NaCo 22/23 assignment 1 report, group 11

Wouter Ykema, Bart Aaldering, Yvo Hu

Leiden Institute of Advanced Computer Science, The Netherlands

Abstract. We propose a genetic algorithm for the `OneMax`- and the `LeadingOnes`-problem. After describing the problem, a general overview of genetic algorithms is given, including some common operators for selection, mutation and recombination. Our algorithm, consisting of truncation selection, flip-bit mutation, exponential parent selection and uniform recombination, uses on average 9375 function evaluations for the `LeadingOnes`-problem and 1758 for the `OneMax`-problem. From testing using the IOHexperimenter we concluded that swap mutation performs extremely badly, and that roulette selection performs worse than truncation selection on these problems. We also do a literature review on Farahani, R.Z., Elahipanah, M.'s genetic algorithm in practice. Finally, in the appendix there is another algorithm we created that performs better than our genetic algorithm.

1 Introduction and problem description

In this paper, we propose a genetic algorithm for two bitstring matching problems: the `OneMax`-problem and the `LeadingOnes`-problem. The common goal of these problems is to find the unknown target bitstring, but the only way to gather information is through a fitness function.

The `OneMax`-function simply counts how many single bit matches there are, while the `LeadingOnes`-function has an intrinsic order of the bits and only counts how many bits match before the first non-matching bit.

The easiest way to visualize this, is using a black and white image. See figure 1. The black squares represent 1's and the white squares represent 0's. The bitstring should be read in the same order as how you would read an English book.

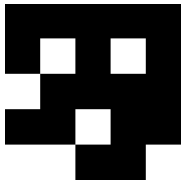

Fig. 1: 5×5 image representing a bitstring of 25 bits.

For this example, assume that the target bitstring corresponds to a completely black image, thus containing all 1's. Then `OneMax` will output a value of 18, since that is how many black squares there are. However, `LeadingOnes` will output a value of 6, because the 7th bit is the first non-matching bit.

There is actually one more complication. The output of the fitness function will be transformed by a random order-preserving function, so that the information can exclusively be used to compare the fitness of different bitstrings to each other.

Genetic Algorithm A genetic algorithm uses the concept of a *population* to solve the problem. At each iteration, the fitness function is evaluated on each *individual* of the population. In our case, the individuals are bitstrings, and the fitness function is either the `OneMax` or `LeadingOnes` version. In general, on each iteration, the following steps are performed.

step 1. Create the next generation using the concept of *mating selection*. This consists of selecting parents out of the population and performing recombination on them to create offspring. The amount of offspring is usually larger than the original population size.

step 2. Apply *variation* to each of the individuals. Small changes to the individuals increase the diversity of the population.

step 3. Evaluate the fitness function on each individual and store the resulting fitness values.

step 4. Apply the concept of *environmental selection* according to the fitness values, similar to how *natural selection* works in nature. The amount of individuals to keep is equal to the original population size.

step 5. Stop when the *termination criteria* are met. Otherwise, go back to step 1.

The termination criteria can include a fixed *budget* of allowed function evaluations, or some fixed *target* fitness to be reached.

Notation Let M denote the original population size, and let N be the amount of offspring to create during mating selection. Let n be the dimensionality of the search space, which is the bitstring-length. Each bitstring $\boldsymbol{x}$ will be denoted by $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$, where each $x_k \in \{0, 1\}$ for $k \in \{1, \ldots, n\}$. Let $f : \mathbb{Z}^n \to \mathbb{R}$ be the fitness function as explained in section 1, sending all bitstrings to real numbers measuring how close they are to the target.

2 Common operators

2.1 Selection

The selection operator is used to choose the parents on which the new generation will be based. There are many different selection methods with different strengths and weaknesses.

Truncation selection Truncation selection works by sorting all individuals based on their fitness and then choosing the best $truncation\_size$ individuals where

$$truncation\_size = population\_size \times selection\_proportion.$$

Roulette selection In roulette selection the fitness is used to create a probability that an individual is selected. The formula for this probability is $p_x = f_x / \sum_{j=1}^{M} f_j$ where p_x is the probability that individual x is selected, M is the total number of individuals in the population and f_x is the fitness of x. This method can be seen as a roulette wheel where each individual gets a chunk of the wheel based on their fitness. The fitness values are assumed to be positive.

Comparison: Truncation selection is one of the most straightforward methods of selection making it easy to implement. Roulette selection is a bit less straightforward as it needs random numbers and uses the proportion of the total fitness. In roulette selection however it is not the case that individuals with low fitness can't be selected. This can be a good thing because there is a chance that individuals with lower fitness have features or characteristics that when combined or

mutated will result in a new best individual. In general this makes truncation better for simple implementation or situations where high selection pressure is needed and roulette selection where these are not the case.

Alternative: Another way to perform selection is sampling from an *exponential* distribution $\mathcal{E}(s)$ with mean s and use the results as indices k to the sorted list of M individuals from highest to lowest fitness. There are two easily fixable problems with this approach. As an index, k must be an integer, so the value must be converted to an integer. Furthermore, k must be between the bounds of the list, so wrap it around using the modulo operator. The distribution of indices k is then

$$\mathcal{E}(s, M) = (\lfloor \mathcal{E}(s) \rfloor \mod M).$$

This has the effect of superimposing the tail of the exponential distribution onto the distribution itself, infinitely many times, but this does not really matter, since the probability of getting very large numbers is very small. Also notice that just like with roulette selection, the same individual can be chosen multiple times.

2.2 Recombination

Recombination (also called crossover) is a genetic operator that combines the chromosomes of two previous individuals (the parents) to form the individuals for the new population. It is used in GA's because two individuals can both hold a part of a good solution and combining can create this solution. This way the GA explores the search space.

One-point crossover Assuming that the chromosomes are represented as an array, one-point crossover works as follows. From the elements in the array a random crossover point is chosen. The two parents will swap all elements to the right of this point resulting in two offspring for the new population. Some versions of one-point crossover only use one of the two.

Uniform crossover Uniform crossover does not crossover chunks of the chromosome but instead works by combining individual elements of the parents chromosomes. For each element in the chromosome it picks one of the parents at random and chooses his element. The picking of the parents can be done fifty-fifty or with a bias toward one of the parents.

For each set of 2 parents $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ and $\boldsymbol{y} = (y_1, y_2, \ldots, y_n)$, generate a random bitstring $\boldsymbol{r} = (r_1, r_2, \ldots, r_n)$ with the discrete uniform distribution $\mathcal{U}\{\mathbf{0}, \mathbf{1}\}$.

The bits of the child $\boldsymbol{z} = (z_1, z_2, \ldots, z_n)$ are calculated as

$$z_k = \begin{cases} x_k \text{ if } r_k = 1 \\ y_k \text{ if } r_k = 0 \end{cases},$$

for all $k \in \{1, \ldots, n\}$, so that each bit is inherited from randomly one of the two parents.

Comparison: One-point crossover and uniform crossover will in general produce similar result. Uniform crossover can however be a bit more computationally expensive, because it needs to pick a parent for each element in the chromosome compared to one crossover point. This will be more the case the larger the chromosome size.

2.3 Mutation

The mutation operator is used to create small changes in individuals. This way a GA can search the local search space around an individual, trying to improve it.

Flip-bit mutation Flip-bit mutation works by randomly flipping elements of the chromosome of an individual. For this to work the GA has to be binary encoded. The frequency of these flips is determined by the mutation rate.

The *mutation rate* is an important parameter, denoted as a real number γ between 0 and 1. It is a measure for the amount of mutation that should happen.

The first step of creating a mutation vector is to generate a list $L \in [0,1]^n$ of n real numbers with the uniform distribution $\mathcal{U}(\mathbf{0}, \mathbf{1})$.

Then perform

$$l \leftarrow \begin{cases} 1 \text{ if } l < \gamma \\ 0 \text{ if } l \geq \gamma \end{cases}$$

on each element $l \in L$. After that, L is a valid mutation vector. It has (on average) more 0's if γ is small, and more 1's if γ is large. To apply it to an individual, simply use *logical xor*.

Swap mutation Swap mutation works by selecting two elements of the chromosome of an individual and swapping them.

Comparison: Swap mutation can be useful in permutation based encoded GA's as it will not change what elements are in the chromosome, while flip-bit mutation will be useful when this does need to change.

3 Implementation

The genetic algorithm has been implemented in Python, in such a way that you can easily change parameters and check out different operators, all because of the modular design.

To avoid using a lot of `for`-loops, the implementation will make good use of the `numpy` math library. It basically simplifies any-dimensional arrays, and makes them as simple as just numbers.

Furthermore, it uses IOHExperimenter for the problem itself, and `shutil` for archiving the data about the performance of the algorithm.

```
import numpy as np # Math Library
import ioh # IOH Experimenter
import shutil # Shell Utilities
```

The GA is implemented as a `class` with a single variable, namely the `budget`. It has only one function[1], which should send a *problem* of type *ioh.problem.Integer* to an *ioh.IntegerSolution*. The fitness function is evaluated on the individuals by calling the *problem* object.

[1] Technically, an object that is an instance of the GA `class` is itself callable, because the 'function' is implemented by defining what `__call__` should do.

3.1 Generic Algorithm

Inside of the `__call__` function, the operators are defined first, and then the following very generic genetic algorithm runs, using these operators.

```
# Generate a random population of solutions
x = initialize_population()
y = [problem(p) for p in x]

# Compute next generations until the termination criteria are met
while not termination_criteria(problem.state):
    x = variation(mating_selection(x, y))
    y = [problem(p) for p in x]
    x, y = environmental_selection(x, y)

# Output the number of evaluations and return the best solution so far
print("Number of evaluations: ", problem.state.evaluations)
return problem.state.current_best
```

As you can see, the core functionality of the GA is so simple that it kind of looks like pseudo code at this point. This is because of the modularity. The operators can now be defined before the loop instead of inside the loop.

3.2 Definition of functions

The operators are defined using factory functions in Python. The following function will create the desired `variation` operator, for example.

```
def flipbit_mutation(gamma=0.1):
    def f(population):
        mut = np.random.random(population.shape) < gamma
        return np.logical_xor(population, mut).astype(int)
    return f
```

For different values of `gamma`, a different function will be returned. The returned function is the operator to be used in the algorithm.

The other functions can be seen in appendix B. In particular, the mating selection operator is interesting, because it takes two functions as arguments that define how to select parents and how to apply recombination.

4 Application of a GA in practice

In this section, we'll be discussing a paper called "A genetic algorithm to optimize the total cost and service level for just-in-time distribution in a supply chain " [1].

4.1 Field and context

This paper describes the application of a multi-objective GA to solve 2 problems in Supply chain management based on a three-echelon supply chain (Figure 2). The first objective is to minimize the total transportation, holding and purchasing costs, and the second one is to minimize the earliness and tardiness of deliveries based on just in time(JIT) delivery.

4.2 Why choose a GA for their application, were there any alternatives?

In the present mixed-integer linear programming model, increasing the size of the problem in polynomial order will lead to an exponential increase in the computational complexity. Therefore, a multi-objective genetic algorithm (GA) is designed to solve the large-size problems which does not suffer from such computational complexity. Its results for small-size problems are then compared with the results obtained from LINGO optimization software, which acts as a baseline, in order to verify the performance of the proposed GA.

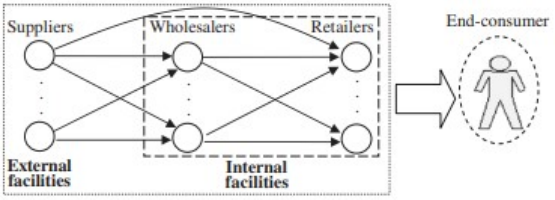

Fig. 2: Distribution network schema.

4.3 Was their approach successful?

The Pareto fronts of both the LINGO and GA solutions are illustrated in figure 3. The GA solution though it does not ever beat the LINGO baseline, it follows closely behind with a similar pattern. This chart displays the results based on a small-size sample, because a similar solution cannot be computed in a reasonable time with the LINGO method.

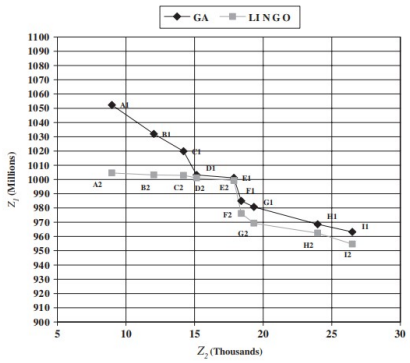

Fig. 3: An example for Pareto fronts of the solutions obtained from LINGO and GA. Z_1 and Z_2 are the objectives as described in section 4.1

We therefore consider the GA solution to be a satisfactory replacement for large-size samples if we take computational time constraints into account.

4.4 Our opinion on their approach

As was explained in section 4.2, mixed-integer linear programming is NP-Hard, so it can't be solved in polynomial time unless P=NP (which no one has been able to solve as of yet). It is therefore only logical to find a different approach to this problem, one which is computable in polynomial time. A GA is perfect to approximate the best solution to this problem, without being too computationally intensive.

4.5 Improve on their setup

The current GA (which may be found in the original paper, but is too large to show here) includes nearly

all aspects of a good GA. However, it lacks good heuristics to improve the process. We may employ a speciation heuristic to penalize cross-over between similar candidate solutions to prevent premature convergence to a less than optimal solution.

5 Analysis and results

To test the different operator variants we have implemented, we have run collect_data(100) for different operator variant configurations. The result of this can be seen in the figures in appendix A. We compared our initial operator variants to the new ones by changing one of the operators and collecting data. Our initial setup uses truncation selection to trim down the population (environmental selection), flip bit for mutation, uniformcrossover for recombination and the parents are chosen with exponential selection. This setup is called GeneticALgorithm_Optimal. We have also implemented and tried swapbit mutation but this performed so poorly it did not collect data. (ERT means Estimated Runtime)

6 Conclusion

What have we learned from this? We have learned how to create our own modular Genetic Algorithm consisting of different selection, recombination, and mutation operators, and are able to swap out operator variants with each other. By implementing these operators ourselves, we have gained an in depth understanding of the processes behind a GA.

Evaluation of the results suggests that the different combinations of operator variants are all very similar and that none really sticks out. Their performances differ based on the type of problem. One combination may perform very poor on one problem, and be the very best on another problem. Nonetheless, there is a clear and general pattern on the performance. The amount of optimization gained diminishes logarithmically with each generation.
This corresponds with the fact that it is easier to mutate a bitstring which contains only bad bits, than a bitstring which is considered optimal (no mutation can improve the corresponding model).

References

1. Farahani, R.Z., Elahipanah, M.: A genetic algorithm to optimize the total cost and service level for just-in-time distribution in a supply chain. International Journal of Production Economics **111**(2), 229–243 (2008). https://doi.org/https://doi.org/10.1016/j.ijpe.2006.11.028, `https://www.sciencedirect.com/science/article/pii/S0925527307001831`, special Section on Sustainable Supply Chain

A Figures of the results

In this appendix, four results are shown:

1. fixed-target performance on the Leading Ones problem,
2. fixed-target performance on the One Max problem,
3. fixed-budget performance on the Leading Ones problem, and

4. fixed-budget performance on the One Max problem.

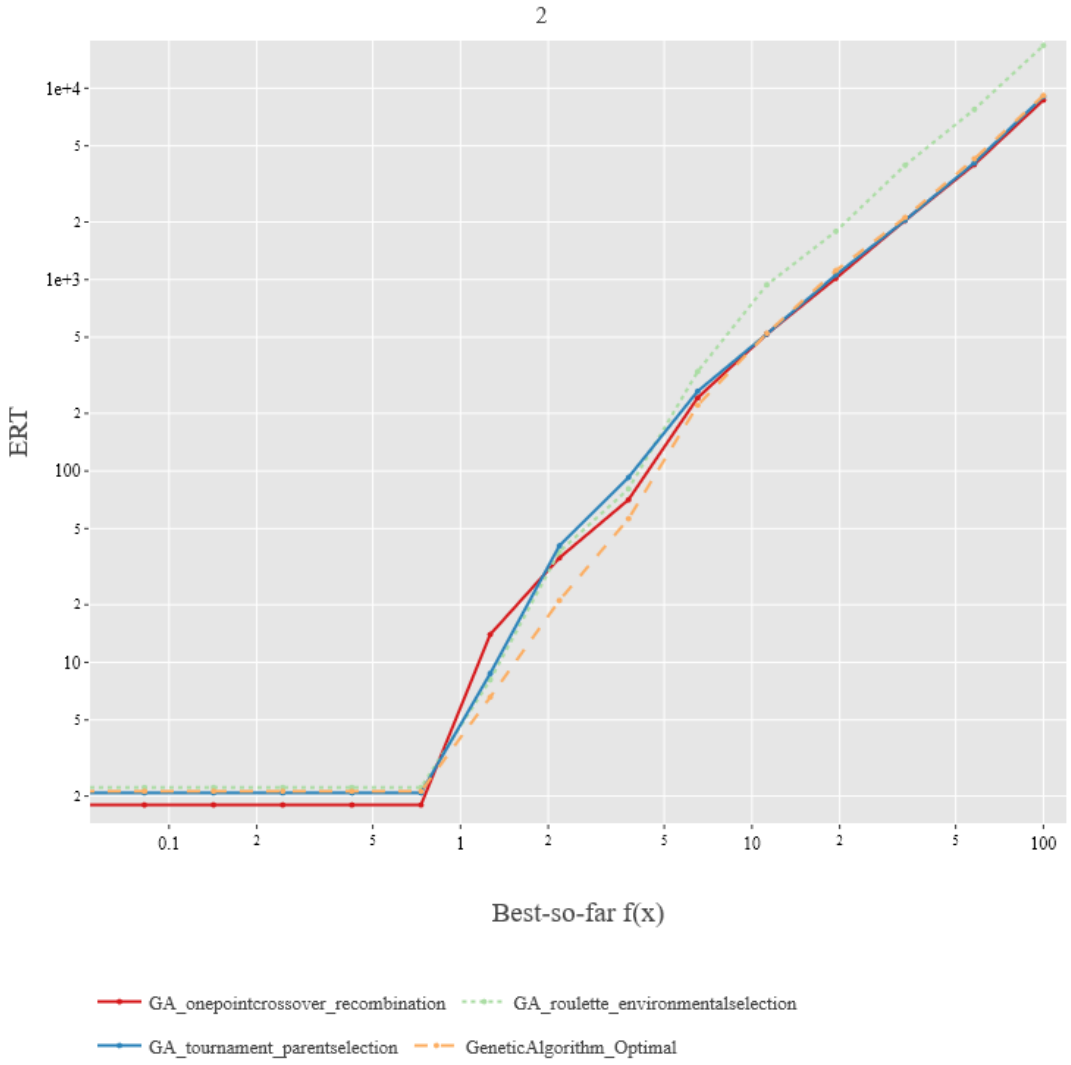

Fig. 4: The fixed-target performance on the Leading Ones problem.

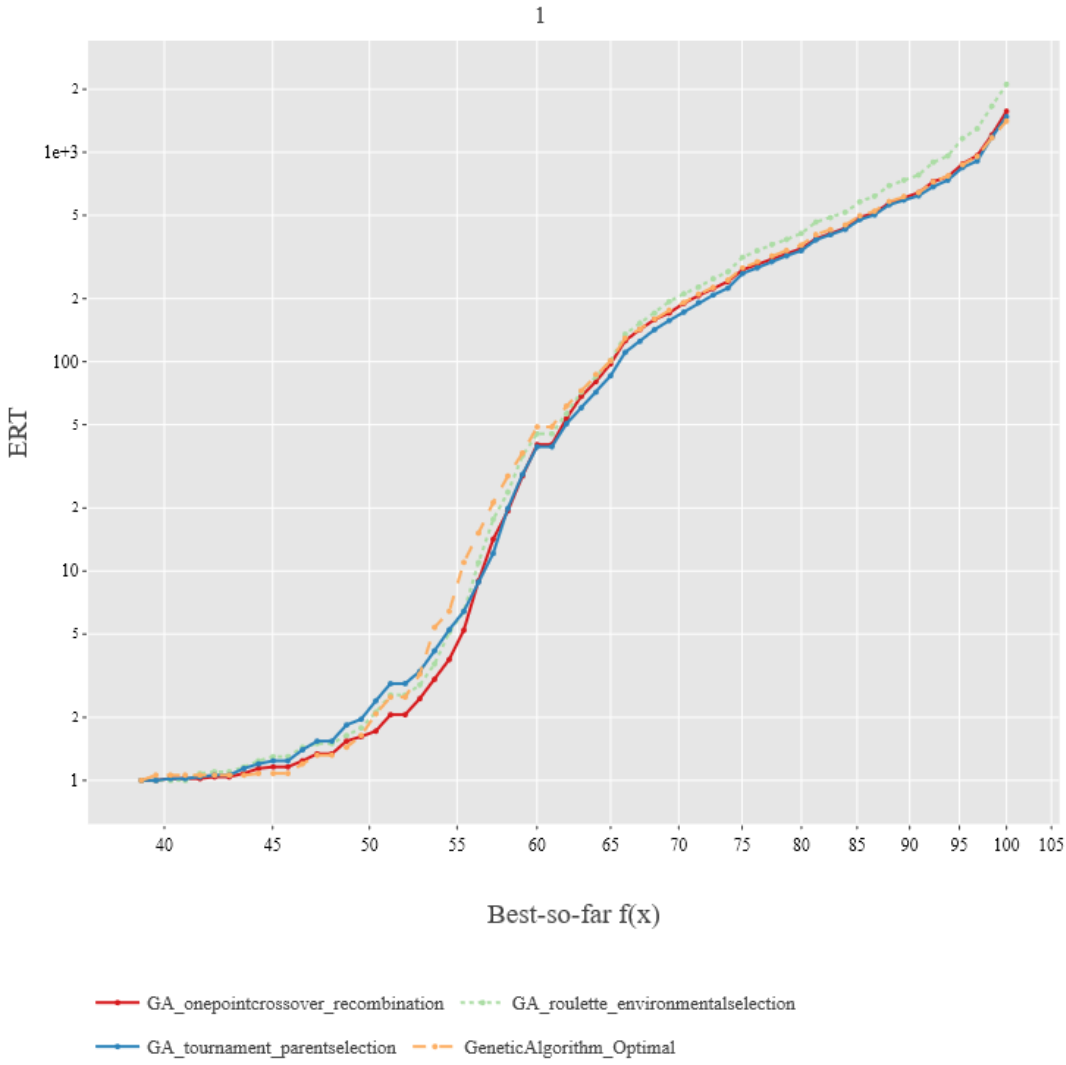

Fig. 5: The fixed-target performance on the One Max problem .

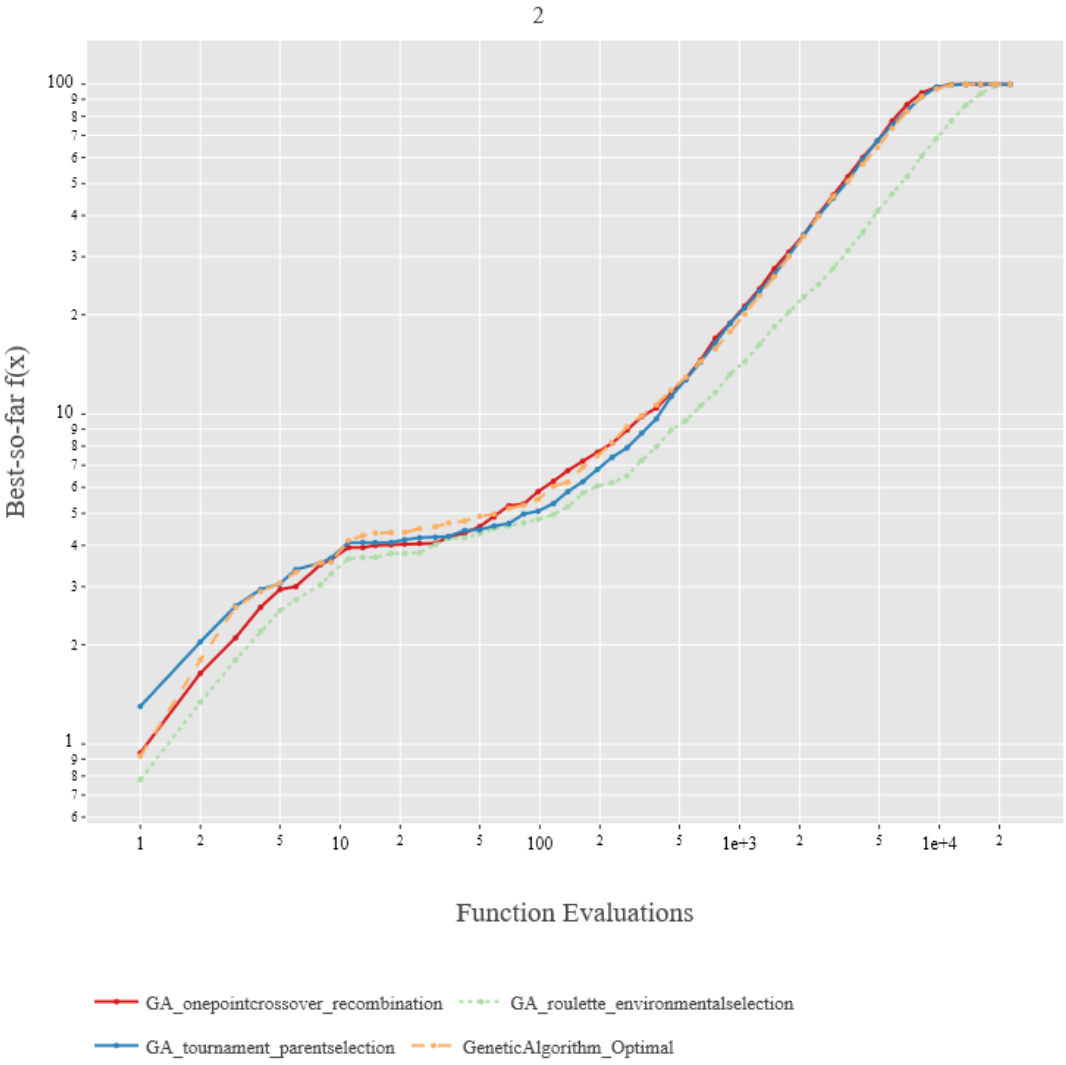

Fig. 6: The fixed-budget performance on the Leading Ones problem.

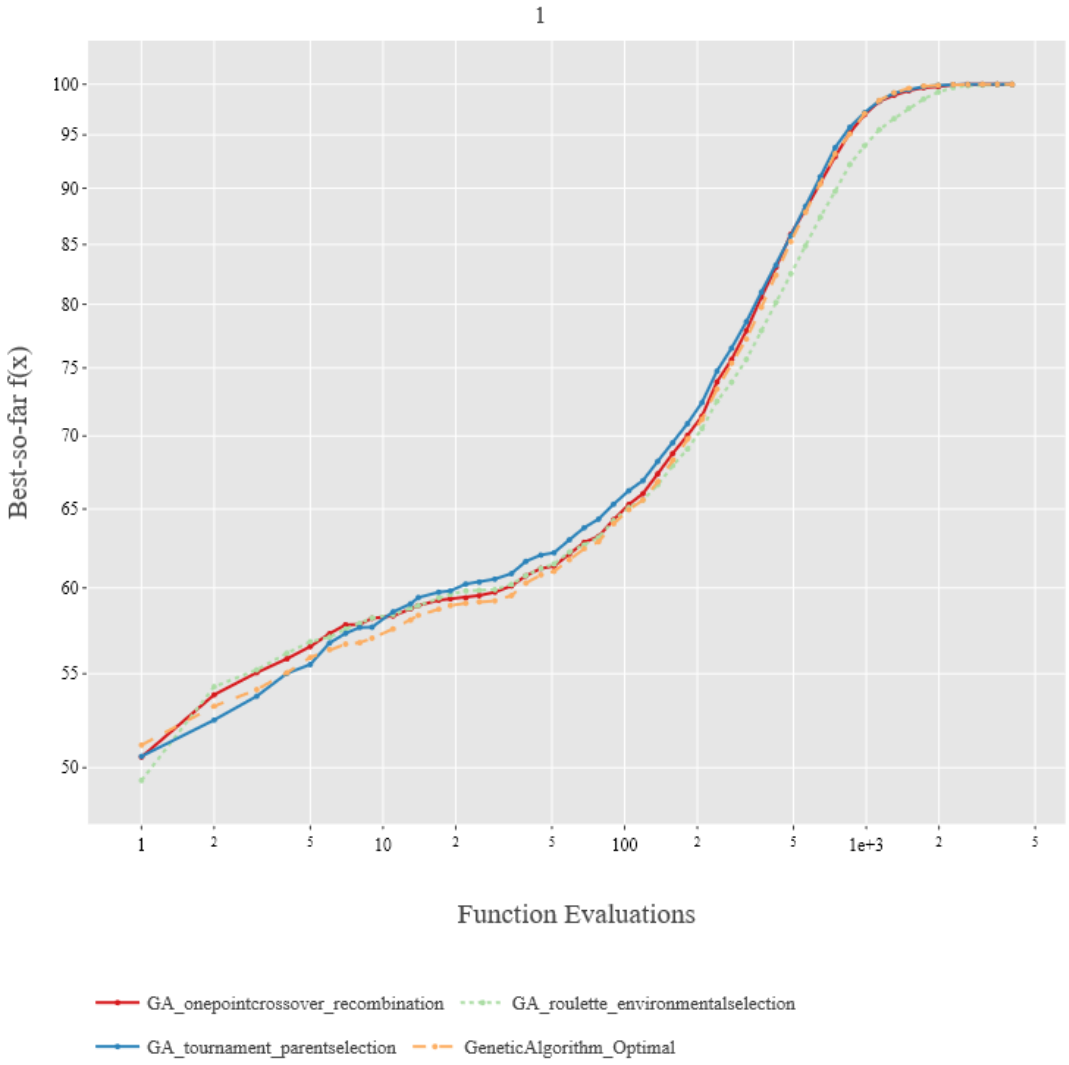

Fig. 7: The fixed-budget performance on the One Max problem.

B All of the factory functions

```
####### FACTORY FUNCTIONS #######



### Two versions of mutation/variation

def flipbit_mutation(gamma=0.1):
    def f(population):
        mut = np.random.random(population.shape) < gamma
        return np.logical_xor(population, mut).astype(int)
    return f

def swap_mutation():
    def f(population):
        x = np.hstack((np.arange(len(population)), np.arange(len(population
                                                    ))[::-1]))
        y = np.random.randint(0, len(population[0]), 2*len(population))
        population[x, y] = population[x, y][::-1]
        return population
    return f



### Two versions of environmental selection

def truncation_environmentalselection(M=10):
    def f(population, fitness):
        selection = np.argsort(fitness)[-M:]
        return population[selection], np.array(fitness)[selection]
    return f

def roulette_environmentalselection(M=10, repair_negs=True):
    def f(population, fitness):
        p = np.array(fitness)
        if repair_negs:
            p -= np.min(p) - 1
        selection = np.argmax(np.cumsum(p / sum(p)) > np.random.random((M,
                                                    1)), axis=1)
        return population[selection], np.array(fitness)[selection]
    return f
```

```
### A version of mating selection

# It takes two other functions, one for parent selection and one for
                                        recombination

def twoparent_matingselection(parent_selection, recombination, N=10,
                                        keep_old=True):
    def f(population, fitness):
        sorted_pop = population[np.argsort(fitness)][::-1]
        n_offspring = N - len(population) if keep_old else N
        selection = parent_selection(n_offspring, len(population))
        choose_first = recombination(n_offspring, len(population[0]))
        new_generation = np.where(choose_first, *sorted_pop[selection])
        return np.concatenate((population, new_generation)) if keep_old
                                                    else new_generation
    return f



# Four possible functions to give as argument to above function

def exponential_parentselection(s=5):
    def f(n_pairs, n_possibilities):
        return np.random.exponential(s, (2, n_pairs)).astype(int) %
                                                    n_possibilities
    return f

def tournament_parentselection(t=10):
    def f(n_pairs, n_possibilities):
        return np.min(np.random.choice(range(n_possibilities), (2, n_pairs,
                                                     t)), axis=2)
    return f

def uniformcrossover_recombination():
    def f(n_pairs, stringsize):
        return np.random.choice((0, 1), (n_pairs, stringsize))
    return f

def onepointcrossover_recombination():
    def f(n_pairs, stringsize):
        return np.arange(stringsize) >= np.random.randint(0, stringsize + 1
                                                    , (n_pairs, 1))
    return f
```

C Much better algorithm

The following algorithm we also made performs way better than the Genetic Algorithm we made.

```
y = problem(x := np.random.choice((0, 1), problem.meta_data.n_variables))
c, g = np.zeros((2, len(x)), dtype = int)
while sum(c) < len(c):
    g[(r := np.random.choice(np.flatnonzero((c == 0) & (g == 0))))] = 1
    x[r] = not x[r]
    c[r], x[r], g, y = (z:=problem(x))!=y, (z<y)^x[r], (z<=y)*g, max(z, y)
```

Exercise: Please have fun trying to figure out how it works and why this is so much better. Can it be seen as a genetic algorithm? What is its complexity?

Solution: It works by collecting at least a single bit of information at each iteration. The information is stored in the c and g variables. Bits that are correct will not be flipped again.

Although it does not really seem to follow the rules of a genetic algorithm, it can actually be seen as one. The population size is $M = 1$. The mating selection just puts the one individual $\boldsymbol{x}$ into the next generation. The mutation has a memetic learning operator built-in to it, that is very specific to the `LeadingOnes`-problem, and happens to also work very well for the `OneMax`-problem. The fitness function f is evaluated on $\boldsymbol{x}$ and the natural selection just chooses to keep $\boldsymbol{x}$ in the population. The iteration stops when 100% of the needed information is collected, which is the termination criterion.

The complexity of the algorithm is $O(n)$ for the `OneMax`-problem, because the maximum number of function evaluations is $n + 1$, where n is the bitstring-length. The complexity is $O(n^2)$ for the `LeadingOnes`-problem, because the maximum number of function evaluations is $\frac{1}{2}n(n+1)+1$.