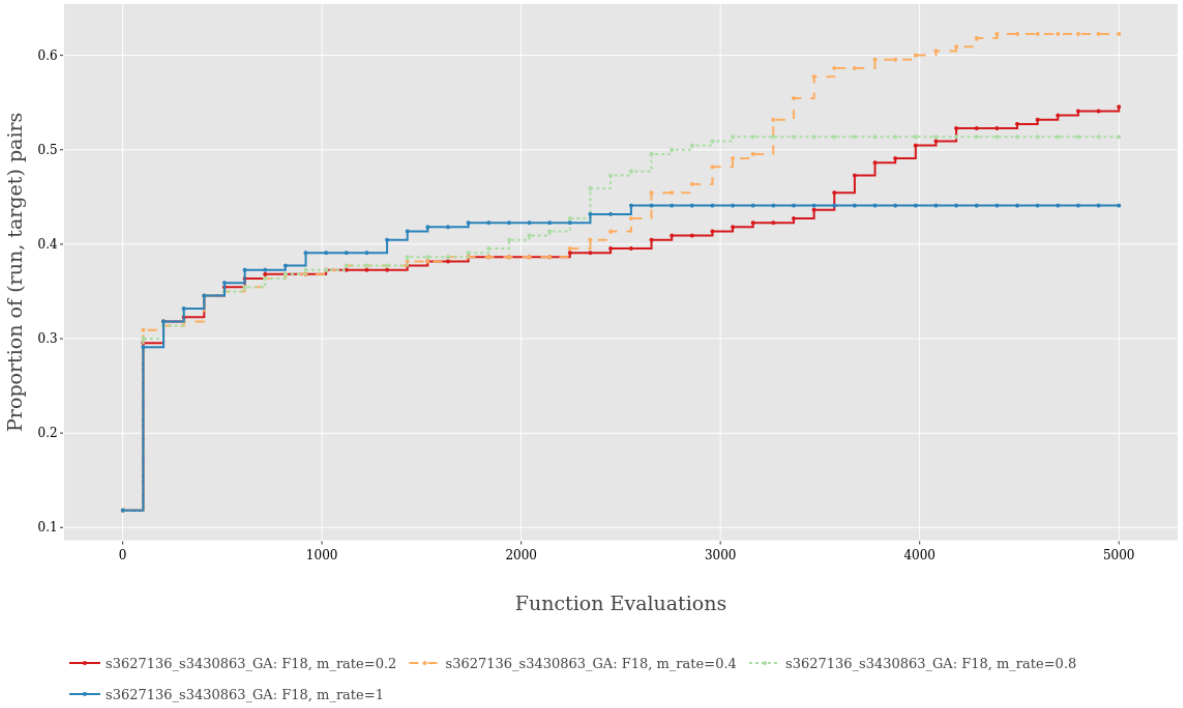


(a) F18 mutation rate fine-tuning: ERT curves. We scale y axis $\log_{10}$.

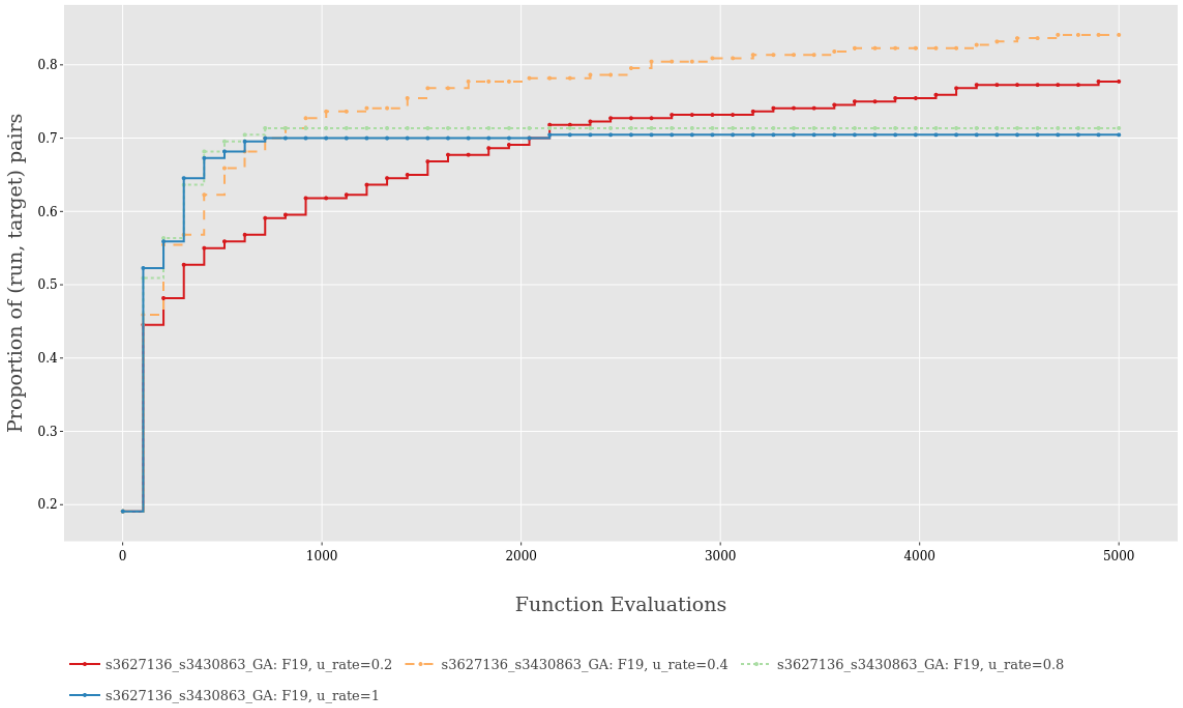


(b) F19 mutation rate fine-tuning: ERT curves. We scale y axis $\log_{10}$.

Figure 7: ERT curves of the genetic algorithm with various mutation rate settings. Figures are downloaded from the IOHanalyzer.

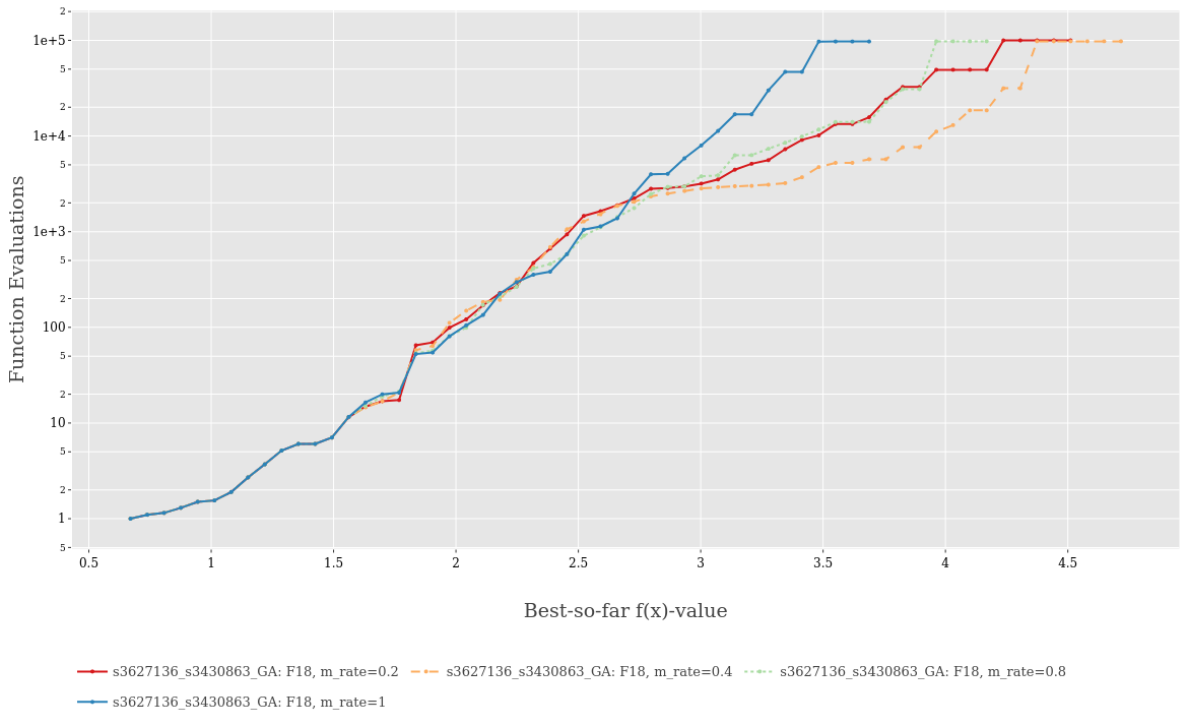


(a) F18 update ratio fine-tuning: ECDF curves. $f_{\min} = 0.67, f_{\max} = 5.36, \Delta f = 0.52$.

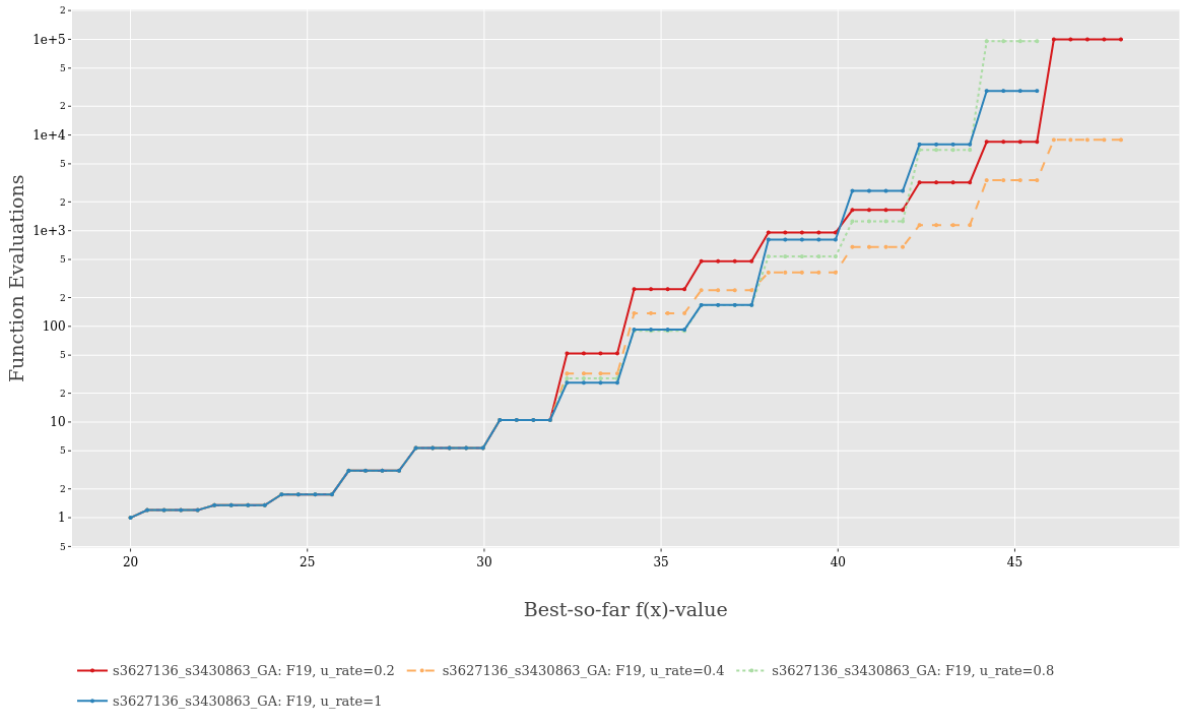


(b) F19 update ratio fine-tuning: ECDF curves. $f_{\min} = 20, f_{\max} = 48, \Delta f = 3.11$.

Figure 8: ECDF curves of the genetic algorithm with various update ratio settings. Figures are downloaded from the IOHanalyzer.



(a) F18 update ratio fine-tuning: ERT curves. We scale y axis $\log_{10}$.



(b) F19 update ratio fine-tuning: ERT curves. We scale y axis $\log_{10}$.

Figure 9: ERT curves of the genetic algorithm with various update ratio settings. Figures are downloaded from the IOHanalyzer.

the observation that updating nearly half of the population is an effective strategy. It is worth mentioning that the vanilla genetic algorithm ($\theta = 1.0$) actually performs poorly on both problems. It is consistent with our argument in Section 2.1.1 that the budget $B = 5,000$ is insufficient for the vanilla genetic algorithm to converge under the problem dimension $n = 50$.

3.2.4 Final Results

We run the genetic algorithm 20 times on F18 and F19 with the best hyperparameter settings we have found during the fine-tuning. We summarize the final results of our implementation of the genetic algorithm as follows:

- **F18**
  - Hyperparameter settings: random seed 42, budget $B = 5,000$, problem dimension $n = 50$, tournament size $t_k = 20$, tournament probability $p_k = 0.8$, population size $\mu = 20$, mutation rate using the piecewise function defined by Equation 1 (In our code implementation, just set the mutation rate to a negative float. The program will automatically switch to the piecewise function), update ratio $\theta = 0.4$.
  - Statistics: the averaged best-found fitness $f(x)$ = 3.90, the median of best-found fitness $f(x)$ = 3.89, the maximum of best-found fitness $f(x)$ = 4.72.
  - AUC: 0.464.
- **F19**
  - Hyperparameter settings: random seed 42, budget $B = 5,000$, problem dimension $n = 50$, tournament size $t_k = 20$, tournament probability $p_k = 0.8$, population size $\mu = 20$, mutation rate $p_m = 0.02$, update ratio $\theta = 0.4$.
  - Statistics: the averaged best-found fitness $f(x)$ = 46.5, the median of best-found fitness $f(x)$ = 46.0, the maximum of best-found fitness $f(x)$ = 48.0.
  - AUC: 0.763.

ECDF and ERT curves are the same as those in Figure 8 and Figure 9 with the update ratio $\theta = 0.4$. We do not plot them again to conserve space. All fine-tuning experiments and the final results of the genetic algorithm can be easily reproduced by running the source code file `s3627136_s3430863_GA.py` in the terminal, i.e., `python s3627136_s3430863_GA.py`.

3.3 Evolutionary Strategy

4 Discussion and Conclusion

References

[1] Adam Slowik and Halina Kwasnicka. Evolutionary algorithms and their applications to engineering problems. *Neural Comput. Appl.*, 32(16):1236312379, aug 2020.

[2] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer Publishing Company, Incorporated, 2nd edition, 2016.

[3] Tom Packebusch and Stephan Mertens. Low autocorrelation binary sequences. *Journal of Physics A: Mathematical and Theoretical*, 49(16):165001, March 2016.

[4] Patrick Briest, Dimo Brockhoff, Bastian Degener, Matthias Englert, Christian Gunia, Oliver Heering, Thomas Jansen, Michael Leifhelm, Kai Plociennik, Heiko Röglin, Andrea Schweer, Dirk Sudholt, Stefan Tannenbaum, and Ingo Wegener. The ising model: Simple evolutionary algorithms as adaptation schemes. In Xin Yao, Edmund K. Burke, José A. Lozano, Jim Smith, Juan Julián Merelo-Guervós, John A. Bullinaria, Jonathan E. Rowe, Peter Tiňo, Ata Kabán, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VIII*, pages 31–40, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[5] Carola Doerr, Hao Wang, Furong Ye, Sander van Rijn, and Thomas Bäck. Iohprofiler: A benchmarking and profiling tool for iterative optimization heuristics. *CoRR*, abs/1810.05281, 2018.

[6] Geof H. Givens and Jennifer A. Hoeting. *Computational Statistics*, chapter 3, pages 59–95. John Wiley & Sons, Ltd, 2012.