1 2  9 0 UNIVERSIDADE D COIMBRA

TP6 - Self-Adaptation in Evolutionary Strategies

Evolutionary Computation

Alexy de Almeida	2019192123	adenis@student.dei.uc.pt
Edgar Duarte	2019216077	edgarduarte@student.dei.uc.pt

PL1

May 2023

Contents

1 Introduction

Genetic algorithms, just as the model of Darwinian evolution defends, evolve a population of individuals using a fitness-based selection. The individuals of a population evolve by mutating and mating with each other in a process called crossover. During this evolution process, there are multiple stages the population goes through. At the start, it's useful for the population to **explore** the problem's domain and, later on, it's also beneficial for the population to **exploit** the neighbourhood. During the algorithm's runs, it may be advantageous to adapt the variation operators parameters dynamically and accordingly to the population's needs. In this report, we propose a self-adaption evolutionary algorithm that will allow us to analyze how changing the mutation's standard deviation during a run will improve the results obtained.

2 Self-adaptation evolutionary algorithm

The main goal of this work is to compare two algorithms and observe how and why their results differ. The first algorithm is a Generic Genetic Algorithm (GGA), while the second one is a GGA where the mutated deviation self-adapts over the generations. The goal is to see what advantages self-adaptation brings to the evolution process and if there really are any.

The implementation of the evolution algorithm is as follows:

- **Representation**: For the representation, each individual is composed of an array of floats, whose values vary between **lb** (lower bound) and **ub** (upper bound), which will be addressed later on. Each float in the array represents a number that will be used in the fitness functions of the benchmark problems.

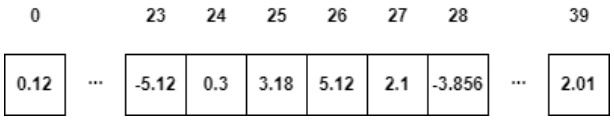

Figure 1: Individual representation

- **Variation operators**: For variation operators, mutation and crossover algorithms were implemented.
  - For the GGA **without self-adaptation**, the **mutation algorithm** goes through an individual's array and mutates each float individually using a Gaussian distribution of $\mu = 0$ and $\sigma = 1$.
  - For the GGA **with self-adaptation**, the process is similar but, while μ stays the same through the generations, a individual's σ value changes before each mutation process. The mutation of the σ value is done using a Gaussian distribution with $\mu = 0$ and $\sigma = \textit{offset\_mutation}$ (being offset_mutation a predefined parameter). This new value of σ is then saved and used for mutating the individual's float arrays.
  - At the end of each generation, we compare the fitness of each individual before and after the mutation. If the individual's fitness is better, then we save the new σ value, if not, we discard it and use the σ value of the previous generation.
  - A **two-point-crossover** was implemented. First, two parents are chosen using **tournament selection** of **5**, then their respective arrays are divided into **3 sections** which will be combined in order to create **two new children**. It's important to note that for the GGA with self-adaptation, the new children will have the mean of the σ value of the parents.
- **Survival Selection**: As we want to preserve the best individuals of the previous generation, an elitism algorithm was implemented.
- **Benchmark problems**: In order to test the two algorithms' performance, the following three benchmark were used: Rastrigin, Griewangk and Ackley, which are minimization problems that pose as good

challenges for this sort of algorithm, seeing that they are full of local optimas where the algorithm could get stuck.

3 Experimental scenario

In order to reach our goal of understanding the impact that self-adaptation has on the evolution process, several experiments were conducted. Each experiment was ran 30 times, with a fixed number of generations, size of the individual, number of individuals, mutation probability, crossover probability, size of the elitism and offset mutation. The table below shows the values used for each parameter. The combination of all the possible values makes for a total of 12 different experiments, since the only values that vary are "lb" (lower bound of domain), "ub" (upper bound of domain) and the benchmark problems.

Runs	30		
Generations	1000		
Individual size	40		
Number of individuals	90		
Mutation probability	10%		
Crossover probability	30%		
Elitism percentage	10%		
Offset mutation	3		
lb	-5.12 and -150		
ub	5.12 and 150		
Benchmark problem	Rastringin	Griewangk	Ackley

Table 1: Experiments values

The main idea behind changing the **ub** and **lb's** values is to induce different domain sizes for the algorithm to explore. This will allows us to analyse both algorithm's performance in terms of exploration and exploitation. The different **benchmark problems** were chosen in order to create challenging environments, full of local optima that the algorithms have to surpass. All of the problems have their global optima at x = [0,0,...,0]. Initially, the **number of generations** chosen was 10000 but, seeing that this did not allow us to see both algorithms performance until their convergence, it was updated to 1000 generations.

The number of individuals chosen was 90 and their array size was kept at 40, as these values not only promote a sizable population, but also give some complexity to the evolution process, seeing that there are 40 values to optimize in each individual.

Regarding the mutation probability, we found out that higher mutation probabilities lead to premature convergence, while a low mutation value leads to a slow evolution. Due to this, a mutation probability of 10% was chosen, as it was not too high, nor too low, allowing for both exploration and exploitation.

Experience	Problem	Self-Adaptive Deviation	lb	ub
1	Rastrigin	No	-150	150
2	Rastrigin	Yes	-150	150
3	Rastrigin	No	-5.12	5.12
4	Rastrigin	Yes	-5.12	5.12
5	Griewangk	No	-150	150
6	Griewangk	Yes	-150	150
7	Griewangk	No	-5.12	5.12
8	Griewangk	Yes	-5.12	5.12
9	Ackley	No	-150	150
10	Ackley	Yes	-150	150
11	Ackley	No	-5.12	5.12
12	Ackley	Yes	-5.12	5.12

Table 2: Experimental sets

4 Results and discussion

4.1 Rastrigin problem

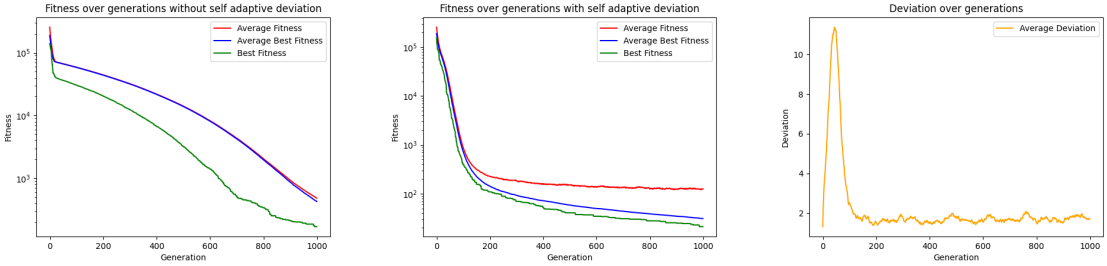

Figure 2: Fitness evolution for **Rastrigin**, **lb** = -150, **ub** = 150

In this experiment we have a lower bound of -150 and an upper bound of 150, meaning the algorithms have a sizable domain space to explore. Initially, the deviation's average value has a big jump, but very quickly stabilizes at values close 1. This hints at initial exploration followed by exploitation. Looking at the multiple fitness lines, it is perceptible that, for the algorithm with self-adaptation, the populations have overall good diversity, with the average fitness quickly converging, while the average fitness of best individuals keeps improving. On the other hand, the algorithm that does not use self adaptation has a lower performance (if we look at the scales of the graphs) and does not converge in 1000 generations. It is to note that for this algorithm it seems that the red and blue line are on top of each other, but that derives from the scale used, which is a logarithmic one. When comparing the results from both algorithms, we can see that self-adaptation considerably increases performance.

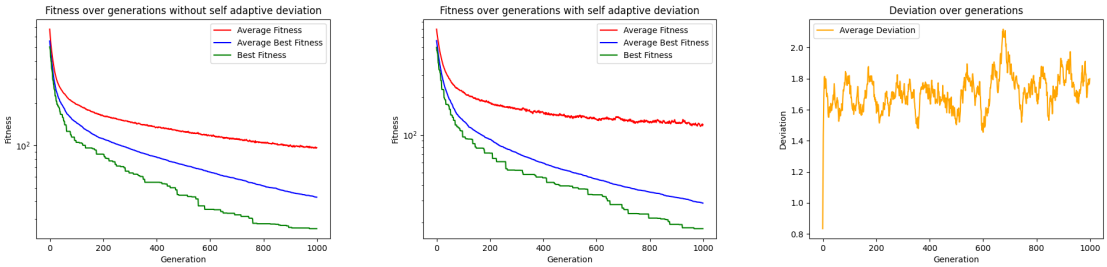

Figure 3: Fitness evolution for **Rastrigin**, **lb** = - 5.12, **ub** = 5.12

In these experiences, we lower down lb to -5.12 and ub to 5.12. Similar to the previous results, for the algorithm with self-adaptation, the average fitness starts to converge while the average fitness of best individuals keeps improving. This time, the average fitness of the algorithm without self-adaptation also begins to converge and the overall performance of both algorithms is very similar, suggesting that, when it comes to exploitation, the self-adaptive method isn't of much help.

Looking at the deviation over generations plot, the value of the deviation quickly stabilizes around 1.8. This suggests that for exploitation, the algorithm finds this value as ideal for this optimization problem.

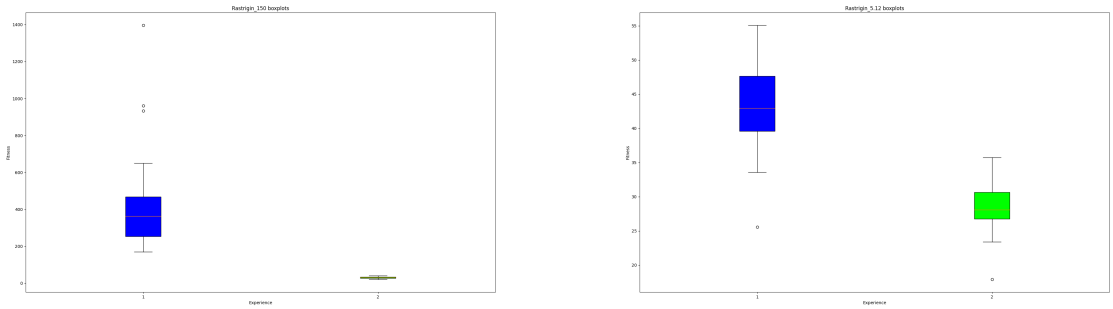

Figure 4: Boxplots **Rastrigin**, **lb** = -150 and **ub** = 150 (left), **lb** = -5.12 and **ub** = 5.12 (right)

For the first 2 experiences, we can see that there is a significant difference between algorithms. The self-adapting algorithm has less outliers and its box is significantly lower. Looking at the right plot, experiences 3 and 4 have almost no outliers and the self-adapting algorithm's box is smaller and lower, meaning more consistency and overall better performance.

Overall, the results are within what is to be expected, seeing that the first 2 experiences have a much larger domain and more exploration needs to be done. This means that there is a lot more variability between runs in 1000 generations.

Between experience 1 and 2, it seems that there is a relevant difference in performance. Not only does experience 2 (with self-adaptation) appear to have way less outliers, it also has its quartiles lower. This could indicate that **it is better than the normal approach**. This will be verified later on in the hypothesis testing.

4.2 Griewangk problem

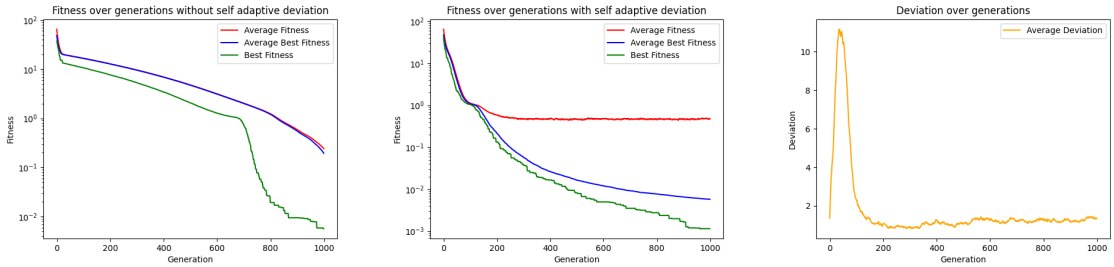

Figure 5: Fitness evolution for **Griewangk**, **lb** = -150, **ub** = 150

When using self-adaptation, the average fitness converges, while the average best fitness and best fitness keep improving. Initially, the deviation has a big jump, but quickly stabilizes around the value of 1 (again, suggesting initial exploration followed by exploitation). On the other hand, when not using self-adaptation, none of the fitness lines ever converge. The average performance of this algorithm is also significantly worse when self-adaptation isn't used, even though both algorithms find a best individual with similar fitness. On the other hand, when **using the self-adaptation, the results are more consistent**.

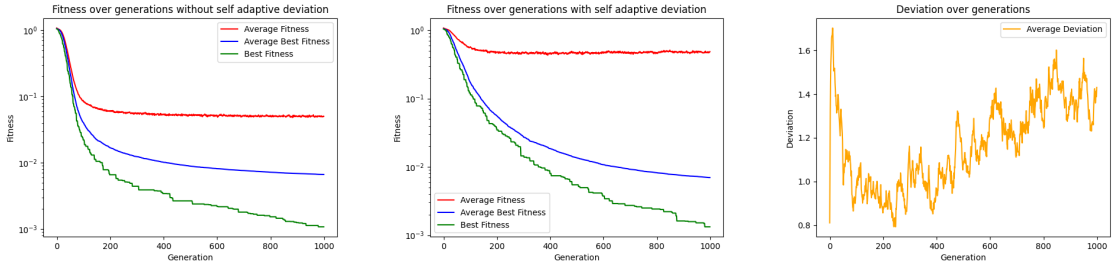

Figure 6: Fitness evolution for **Griewangk**, **lb** = -5.12, **ub** = 5.12

When the domain is reduced, both algorithms' average and average best fitness converge at values around of scale 10^{-1} and 10^{-2} respectively. It **does not appear that self-adaptation brings any benefits in terms of performance** for this benchmark problem, but it **does bring a lot more diversity**, seeing that the gap between the average and average of best fitness is a lot larger when using this algorithm.

Looking at the deviation graph, the deviation fluctuates between 0.8 and 1.7. As there is a small domain in these experiences, the results are within our expectations, as **the algorithm will predominantly want to exploit**. What this means is that it wants smaller and more controlled changes each generation, which implies a lower deviation.

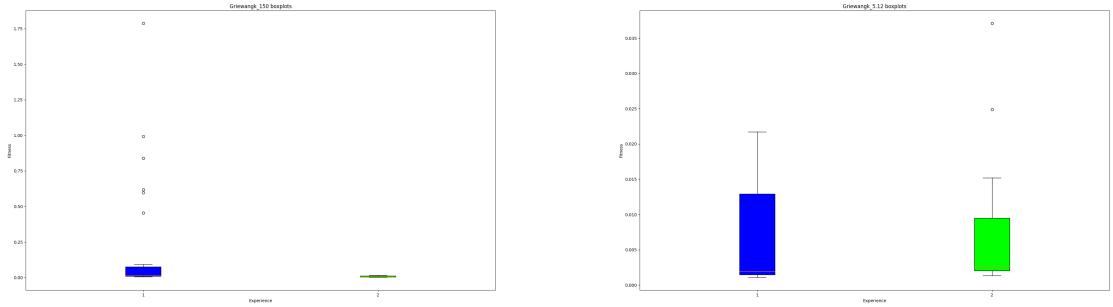

Figure 7: Boxplots **Griewangk**, **lb** = -150 and **ub** = 150(left), **lb** = -5.12 and **ub** = 5.12 (right)

The left graph shows that the the self-adapting algorithm's box almost disappears in relation to its counterpart, meaning that its performance is superior. On the right plot, the algorithms present a similar box, meaning that they have similar performances.

4.3 Ackley problem

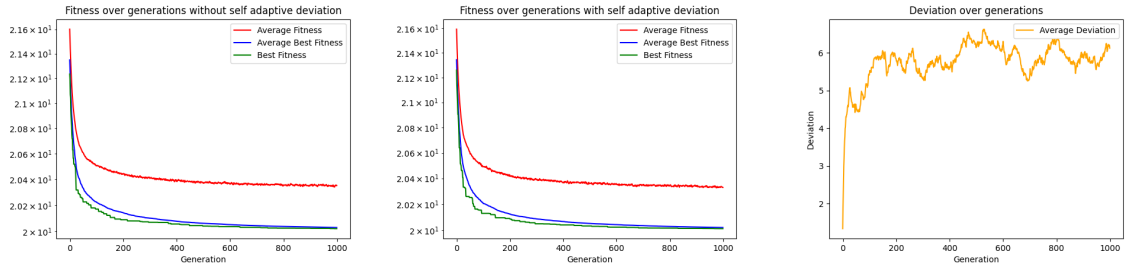

Figure 8: Fitness evolution for **Ackley**, **lb** = -150, **ub** = 150

The results for these experiences are not within what was expected. Not only do both function quickly converge, but they do so to the same high value. The only explanations are that the algorithms get stuck in a strong local optima every generation or that the Ackley function is not feasible for these upper and lower bound values, even though no documentation online mentioned such restrictions. It is interesting that, looking at the deviation over generations graph, the deviation goes to very high values, trying to get the population out of the optima it is stuck at.

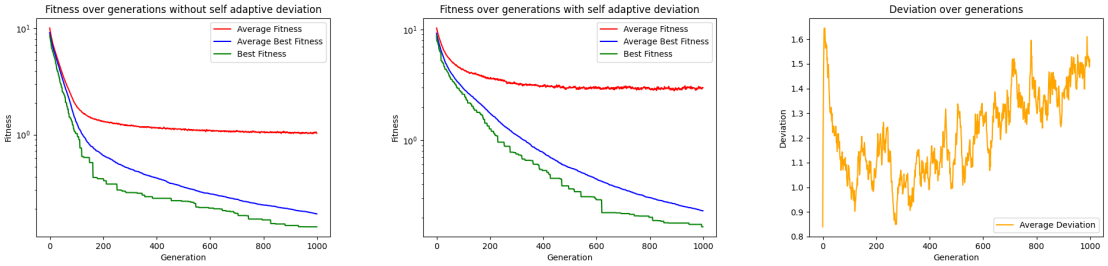

Figure 9: Fitness evolution for **Ackley**, **lb** = -5.12, **ub** = 5.12

More in line with what is expected, the Ackley function presents very similar results to the past two benchmark problems for a small domain. Again, both algorithms present similar performance levels, and again the self-adapting's population has a bigger diversity. Looking at the deviation, the deviation is very contained in the interval [0.8,1.7], meaning that there aren't any big jumps or drops. Again, this is to be expected, seeing that with a small domain the algorithm will want to exploit.

The boxplots of the above mentioned experiences are as follows:

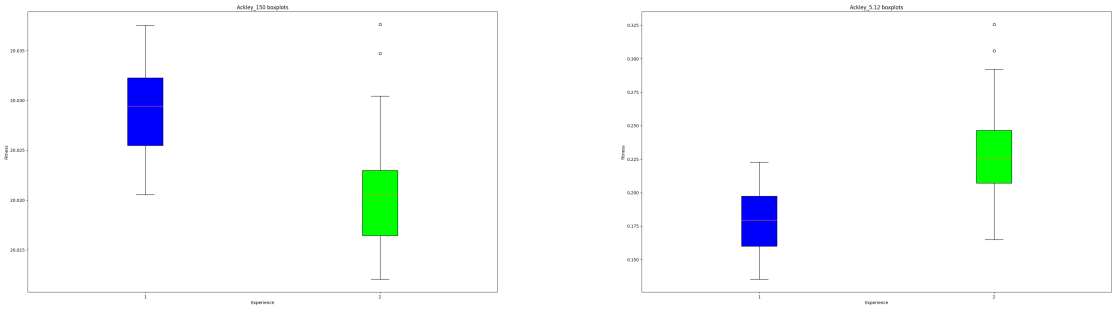

Figure 10: Boxplots **Ackley**, **lb** = -150 and **ub** = 150 (left), **lb** = -5.12 and **ub** = 5.12 (right)

In the left graph, it appears that the self-adapting algorithm has a slight edge over the normal algorithm, but once we inspect the scale of the graph, we can see that the difference is insignificant. Looking at the graph on the right, it appears that the algorithm that does not use self-adaptation has a better overall performance, when compared with its counterpart. This is surprising but understandable. Seeing that in this case the algorithm needs to exploit, the variability that a changing mutation deviation introduces may induce in a slower rate of descent. But if we look at 9, we can see that the trade off for this slower descent is a much higher population diversity.

4.4 Statistical Analysis

To draw some conclusions about those results, we want to test the following hypotheses:

- H_0: The algorithms have the same performance
- H_1: The algorithms have different performances

Those hypotheses are specific to the benchmark problem we're testing, so for example, the hypotheses for the Rastrigin problem are:

- H_0: The algorithms have the same performance for the Rastringin problem
- H_1: The algorithms have different performances for the Rastringin problem

To choose our statistical test, we need to verify if the data meets assumptions for parametric tests. The first thing to do is to check if the data normally distributed. Since the number of runs done for each experiment is below 50, this can be done by using the Shapiro-Wilk test, which will give us the p-value of that experiment. If the p-value is below 0.05, then the data is not normally distributed.

Below is a table for each pair of experiments (data with deviation and data without deviation). If the p-value is above 0.05 in both experiments from the same pair, then we can proceed to check the homogeneity of variance.

Experience	1	2	3	4	5	6	7	8	9	10	11	12
p-value	3.57e-05	0.99	0.84	0.21	1.39e-08	1e-05	4.47e-05	1.03e-06	0.72	0.02	0.45	0.06

As we can observe, the data is normally distributed in the pair of experiments 3&4 and 11&12, so we need to proceed with the homogeneity of variance to check if the variances of the groups are different. If the p-value of the test is less than 0.05, then the variances may be considered significantly different and the assumption of homogeneity isn't checked. Since our data contains interval data and independence, not checking the homogeneity of variance leads to having non-parametric data. On the other hand, if the assumption is checked, then the data is parametric. By using Levene's test for homogeneity of variance, the results are as follows:

Experience	3	4	11	12
p-value	0.009		0.38	

We can see that the second pair of experiences obtains a p-value above 0.05, which is the only pair of experiences that allows us to use the t-test since the data meets the assumptions for parametric test. As the rest of the experiences, they do not meet those same assumptions and we need to use the Mann-Whitney test. The results are as follows:

Experience	1	2	3	4	5	6	7	8	9	10	11	12
p-value	3.02e-11		8.9e-10		5.19e-07		0.16		3.81e-07		8.01e-09	

All the experiences, on the exception of the pair 7&8, have very small p-values, so we can verify the null hypothesis that the two algorithms have the same performance for their benchmark problems and lb/ub values. Therefore, we can say that there is not a significant difference in the performance of the two algorithms.

On the other hand, the p-value of the pair 7&8 is a little above 0.05, so we reject the null hypothesis. We cannot directly accept the alternative hypothesis and conclude that the two algorithms have different performances for the Griewangk problem, in which case we would need to estimate effect sizes or explore what variables may be impacting the results.

5 Conclusion

In conclusion, with this project we were able to show that the use of self-adaptation does indeed bring benefits to the evolution process. Not only did it reveal to have a better performance, it also creates a more diverse population. It is an especially valuable algorithm to use if the problem at hand has a very big domain, that needs to be thoroughly explored, as the deviation can adapt to the algorithm needs.

For future work, a multiple population approach could be an interesting comparison point to the other algorithms, especially for these kind of benchmark problems. A cooperative coevolution algorithm with self-adapting mutation should prove to be more efficient in finding the lowest value, but should also prove to be much heavier than the other approaches.