GENERATIONAL GA MODEL

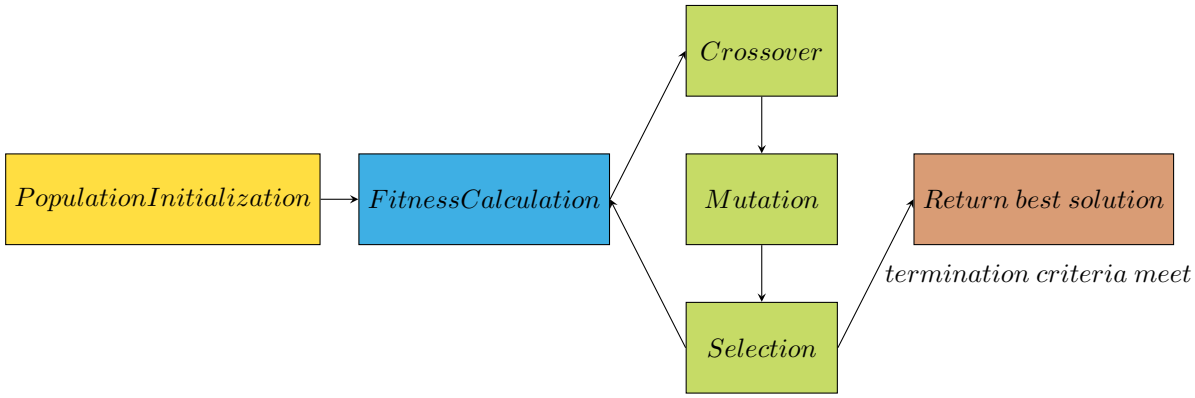
Ivan Horokhovskyi
s3069176
i.horokhovskyi@umail.leidenuniv.nl

Kitti Bernadett Keller
s3109089
k.b.keller@umail.leidenuniv.nl

November 5, 2021

1 Introduction

In this assignment we implemented the Genetic Algorithm which is a search-based optimization technique that uses principles of natural selection and genetics. Genetic Algorithms are a subset of Evolutionary Computation. It is frequently used to find optimal or near-optimal solutions to problems which otherwise would take a very long time to solve. In machine learning it is very often used to solve optimization problems.
In GA we have a population of possible solutions to the given problem. After first initializing our population we calculate for each individual a fitness value and then our population undergo recombination by applying crossover and mutation to them. These functions are inspired by natural genetics. These produces new children and the process is repeated over various generations. After each cycle we recalculate the fitnesses and the fitter the individual the higher the chance to it to mate.
Basically this algorithm is the implementation of the Darwinian Theory of "Survival of the Fittest".
With this algorithm we keep providing better solutions over generations till we meet the termination criteria.
The figure below shows the important steps of a Genetic Algorithm. This is the main concept behind our implementation as well.



2 Implementation

For this assignment our task was to test our algorithm on these 3 problems below and to find the best optimization.
The OneMax (om) problem:

$$OM : \{0,1\}^n \to [0..n], x \mapsto \sum_{i=1}^{n} x_i$$

The LeadingOnes (lo) problem:

$$LO : \{0,1\}^n \to [0..n], x \mapsto max\{i \in [0..n] | \forall j \leq i : x_j = 1\}$$

And finally Low autocorrelation binary sequences (labs) problem:

$$LABS : \{0,1\}^n \to \mathbb{R}, x \mapsto \frac{n^2}{2E(x)}$$

$$E(x) = \sum_{k=1}^{n-1} (\sum_{i=1}^{n-k} y_i * y_{i+k})^2 \to min$$

$$y_i = 2x_i - 1$$

Below we can see what parameters we are using for our algorithm and what settings did we use for those parameters.

Algorithm 1: A framework of Genetic Algorithm

Input : Population size μ
Mutation probability p_m
Crossover probability p_c
Crossover type t_c
Mutation type t_m

Termination : The algorithm terminates when it found the best solution

```
Initialize(P(t));
Evaluate(P(t));
for i = 0 to Budget do
    P'(t) ← matting_selection(P(t));
    P''(t) ← crossover(P'(t), p_c, t_c);
    P'''(t) ← mutate(P''(t), p_m, t_m);
    Evaluate(P'''(t))
end
```

Sample
fitness func x : list, ndarray

For implementing this algorithm we tried two different methods. There is one where we implemented an OOP approach with Sample class for our population and there is one where we worked with numpy data science library. Since numpy uses C backend and has vectorization acceleration under the hood this approach is much faster.

Below we can see our functions and these are almost the same for the class implementation as well. The only difference is that, there we pass list of samples as population to our functions.

For matting selection we used the proportional selection type that was taught at the lecture.

At Crossover we can choose between three types of method. We can choose to use one point crossover, multiple point crossover and uniform crossover.
At one point crossover we cut our parents at a randomly chosen point and bits to the right of that point are swapped between the two parent chromosomes.
At multiple point crossover we choose more points randomly and do the same as we did at one point crossover.
At uniform we basically flip a coin for each chromosome to decide whether or not it'll be included in the children.

For mutation we can choose between 3 types.
First is the flip type where we select one or more random bits and flip them.
Second one is the swap where we select two positions on the chromosome at random, and interchange the values. In the third approach it was made an attempt to implement dynamic mutation rate which was inspired by neural networks optimizers such as ADAM and Momuntum.

Algorithm 2: Matting selection

DEFINE FUNCTION $matting\_selection(population, fitnesses)$:
$min\_fit \leftarrow np.min(fitnesses)$;
$sum\_fit \leftarrow np.sum(fitnesses)$;
$ps \leftarrow (fitnesses - min\_fit)/(sum\_fit - fitnesses.shape[0] * min\_fit)$;
$ps \leftarrow np.nan\_to\_num(ps, copy = False)$;
if $np.sum(ps) = 0$ **then**
  $ps+ = (1 - np.sum(ps))/(ps.shape[0])$
end
RETURN $population[np.random.choice(population.shape[0], size = population.shape[0], p = ps)]$;

Algorithm 3: Crossover

DEFINE FUNCTION $crossover(population, p_c = 0.3, t_c = 1)$:
$res \leftarrow []$;
if $t_c = 1$ **then**
  for $i \in (0, population.shape[0], 2)$ **do**
    $res+ = one\_point\_crossover(population[i], population[i + 1], p_c)$
  end
  RETURN $np.array(res)$;
end
if $t_c = -1$ **then**
  for $i \in (0, population.shape[0], 2)$ **do**
    for $j \in (population.shape[0])$ **do**
      if $np.random.rand() < p_c$ **then**
        $population[i], population[i + 1] \leftarrow population[i + 1], population[i]$
      end
    end
  end
  RETURN $population$;
end
for $i \in (0, population.shape[0], 2)$ **do**
  $res+ = n\_point\_crossover(population[i], population[i + 1], p_c, t_c)$
end
RETURN $np.array(res)$;

Algorithm 4: Mutation

DEFINE FUNCTION $mutate(population, p_m, t_m)$:
if $t_m =' flip'$ **then**
  RETURN
  $np.logical\_xor(population, (np.random.uniform(size = population.shape) < p_m)).astype(int)$
end
if $t_m =' swap'$ **then**
  for $sample \in population$ **do**
    for $i \in sample.shape[0]$ **do**
      if $np.random.rand() < p_m$ **then**
        $pos2 \leftarrow np.random.randint(0, sample.shape[0])$;
        $sample[i], sample[pos2] \leftarrow sample[pos2], sample[i]$
      end
    end
  end
  RETURN $population$;
end

3 Experimental Results

To tune the hyperparameters it was decided to use random-search. The search ranges can be observed in Table 1. It was decided to use logspace for majority of parameters since uniformal distribution would create too many similar parameters which will lead to slower search. The logspace for probabilities (mutation and crossover) range from 1e-4 till 1 was used because there is no sense to use probability smaller than 1e-4 in such budget ranges, moreover it was interesting to test how high mutation probability will perform. For mutation it was decided to use flip or swap in majority of the cases and smart type is more experimental one. For crossover, -1 stands for uniform crossover, 1 - for single point crossover and N>1 for crossover with multiple crossover points.

Table 1: Hyperparameters Ranges

Parameter	Range
population size	logspace(1,3, 50)
mutation probability	logspace(-4, 0, 30)
crossover probability	logspace(-4, 0, 30)
crossover n	[-1] + [1] * 20 + range(2, 21)
mutation type	['flip', 'swap'] * 3 + ['smart']

In addition, since genetic algorithms are very stochastic the written program has an opportunity to fix random state to reproduce the results. Different random states were tested on each configuration to be able to get average performance of configuration. There were made 1000, 1000 and 500 random search iterations for OneMax and LeadingOnes and LABS problems respectfully. After that to eliminate lucky but unoptimal configurations it was taken 50 best configuration and it was made 15 more additional runs with different random states for each configuration to have more robust and accurate results. Moreover, for top 50 configuration there were made two-sample Kolmogorov-Smirnov statistical tests to compare distributions with the rest of runs, in such way we prove that the selected configurations are statistically better than just random configurations.

3.1 OneMax

This problem is pretty easy and more than 70% of reached global optimum value, the distribution of final function values can be observed on Figure 1. Due to this reason the budget spent and quantity of iterations before getting optimum were the most interesting metrics.

As it can be observed from Table 3.1 since the problem is pretty easy it can be solved in small size of iteration with small population size. It should be taken into the account that high mutation probability is present only with swap mutation type. Even though some configurations may take many iterations (rerun_mean column) these configurations have small population size and the main metrics is still quantity of evaluations. Neither crossover probability nor crossover N does not impact the result dramatically.

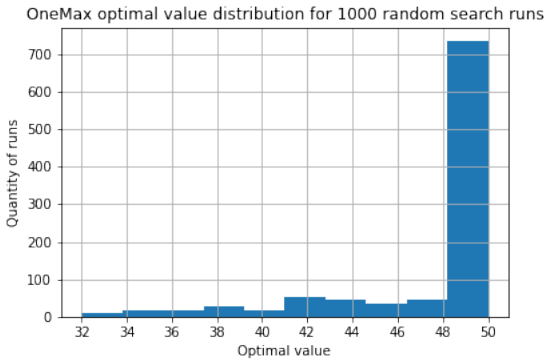


Figure 1: OneMax optimal value distribution for 1000 random search runs

population_size	mutation_probability	crossover_probability	crossover_n	mutation_type	rerun_mean	evaluations_cnt
10	0.011721	0.148735	19	swap	21.07	211
12	0.385662	0.030392	17	swap	18.60	223
12	1.000000	0.727895	2	swap	19.07	229
11	0.000489	0.280722	20	smart	22.07	243
12	0.006210	0.529832	7	swap	21.93	263
18	0.001743	0.148735	17	swap	18.00	324
16	0.727895	0.057362	6	swap	21.33	341
19	0.385662	1.000000	12	swap	18.33	348
21	0.004520	0.204336	20	swap	22.93	482
23	1.000000	0.385662	15	swap	22.67	521
26	0.000924	0.280722	19	swap	20.60	536
28	0.011721	0.108264	20	swap	22.33	625
34	0.030392	0.108264	18	swap	19.33	657
31	0.057362	0.385662	17	swap	22.73	705
23	0.000924	0.727895	16	swap	29.20	672
37	0.057362	0.057362	6	swap	20.73	767
31	0.078805	0.529832	11	swap	21.73	674
41	0.000489	0.529832	19	flip	21.13	866
49	0.280722	1.000000	16	swap	17.93	879
49	0.529832	1.000000	1	swap	19.40	951
26	0.000137	0.280722	14	swap	41.40	1076
54	0.022122	0.385662	6	swap	20.40	1102
54	0.001269	0.727895	17	flip	22.00	1188
60	0.030392	0.385662	9	swap	20.07	1204
72	1.000000	0.385662	4	swap	19.73	1421
66	0.000137	0.529832	11	swap	23.73	1566
72	0.022122	1.000000	10	swap	22.07	1589
72	0.008532	0.204336	14	swap	23.00	1656
66	0.030392	1.000000	1	swap	27.33	1804
105	0.727895	1.000000	12	swap	18.53	1946

3.2 LeadingOnes

LeadingOnes problem also was solved optimally by 35% of runs, distribution of objective function values is provided on the Figure 4. Unlike previous problem population size here is not sorted but still fluctuate with small population sizes. Best solutions have something in common. The swap mutation is leading again. Moreover, there is a strong negative correlation between crossover probability and crossover N which makes sence because too many mutations/changes can negativelly affect the population. The mutation probability values are pretty chaotic in swap mutation, but are low

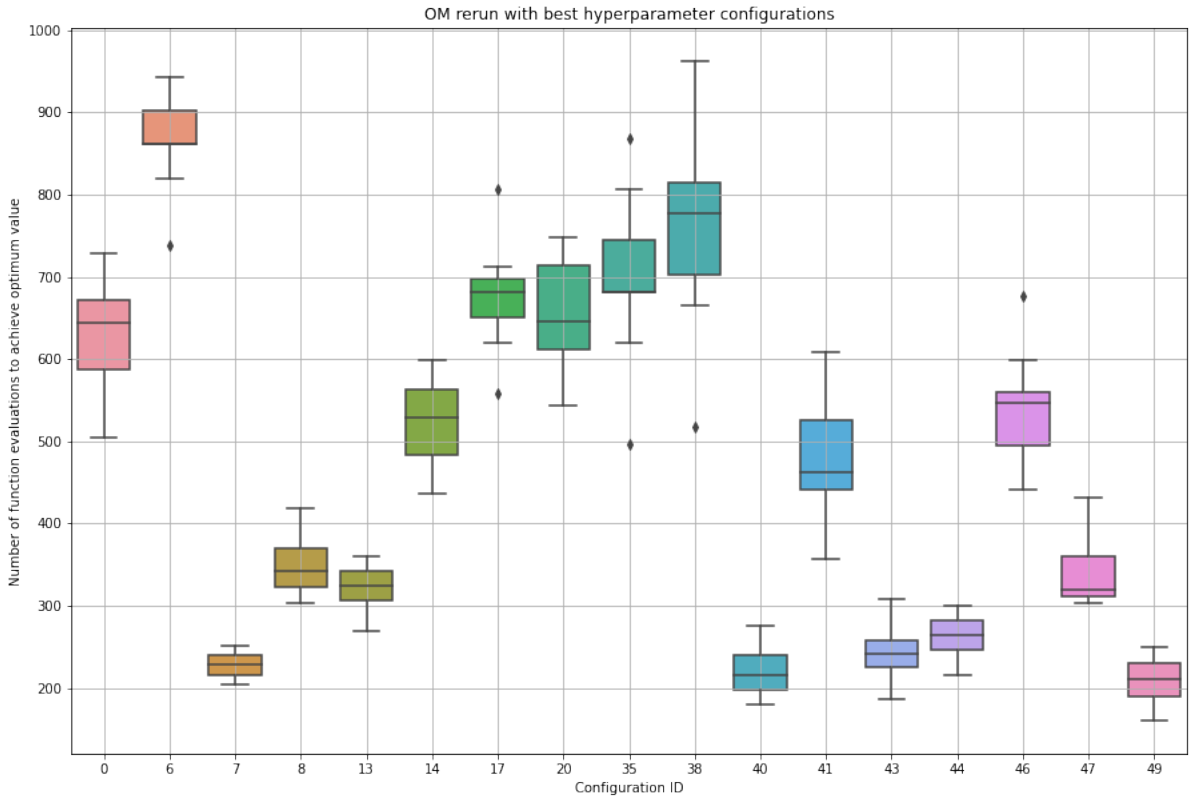


Figure 2: OneMax rerun with best hyperparameter configurations

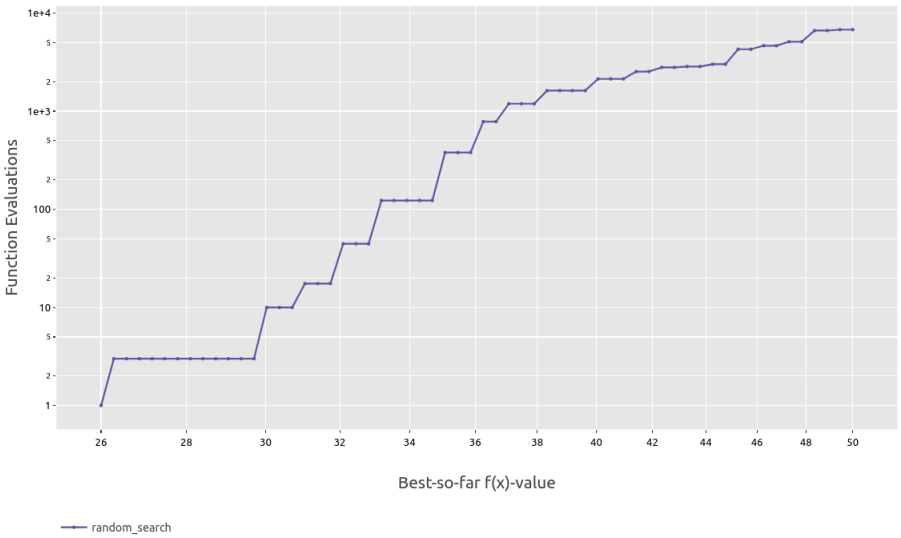


Figure 3: OneMax with optimal hyperparameters

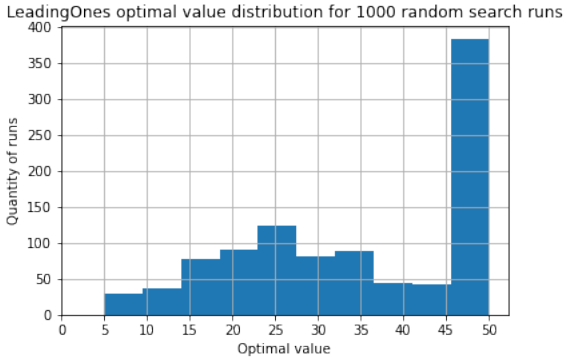


Figure 4: LeadingOnes optimal value distribution for 1000 random search runs

on other mutation types.

Overall, LeadingOnes problem requires more iterations to be solved, but is still solved with global optimum within given budget.

population_size	mutation_probability	crossover_probability	crossover_n	mutation_type	evaluations_cnt
13	0.529832	0.727895	3	swap	247
12	0.030392	0.529832	1	swap	317
18	0.385662	0.108264	18	swap	350
18	0.057362	0.108264	6	swap	446
12	0.016103	0.204336	1	swap	456
16	0.022122	0.385662	1	swap	492
12	0.385662	0.002395	8	swap	515
18	0.016103	0.280722	1	swap	536
21	0.011721	0.057362	19	swap	547
11	0.022122	0.006210	9	swap	572
16	0.148735	0.016103	4	swap	592
28	0.022122	0.727895	3	swap	597
12	0.529832	0.002395	7	swap	602
26	0.204336	0.001743	8	swap	664
19	0.108264	0.004520	12	swap	712
37	0.108264	0.529832	11	swap	713
21	0.030392	0.280722	1	swap	717
19	0.204336	0.041753	1	swap	774
41	0.280722	0.148735	16	swap	798
19	0.078805	0.022122	2	swap	817
34	0.385662	0.022122	7	swap	832
45	0.148735	1.000000	17	swap	846
31	0.078805	0.057362	4	swap	899
26	0.529832	0.006210	2	swap	931
37	1.000000	0.008532	15	swap	1144
60	0.041753	1.000000	14	swap	1148
60	1.000000	0.108264	13	swap	1180
37	0.108264	0.385662	1	swap	1191
11	0.016103	0.529832	3	smart	1217
15	0.030392	0.148735	3	smart	1270

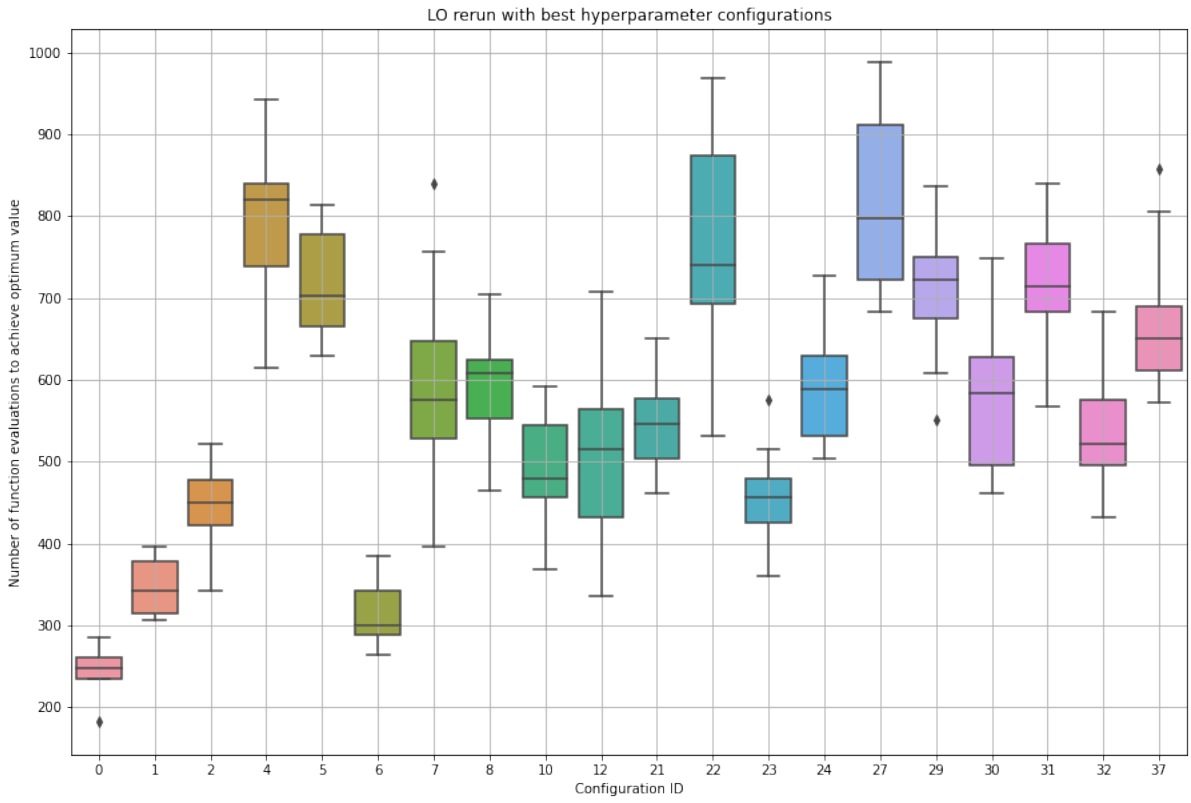


Figure 5: LeadingOnes rerun with best hyperparameter configurations

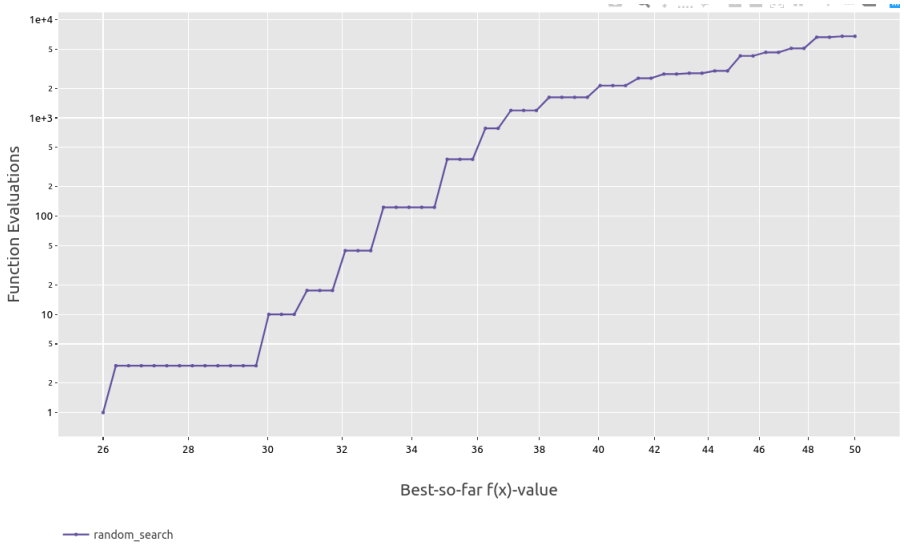


Figure 6: LeadingOnes with optimal hyperparameters

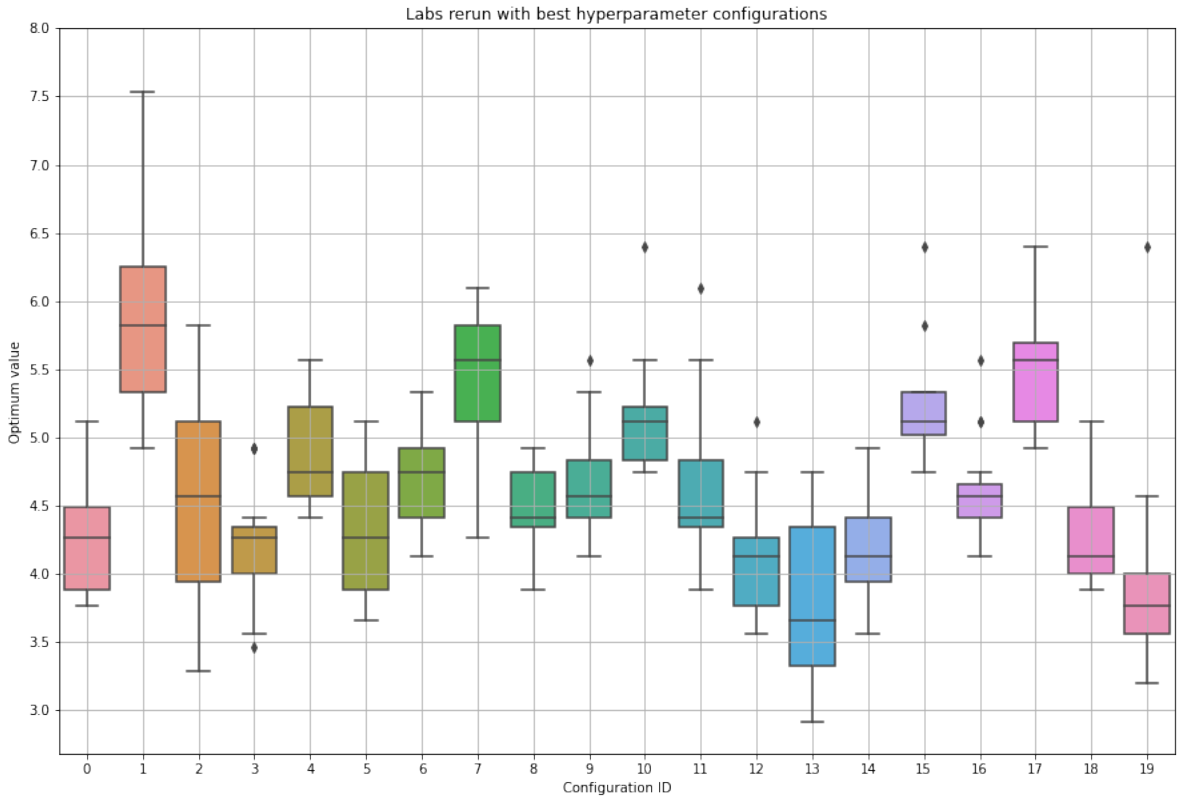


Figure 7: LABS rerun with best hyperparameter configurations

3.3 LABS

LABS problem is the most complicated one and non of the configurations was able to get optimum value of 8. The closest value was received is 7.529. Unfortunatelly this result was random seed dependent and more runs with same configuration did not give similar results.

After careful selection (500 hyperparameters, after that 50 reruns with 50 best ones) it was decided to highlight following hyperparameters (Table 3.3).

population_size	mutation_probability	crossover_probability	crossover_n	mutation_type	func_opt_mean
10.000000	0.030392	0.204336	1	flip	6.250183
59.636233	0.041753	0.078805	1	flip	5.945607
28.117687	0.030392	0.000100	7	flip	5.695234
14.563485	0.041753	0.008532	1	flip	5.652369
40.949151	0.030392	0.108264	4	flip	5.580588

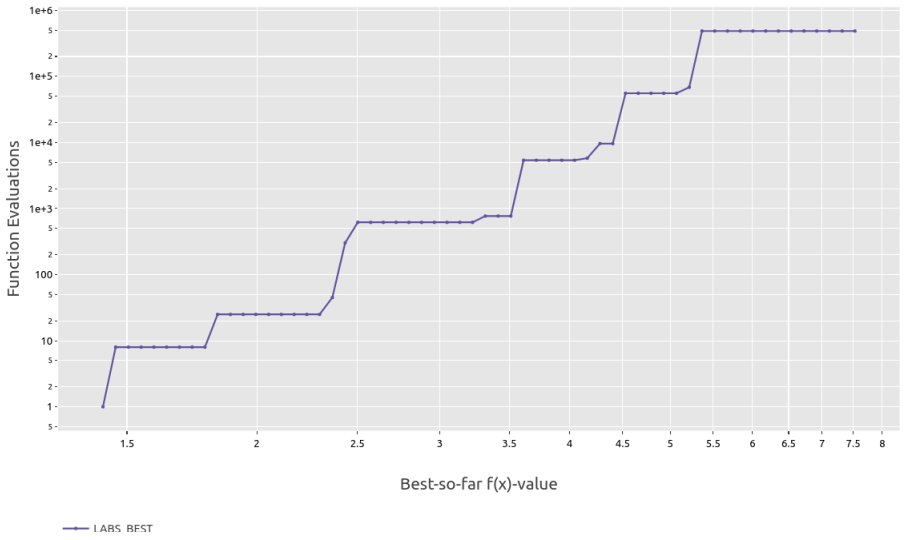


Figure 8: LABS with optimal hyperparameters

4 Discussion and Conclusion

It was made 2 versions of Genetic Algorithm were made: the "fancy" one with good code style and OOP approach and the fast vectorized variant. It is recommended to use 2nd version since it allows to experiment significantly faster though sometimes code is less understandable.

For OneMax and LeadingOne problems there is a clear trend that swap mutation work better, it was assumed that it helps to acheive maximum diversity within population for future crossover. While all best configurations for LABS problem use flip mutation with low mutation and crossover probabilities.

In all problems system benefits from small population sizes since with the same amount of function evaluations it is possible to make more generations.

The optimal hyperparameter configuration can be observed in the Table **??**.

problem	population_size	mutation_probability	crossover_probability	crossover_n	mutation_type	mean_func_val
OneMax	10	0.011721	0.148735	19	swap	50
LeadingOne	13	0.529832	0.727895	3	swap	50
LABS	10	0.030392	0.204336	1	flip	6.250183

Future work: more efficient hyperparameters optimization technique should be used (for instance Bayesian optimization). Moreover, advanced mutation rate techniques can be used.

Useful paper for OneMax [1] Basics about GA [2] GA-s in general [3] Useful paper for crossover [4] Useful paper for mutation [5]

References

[1] Georges Harik and Fernando Lobo. “A Parameter-Less Genetic Algorithm”. In: *academia.edu* (1999).

[2] Arthur Muthee. *The Basics of Genetic Algorithms in Machine Learning*. URL: https://www.section.io/engineering-education/the-basics-of-genetic-algorithms-in-ml/.

[3] Unknown. *Genetic Algorithm*. URL: https://algodaily.com/lessons/introduction-to-genetic-algorithms-in-python/matting-selection.

[4] Unknown. *Genetic Algorithms - Crossover*. URL: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htmn.

[5] Unknown. *Genetic Algorithms - Mutation*. URL: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm.