Genetic Algorithms & Evolution Strategies

TAIA - UFPE - Project Report

UNIVERSIDADE FEDERAL DE PERNAMBUCO

utc Université de Technologie Compiègne

Introduction

Genetic Algorithms and Evolution Strategies are two search heuristic that mimics the process of natural selection. These algorithms are part of the Evolutionary Computation, a sub category of the Artificial Intelligence field.

These algorithms are bio-inspired, which means that they generate solutions to optimisation problems using techniques inspired by natural evolution, such as inheritance, mutation, selection and crossover.

Project objectives

The different objectives for the TAIA project were the following:

- Propose an implementation of an Evolution Strategy
- Propose an implementation of the Ackley Function
- Propose an implementation of a Genetic Algorithm, to compare the results

Project Tree

```
/jeankevin/                              : Main folder
      __init__.py__                      : Python config file
      jeankev.py                         : Script Launching file
      + “simuxxx.pdf”                    : Simulation saves files
      doc/                               : Documentation & Report folder
            + “xxx.xxx”                  : “xxx” documentation file
      modules/                           : Modules folder
            __init__.py                  : Python config file
            individualClass.py           : Individual generic class
            ackleyIndividualClass.py     : Ackley Individual class
            numberCoupleClass.py         : NumberCouple Individual class
            populationClass.py           : Population class
      views/                             : Views folder
            __init__.py                  : Python config file
            settingsView.py              : Settings views module
            displayView.py               : Display view module
            saveFigure.py                : SaveFigure view module
      venv/                              : Virtualenv folder
```

General functioning

python version required :

```
python –version == ‘Python 2.7.9’
```

Step 1 : launch the script

```
➜ python jeankev.py
```

Step 2 : choose the individual and the algorithm

```
GA or ES
--> 1: GA with NumberCouple
--> 2: ES with AckleyIndividual
--> 3: GA with AckleyIndividual
2
```

This ES implementation project is an adaptation of an older project on GAs. The NumberCouple class is the implemented class from that older project and represents a couple of real, potential solution of the old function we had to test.

An AckleyIndividual is a potential solution of the Ackley function, indeed that’s the individual to choose to this project’s execution (however you still can choose to play with the NumberCouple individuals)

This view allows you to choose between Evolution Strategy algorithm (choose ES) and Genetic Algorithm (choose GA)

Step 3 : preset or manual set

```
===== OPTIONS =====

PRESET
Use preset ?


-> 1: Source based preset

-> 2: I WANT TO SET BY MYSELF
2
```

This view provides you the possibility to use source based preset (faster, set the function parameters in the sources) or to use the console interface (slower, but more user friendly ?) to set the algorithm settings.

Step 4 : algorithm's settings

—> In the sources :

```
    preset = int(raw_input(
        "PRESET\n"
        "Use preset ?\n"
        "\n\n-> 1: Source based preset\n"
        "\n-> 2: I WANT TO SET BY MYSELF\n"
    ))

    os.system("clear")

    if preset == 1:
        options["iterations"] = int(100000)
        options["stopFitness"] = float(0.90)
        options["mode"] = 'real'
        options['crossMode'] = 'randomMultiPoint'
        options["maximalPopulation"] = int(30)
        options["mutationMode"] = 'everyNucleotid'
        options["mutationProbability"] = float(2)
        options["verbose"] = False
        options["initialPopulation"] = int(300)
        options['selectionMode'] = 'rouletteWR'
```

You have the possibility to use the sources to determine a preset that can be used, for example, when you need to make a test series with the same settings preset.

—> Console interface :

```
BASICS
Stop Iterations Number:
100

Stop Fitness:
0.9

GENERATIONS
n setting:
lambda (number of child from the father) = 8 * n
mu (number of best child selected to make new father) = lambda / 4
t (global step size) = 1 / (n)^(1/2)
ti (component step size) = 1 / (n)^(1/4)
10
RECOMBINATION
Recombination mode:
1- Intermediate
2- Select Best
3- Weighted
3
MUTATION
Mutation mode:
1- 2 Learning Rates, N Sigmas
2- 1 Learning Rate, 1 Sigma
2
```

The other possibility is to use the console interface. This pretty simple, just enter the desired values.

NB : please don't enter unadapted values, I didn't implement an exception raising system.

Step 5 : verbose mode

```
VERBOSE
Verbose Mode
-> 1: Enabled
-> 0: Disabled
```

The application provides a verbose mode, which allows the user to see a recap at every algorithm step as the following example :

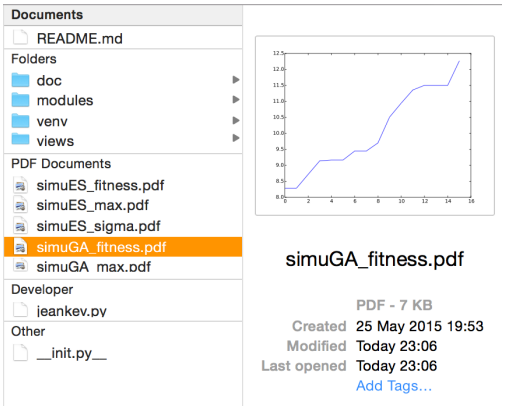
```
=== RECOMBINATION WEIGHTED ===


- New individual sigmas : [6.3857110408109765, 6.3857110408109765, 6.3857110408109765, 6.
3857110408109765, 6.3857110408109765, 6.3857110408109765, 6.3857110408109765, 6.385711040
8109765, 6.3857110408109765, 6.3857110408109765, 6.3857110408109765, 6.3857110408109765,
6.3857110408109765, 6.3857110408109765, 6.3857110408109765, 6.3857110408109765, 6.3857110
408109765, 6.3857110408109765, 6.3857110408109765, 6.3857110408109765, 6.3857110408109765
, 6.3857110408109765, 6.3857110408109765, 6.3857110408109765, 6.3857110408109765, 6.38571
10408109765, 6.3857110408109765, 6.3857110408109765, 6.3857110408109765, 6.38571104081097
65]
- New individual xs : [-12.633878424385077, 3.4936607860507864, -4.792273232414377, -2.95
2308458471977, -13.99954891026696, -4.769663821181614, 11.04907828194429, -5.566190614998
8924, -6.123705864172369, 2.5936941253399395, -13.420838558900355, 10.363211715496071, -7
.180816568830446, 3.2312753030592565, -7.774073571523031, -11.574355353343039, 9.90545039
8239172, 6.145965054516727, -10.684121213665177, -14.798828454776316, -6.639168650379104,
 1.8101091089527706, -9.740070101026625, -10.5726937628587, 11.846615700121065, -6.588121
895035984, -13.523348717722548, -6.496162374354424, -5.395598190737603, -4.24772032323083
7]
```

or, in non verbose mode, the current iteration, as follows :

```
Iteration : 14 / 100000
```

Step 6 : results consulting

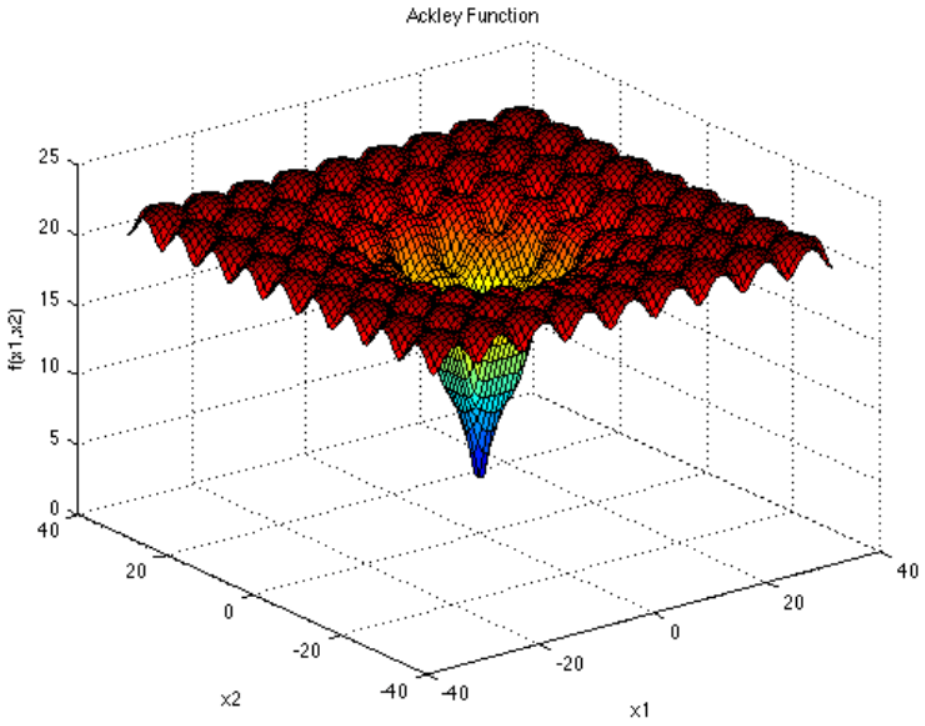
At the end of the algorithm execution, every result curves are available in pdf in the root ‘jeankevin’ folder (jeankevin is the name I’ve chosen for my project).



1. Evolution Strategy

1.1. The Ackley Function

All along this project, we work on the Ackley function.



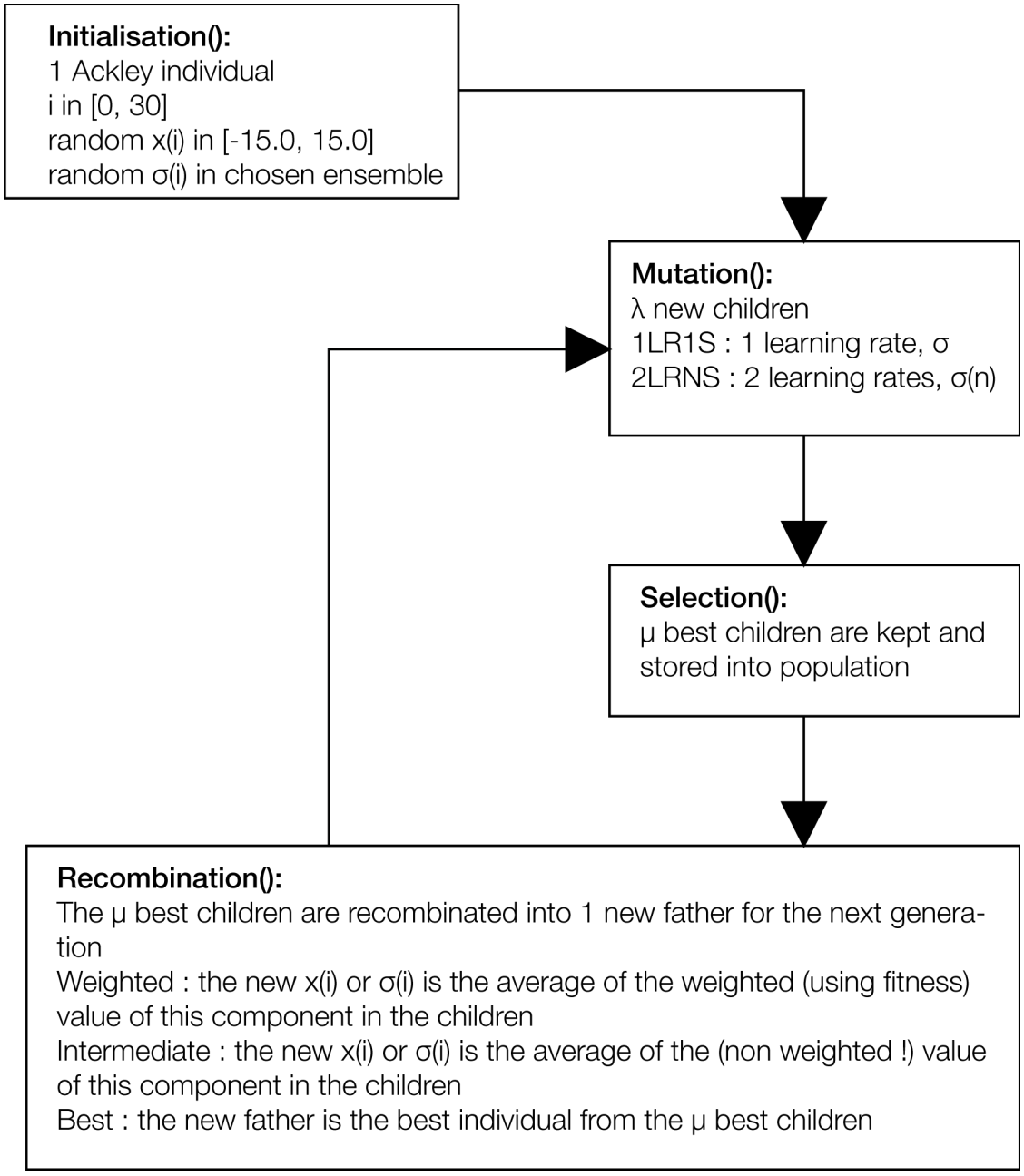
$$f(\mathbf{x}) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d}x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$

Used parameters:

- a = 20
- b = 0.2
- c = 2π
- d = 30

The interest in using this function is the high number of local minimals, used as traps to test the limits of the algorithm.

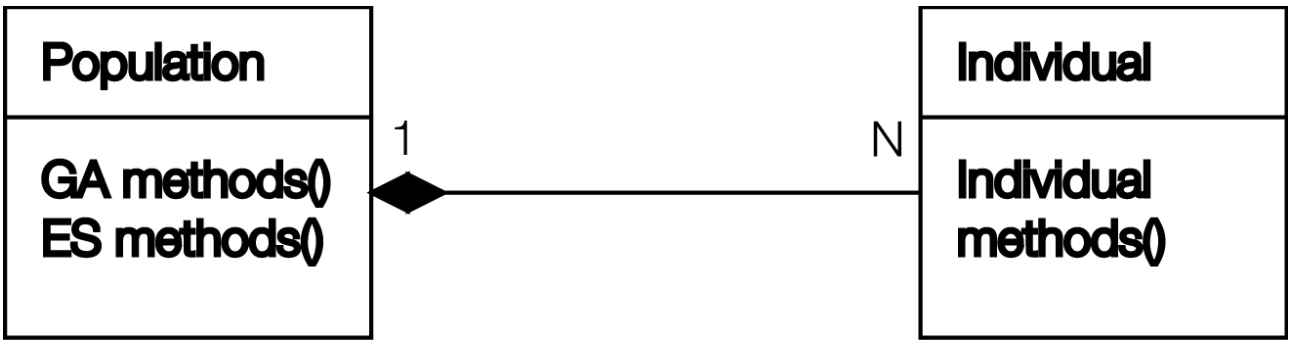
1.1. Schematic representation of the used algorithm



1.2. Individuals Representation

I've learned to code in Python with this project. My aim was to learn the object oriented python, and to make a program as generic as possible.

That's why I've used two main classes to represent the individuals used for my tests and the population.



The individual has two main properties :

- His "key" : contains the representation of the individual, the potential solution of the problem
- His fitness : the corresponding fitness, of that solution to the problem

For both the ES and the AG, the main information, which is the representation of the potential solution, is stored in the key.

This key, for the Ackley individual is represented the following way :

`[ [x(1), 'x'], … , [x(30), 'x'], [σ(1), 'sigma'], … , [σ(30), 'sigma'] ]`

The key is stored in a python list (easy to manipulate), containing tuples (value, type). I've provided the type of the value within the tuples to allow any interpretation of the key in future algorithm implementations, in function of the type provided.

NB : as my mutation and recombination algorithm don't use the binary or real string array representation, I don't explain this technic here. Please find more in the "Genetic Algorithm" section.

1.3. Fitness function

The problem is an optimisation problem : we basically want to find the minimum of the Ackley function for the chosen parameters.

The problem is similar to the precedent problem we studied : the function is positive and we want to get as close as possible to 0. That's why I've chosen the same fitness function, which gets higher when the result of the function gets lower.

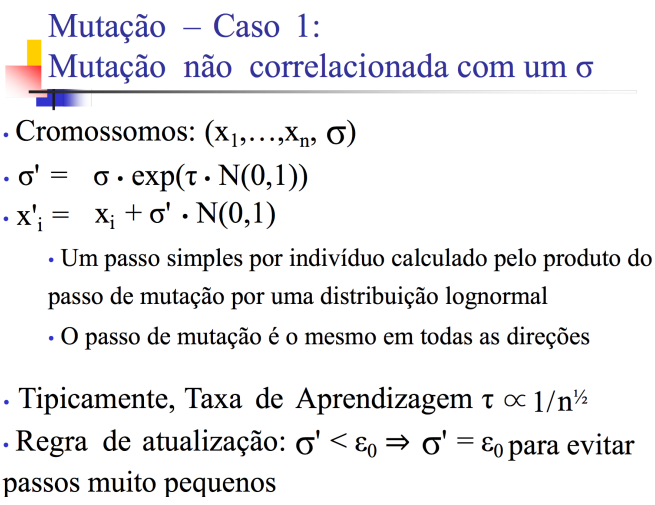
Fitness(x) = 1 ÷ (1+f(x)) with x an individual, and f(x) the result of the Ackley function for that individual.

1.4. Mutation operator

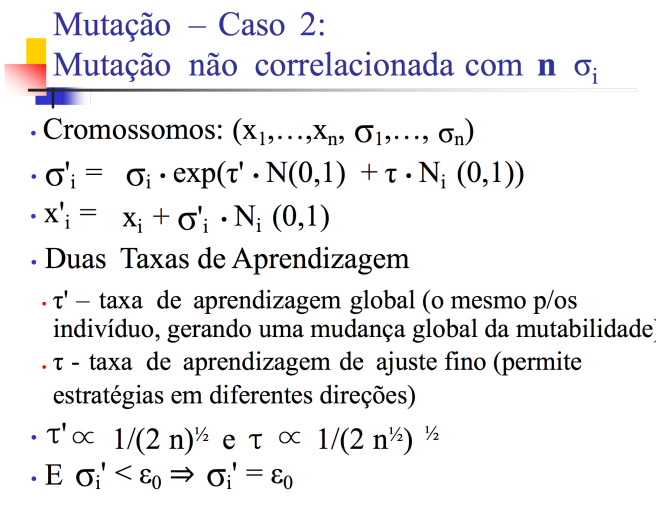
As mutation operator, I've decided to implement two algorithms from the course slides:

- 1LR1S (1 learning rate, 1 sigma) : correspond to the mutation non-correlated, with σ
- 2LRNS (2 learning rates, n sigmas) : correspond to the mutation non-correlated, with n σ(i)

Recall :



1LR1S



2LRNS

Please find here some precisions about the algorithms i pseudo-code :

- 1LR1S

```
father <- population.father
dimension <- d

for i in range(0, λ) do:
    global_step_size <- τ × 𝒩(0,1)
    list_sigma <- []
    list_xi <- []

    for j in range(0, d) do:
        σ <- father.σ(i)
        σ  <- σ × exp(global_step_size)
        list_sigma.append(σ)
        x(i) <- father.x(i) + σ × 𝒩(0,1)
        list_xi.append(x(i))
    endfor

    newkey <- list_xi + list_sigma
    new_individual <- AckleyIndividual(newkey)
    population.elitist_storage(new_individual)
endfor
```

Father is the only individual contained by population at this step

In the case of 1 σ, I still use the n σ(i) structure for the key, but every σ(i) has the same value (method uniformise_sigma() in Individual class).

#Elitist storage keeps only the μ best children, the selection is realised that way.

- 2LRNS

```
father <- population.father
dimension <- d

for i in range(0, λ) do:
    global_step_size <- τ × 𝒩(0,1)
    list_sigma <- []
    list_xi <- []

    for j in range(0, d) do:
        local_step_size = τ'' × 𝒩(0,1)
        σ <- father.σ(i)
        σ  <- σ × exp(global_step_size + local_step_size)
        list_sigma.append(σ)
        x(i) <- father.x(i) + σ × 𝒩(0,1)
        list_xi.append(x(i))
    endfor

    new_key <- list_xi + list_sigma
    new_individual <- AckleyIndividual(new_key)
    population.elitist_storage(new_individual)
endfor
```

This time every σ(i) is different.

These algorithms can be checked in the population class.

1.5. Recombination operator

I've chosen to implement 3 different recombination operators :

- Best : very basic. Just choose the best from the μ children to be the new father
- Intermediate : calculates the average of every x(i) and σ(i) from the children population to make a new father
- Weighted : calculates the **weighted** average of every x(i) and σ(i) from the children population to make a new father (weights are applied in function of fitness)

Here are their pseudo-codes :

- Best (hahaha)

```
father <- population.best
```

- Intermediate

```
new_key <- []
for (individual, fitness) in population do:
    for (value, type) in key do:
        sum_value_to_component_at_index_i(sumlist)
    endfor
endfor
new_key <- divide_each_component_by_μ(sumlist)
new_father <- AckleyIndividual(new_key)
```

- Weighted

```
new_key <- []
for (individual, fitness) in population do:
    for (value, type) in key do:
        sum_(value * fitness)_to_component_at_index_i(sumlist)
    endfor
endfor
new_key <- divide_each_component_by_fitness_sum(sumlist)
new_father <- AckleyIndividual(new_key)
```

NB : These pseudo-codes are really a simplified view of what I've coded. And to be honest, what I've coded isn't the most optimal way to do it ;).

1.6. Precisions

Stop condition :

Max fitness or max iteration number reached

General parameters values :

The user chooses a number n.

- $\lambda = 8 \times n$
- $\mu = \lambda \div 4$
- $\tau = 1 \div \sqrt{(n)}$
- $\tau' = 1 \div \sqrt[4]{(n)}$

Sigma Boost :

A sigma boost special nitro YOLO swag mode can be enabled. It resets every value of σ(i) with a random float between 0 and 1 every time the father isn't better than the previous father 5 consecutive times.

The aim of that mode is to avoid staying trapped in a local minimum.

1.7. Results analysis

All experimental results can be consulted in doc > results_analysis.

10 tests are effectuated for each parameters set.

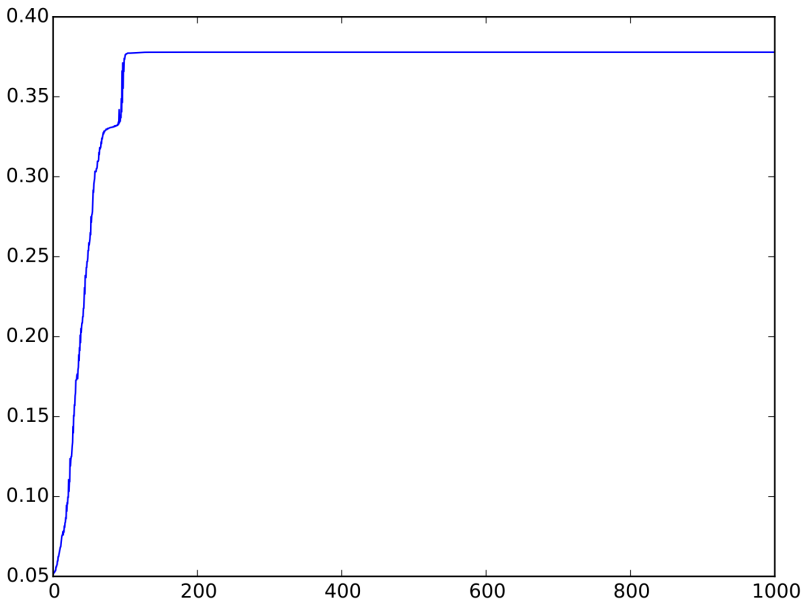
Mutation comparison :

- max iteration number = 5000
- stop fitness = 0,95
- recombination method = intermediate

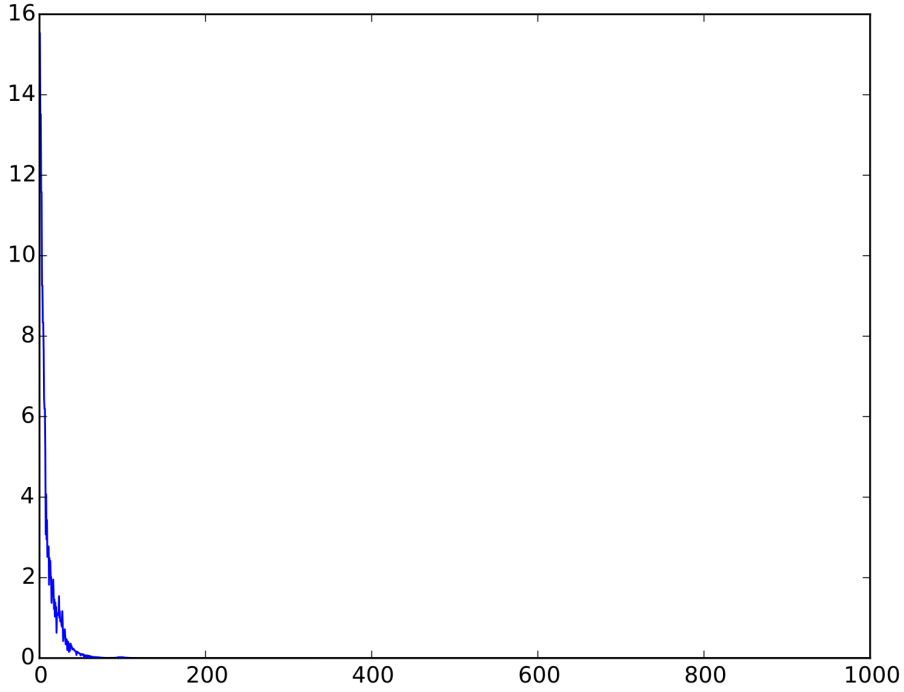
Mutation method	1LR1S	2LRNS
Average final iteration	5000	5000
Standard Deviation - Iterations	0	0
Average final fitness	0,2194229007767	0,3850738084271
Standard Deviation - Fitness	0,0304187948519112	0,091949708108933

Conclusion : These values lead us to think that the 2LRNS mutation is more efficient than the 1LR1S.

Low final fitness : We can see that the final fitness, in both cases, is very low. This can be explained thanks to the related curves :



Evolution of the max fitness across time



Evolution of the average σ across time

Interpretation : In my basic implementation, I didn't implement any minimal mutation step, that's why the ES very often ends in a local minimum.

Recombination comparison :

- max iteration number = 5000
- stop fitness = 0,95
- mutation method = 2LRNS

Recombination method	Intermediate	Weighted
Average final iteration	5000	5000
Standard Deviation - Iterations	0	0
Average final fitness	0,3850738084271	0,4490736500987
Standard Deviation - Fitness	0,091949708108933	0,185433608794481

I've not included the "best" recombination to the experiments because didn't founded it as interesting these two.

Conclusion : weighted recombination gives better results than the intermediate one.

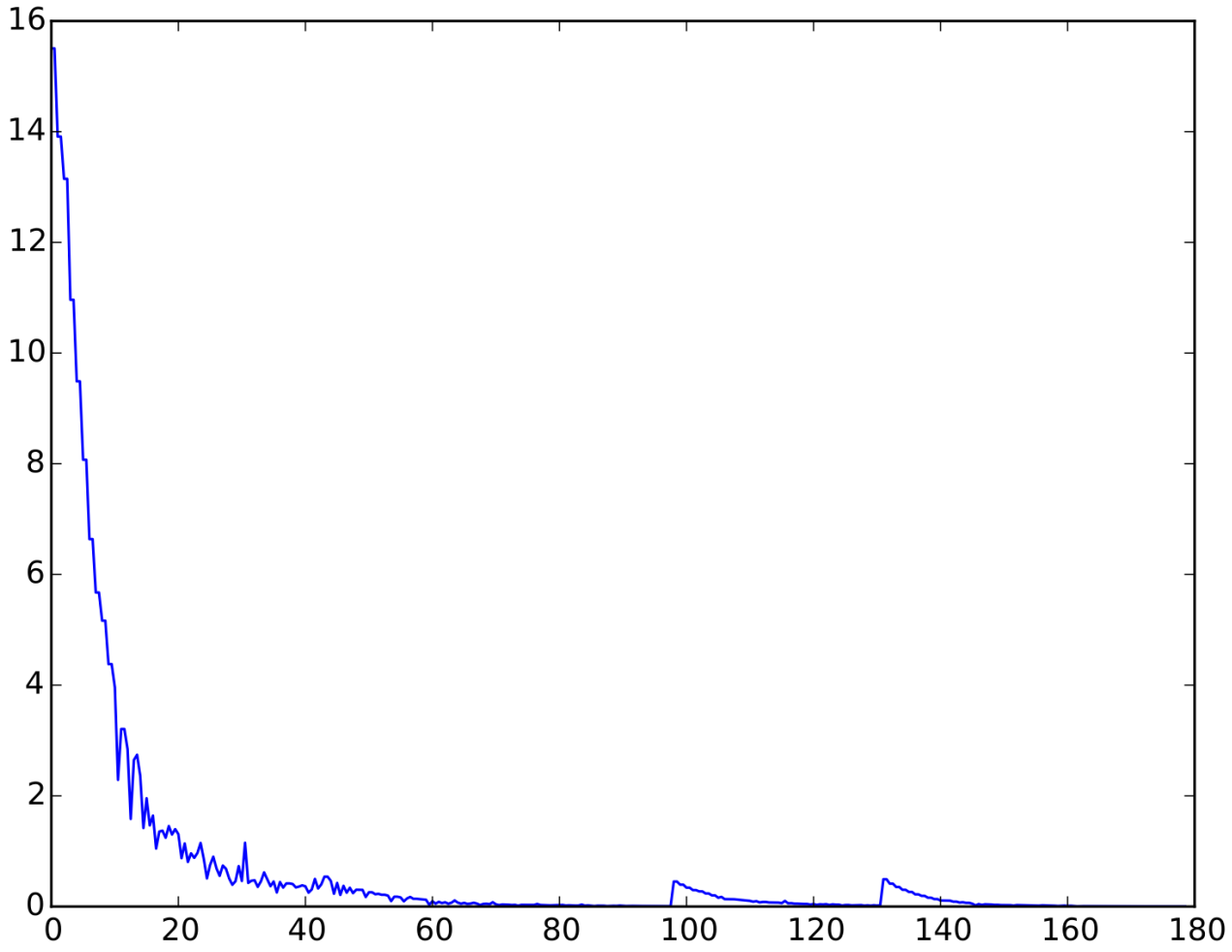
Note : Same local minimum problem.

The solution : sigma boost

- Regra de atualização: $\sigma' < \varepsilon_0 \Rightarrow \sigma' = \varepsilon_0$ para evitar passos muito pequenos

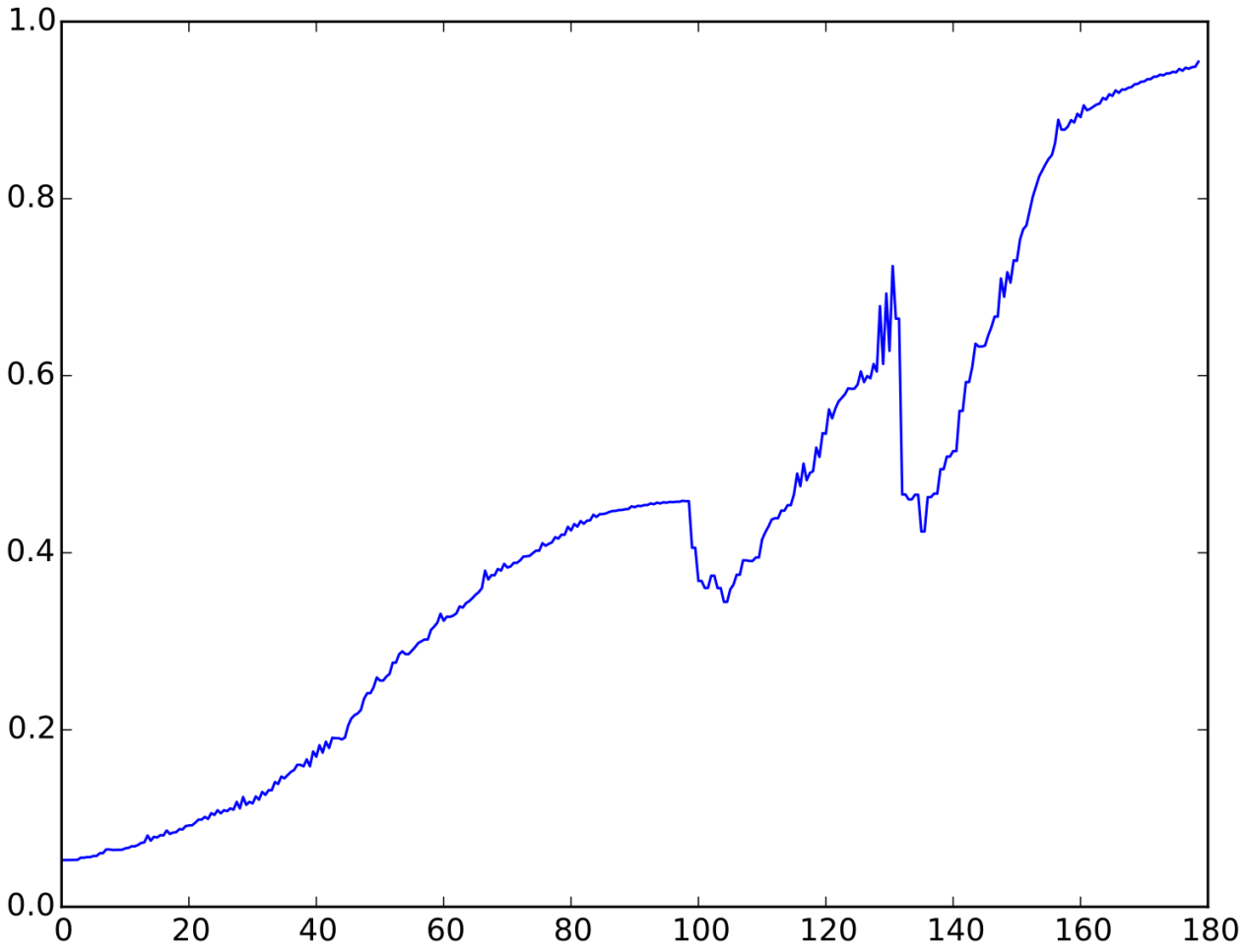
As advised in the course slide, actualising the mutation step is the solution to avoid being trapped by local minimums. That's why I've implement a feature I've called "sigma_boost()", which resets all σ(i) with the value between 0 and 1 every 5 iteration without fitness improvements.

This is very visual in the following curves :



Evolution of the average σ across time

This curve shows very well the initial natural evolution of the average sigma of the individual, followed by two small increases, caused by the sigma_boost function.



Evolution of the max fitness across time

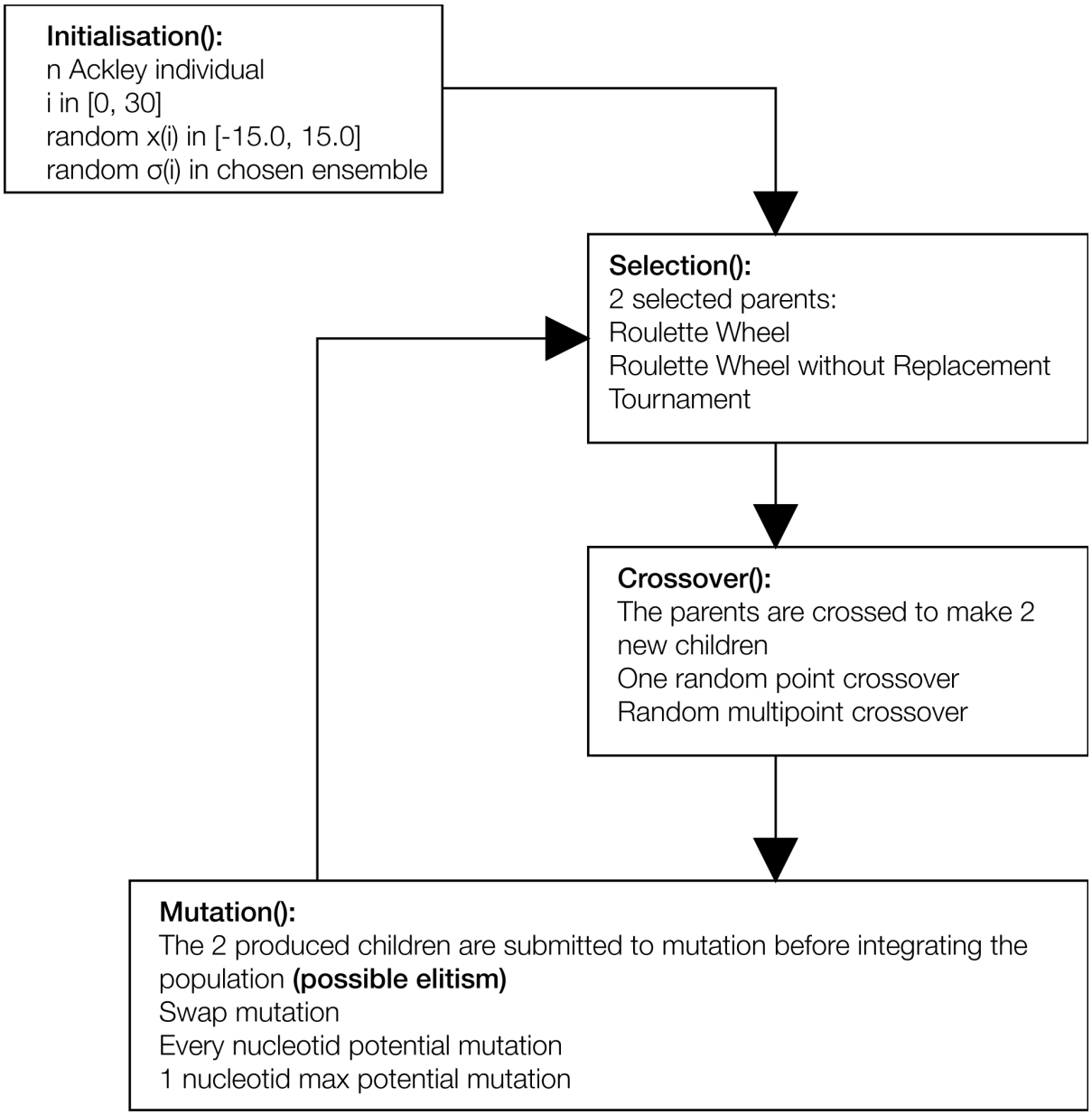
And this curve shows the influence this got on the maximal fitness, which doesn't stay trapped in a local minimum, and succeed to exceed 0,95 in a very good number of iterations.

Results with sigma_boost (mutation = '2LRNS', recombination = 'weighted', max iteration number = 5000, stop fitness = 0,95)

Experiment :	Without sigma_boost	With sigma boost
Average final iteration	5000	218,4
Standard Deviation - Iterations	0	47,2962295889782
Average final fitness	0,4490736500987	0,9512961754085
Standard Deviation - Fitness	0,185433608794481	0,00117278975620484

2. Genetic Algorithm

2.1. Schematic representation of the used algorithm

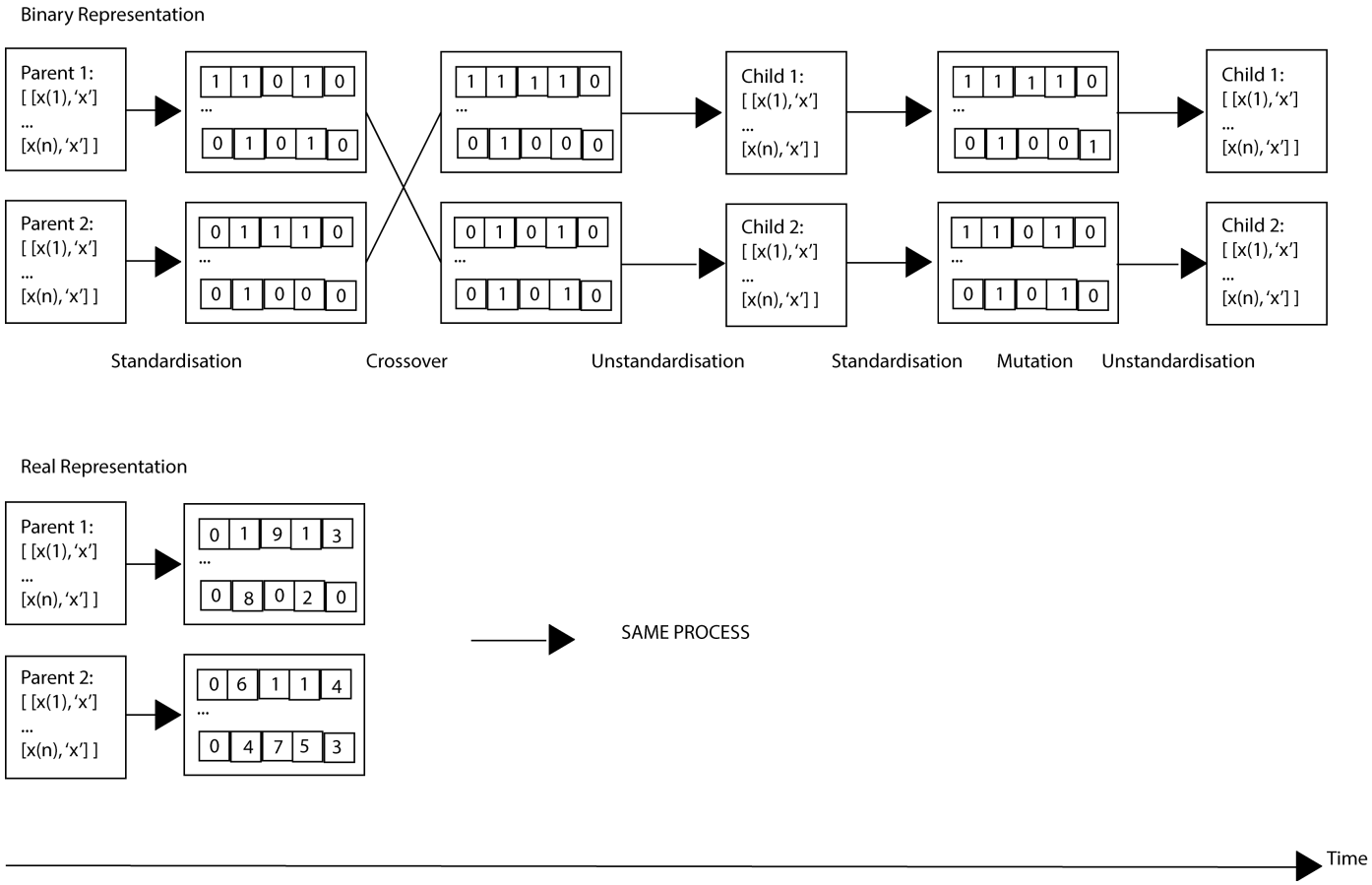


2.2. Individuals representation

When stored in the population, the individual representation is the same as seen in the section 1.2.

That's not the case during crossover and mutation, during which the individuals are standardised : that's the binary or the real standardisation.

Here's a schematic representation of the process over the time :



Important precision :

In order to standardise the float number used as x(i) component, in both Binary and Real representation, I've multiplied before standardisation these members by 1000 to get for example for a x(i) equals to 0.985 a standardised number from the integer 985.

I've made this choice before knowing the **floating point representation** and didn't have time to modify it after **even if I really wanted to implement it** (maybe next time ?!).

The multiplication by 1000 choice as been lead by the fact that in the precedent project subject, the number of significative numbers after comma were equals to 3. As this implementation is adapted from that one, I didn't change these functions.

These standardisation functions can be consulted in the AckleyIndividual class.

2.3. Fitness function

Cf section 1.3. , the fitness function is the same.

2.2. Selection algorithms

I've decided to implement the 3 following selection algorithm :

- Roulette Wheel algorithm : very famous algorithms, every individual has a chance proportional to his fitness to be selected.
- Roulette Wheel algorithm without replacement : while my basic roulette wheel algorithm has the possibility to choose two times the same individual, this roulette algorithm without replacement doesn't allow the same individual to be picked two times for the same children generation.
- Tournament algorithm : also very famous, the two individuals with the best fitness are picked to generate the children.

Pseudo-code details:

- Roulette Wheel

```
position_list <- []
aggregat <- 0
sum_fitnesses <- population.sum_fitnesses

for (individual, fitness) in population do:
    agregat += fitness / sum_fitnesses
    position_list.append((individual, agregat))
endfor

pick <- random.uniform(0, 1)

for (individual, position) in sorted(position_list) do:
    if position > pick:
        break
    endif
endfor

return individual
```

Each individual occupies a position between 0 and 1 on the roulette. A random number is picked between 0 and 1, and the corresponding individual is returned

- Roulette Wheel without replacement

A first individual is picked with the roulette selection and removed from the population (but keeped in a class variable). Then a second individual is picked using the same method. They are reincorporated from the class variable into the population before the crossover and mutation are processed.

- Tournament

The algorithm is quite the same : the best individual is picked and removed from the population two times, and reincorporated before crossover and mutation.

2.3. Crossover operator

I've implemented 2 crossover operators :

- One random point : a random crosspoint is picked and the two parents are crossed
- Random multipoint : for each nucleotide in the string array, the algorithm randomly choose for which parent nucleotide will be used in which child

I've chosen not to implement fixed crosspoint, because I was thinking that a random crosspoint is more funny and more effective.

Pseudo-code details:

- One random point

```
standardparent1 <- parent1.getstandardised
standardparent2 <- parent2.getstandardised

for i in range(0, len(standardparent)) do:
    string1, size1, min_position1, max_position1 = standardparent1
    string2, size2, min_position2, max_position2 = standardparent2

    crosspoint <- random.randint(min_position, max_position)

    string1 <- string1[0:crosspoint] + string2 [crosspoint:]
    string2 <- string2[0:crosspoint] + string1 [crosspoint:]

    child1.append(string1)
    child2.append(string2)

child1 = getunstandardised(child1)
child2 = getunstandardised(child2)

child1 = AckleyIndividual(child1)
child2 = AckleyIndividual(child2)

```

every standardised returned is a list containing n tuples with the following structure

(standard_string, size, min_position, max_positon)

once again, this pseudo-code doesn't exactly stick to the actuel implemented code

- Random multipoint

```
standardparent1 <- parent1.getstandardised
standardparent2 <- parent2.getstandardised

for i in range(0, len(standardparent)) do:
    string1, size1, min_position1, max_position1 <- standardparent1
    string2, size2, min_position2, max_position2 <- standardparent2

    for i in range(min_position, max_position) do:
        case <- random.randint(1, 2)
        if case == 1:
            newstring1 += string1[i]
            newstring2 += string2[i]
        elif case == 2:
            newstring1 += string2[i]
            newstring2 += string1[i]
    endfor

    child1.append(string1)
    child2.append(string2)

endfor

child1 = getunstandardised(child1)
child2 = getunstandardised(child2)

child1 = AckleyIndividual(child1)
child2 = AckleyIndividual(child2)

```

same comments

2.4. Mutation operator

Three mutation operators are implemented in my project :

- Swap mutation : chooses randomly two nucleotide to switch
- Every nucleotide potential mutation : every nucleotide can mutate into another value
- 1 nucleotide max potential mutation : maximum 1 nucleotide by chromosome can mutate by generation

The mutation probability can be set by the user, or randomly chosen, between 1 ÷ (size of the string array) and 1 ÷ (size of the population)

Pseudo-code details:

- Swap mutation

```
standardchild <- child.getstandardised()
mutated_child <- []

for (string, size, min_position, max_position) in standardchild do:
    mutation_probability <- set_probability()
    a <- 0
    b <- 0
    while a >= b, do:
        a <- random.randint(min_position, max_position)
        b <- random.randint(min_position, max_position)
    endwhile
    string <- string[0:a] + string[b] + string [a+1:b] + string[a] + string[b+1:]
    mutated_child.append(string)
endfor

mutated_child <- getunstandardised(mutated_child)
mutated_child <- AckleyIndividual(mutated_child)

```

- Every nucleotide potential mutation

```
standardchild <- child.getstandardised()
mutated_child <- []

for (string, size, min_position, max_position) in standardchild do:
    mutation_probability <- set_probability()

    for i in range(min_position, max_position) do:
        pick <- random.uniform(0, 1)
        if pick < mutation_probability do:
            string[i].mutate
        endif
    endfor

    mutated_child.append(string)
endfor

mutated_child <- getunstandardised(mutated_child)
mutated_child <- AckleyIndividual(mutated_child)

```

- 1 nucleotide max potential mutation

Same global principle as the every nucleotide mutation, with a difference, the pick is done just once, and if a mutation has to be applied, the nucleotide to apply it is randomly chosen.

2.5. Precisions

Stop conditions :

Max fitness or max iteration number reached

Elitism :

The user can choose to be elitist during the storage process by indicating a maximal population superior to 0. If equals to 0, there is no maximal population and every individual will be stored.

Initialising individuals number :

set by the user.

2.6. Results analysis

2.6.1. Methodology

As several algorithms are implemented for each part of the process, I wanted to find the best algorithm for each of these steps.

That's why I've made a great series of measures, with only 1 parameter changing every time.

This series can be consulted in the doc > results_analysis file.

2.6.2. Results

First question: What is the best mutation algorithm ?

Parameters:
- Binary representation
- Random one point crossover
- Roulette selection
- Random mutation probability
- Stop fitness = 0,95
- Max iterations = 10000
- Max population = 50
- Initial population = 100

Experiment:	Swap	Every Nucleotide	One Nucleotide
Average final iteration	10000	10000	9381,7
Standard Deviation - Iterations	0	0	1004,16953526561
Average final fitness	0,0767315822934	0,1110245178581	0,5125264477888
Standard Deviation - Fitness	0,00574339148985678	0,00596967440900399	0,303480666629505

Hypothesis: One nucleotide mode seems to be the best mutation algorithm.

Question: Will we get better results with real representation ? Will it confirm that this mutation algorithm is the best ?

Parameters:
- Real representation
- Random one point crossover
- Roulette selection
- Random mutation probability
- Stop fitness = 0,95
- Max iterations = 10000
- Max population = 50
- Initial population = 100

Experiment:	Swap	Every Nucleotide	One Nucleotide
Average final iteration	10000	10000	9118,9
Standard Deviation - Iterations	0	0	517,244504830914
Average final fitness	0,05755720708412	0,3187107357218	0,9518417898622
Standard Deviation - Fitness	0,00105276196483477	0,0247942321550332	0,00250756179675331

Conclusion : Results in real representation are a bit better than in binary representation. These measures also confirm that the best mutation algorithm seems to be the “one nucleotide maximum potential mutation”.

Question: What’s the best crossover algorithm ?

Parameters:
- Real representation
- One nucleotide mutation
- Roulette selection
- Random mutation probability
- Stop fitness = 0,95
- Max iterations = 10000
- Max population = 50
- Initial population = 100

Experiment:	Random single point crossover	Random Multipoint crossover
Average final iteration	9118,9	7413,2
Standard Deviation - Iterations	517,244504830914	1171,61757128055
Average final fitness	0,9518417898622	0,96345776513
Standard Deviation - Fitness	0,00250756179675331	0,0116546880858031

Conclusion: Results with a random multipoint crossover are better than results with a single point crossover with the same parameters. We keep it for the following experiments.

Question: What's the best selection algorithm ?

Parameters:
- Real representation
- One nucleotide mutation
- Random Multipoint Crossover
- Random mutation probability
- Stop fitness = 0,95
- Max iterations = 10000
- Max population = 50
- Initial population = 100

Experiment:	Roulette	Roulette WR	Tournament
Average final iteration	7413,2	6207,5	5508,5
Standard Deviation - Iterations	1171,61757128055	570,333089810975	1296,33631181624
Average final fitness	0,96345776513	0,9563897784251	0,988681892218
Standard Deviation - Fitness	0,0116546880858031	0,00709735671973752	0,00797977474334966

Conclusion: Results with a tournament selection are way better than the others. It seems to be the best selection algorithm. We keep it for the following experiments.

Question: What's the best mutation probability ?

Parameters:
- Real representation
- One nucleotide mutation
- Random Multipoint Crossover
- Tournament selection
- Stop fitness = 0,95
- Max iterations = 10000
- Max population = 50
- Initial population = 100

Mutation probability:	0,5	0,25
Average final iteration	10000	10000
Standard Deviation - Iterations	0	0
Average final fitness	0,1586960502145	0,6443164590477
Standard Deviation - Fitness	0,00724106890191724	0,129470748508046

Mutation probability:	0,1	0,05	0,025
Average final iteration	5345,3	4117,1	6258,2
Standard Deviation - Iterations	1295,72031018358	428,543644866606	1642,7117012225
Average final fitness	0,9755002358449	0,974819087544	0,9860072381483
Standard Deviation - Fitness	0,0143962475610779	0,0125944666094754	0,00871392381986359

Conclusion: Results are getting pretty good with a mutation probability close to 0,05.

2.6.3. Analysis

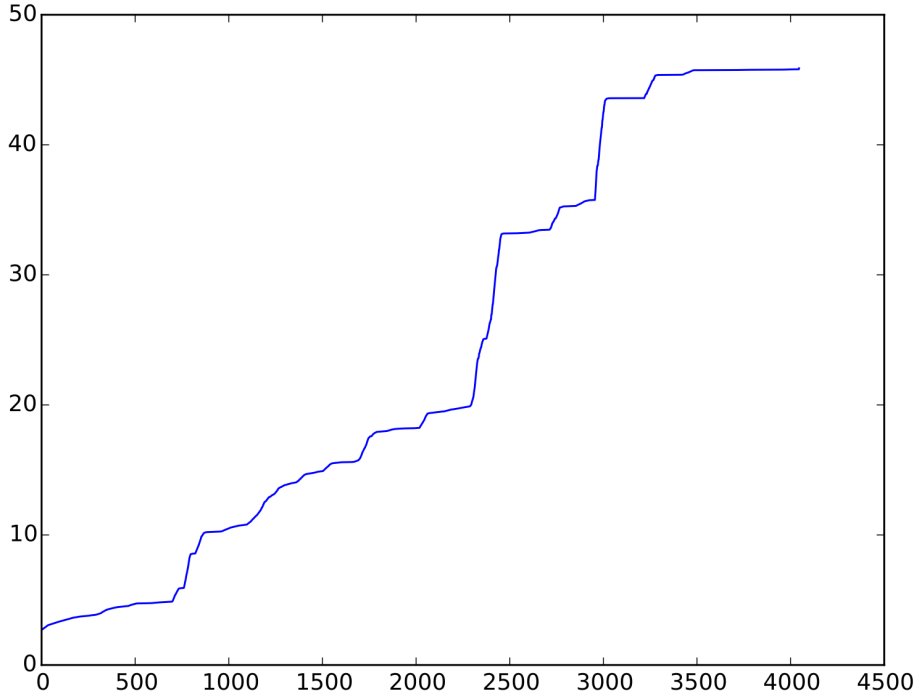
After all these measure, we can conclude that the best parameter set for my proposed Genetic Algorithms implementation would be :

- Tournament selection: helps to avoid the apparition of a “super-individual”
- Random Multipoint crossover: good genetic shuffling
- 1 Nucleotide maximum potential mutation
- A mutation probability close to 0,05: too much mutations doesn’t allow to reach THE solution

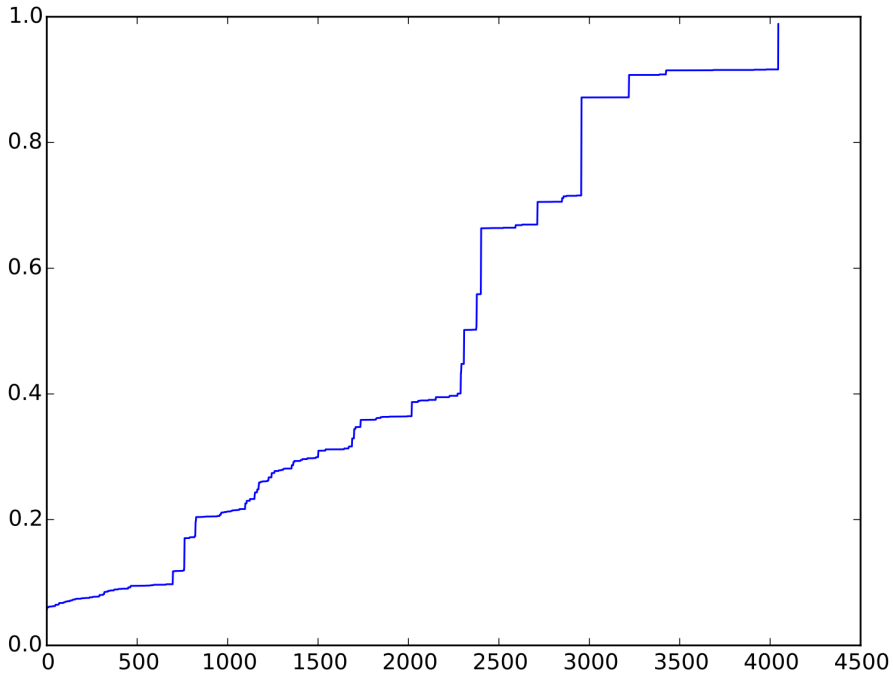
I didn’t make any tests on :

- Initial population : because it is obvious that the more individuals you use to initialise your population, the more the base individuals will have a good global fitness
- Elitism : because, again, it’s obvious that without elitism, the fact that you store every individual will rapidly slow down your program (you will have to evaluate each of them for the roulette selection) or be useless (with my implementation of the tournament selection, just the two best individuals are chosen, so the x others would be useless).

Here's the appearance of the curves with the optimal parameters set:



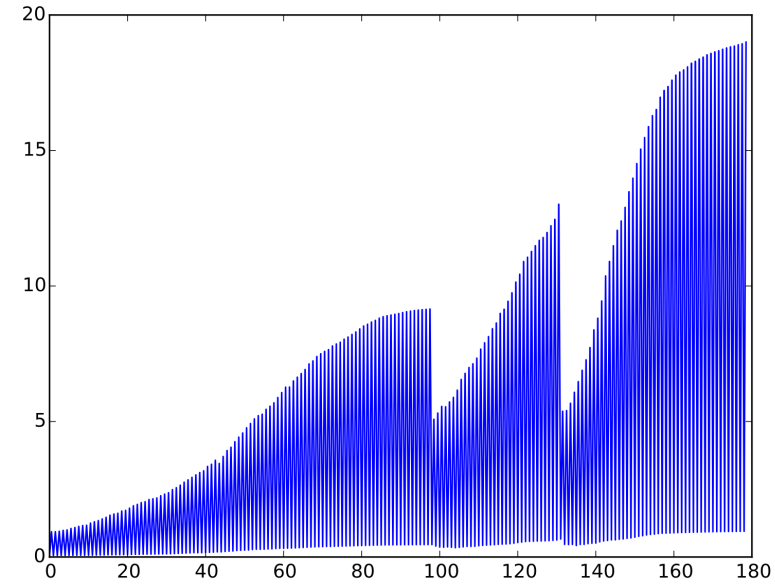
Evolution of the sum of every fitnesses across time

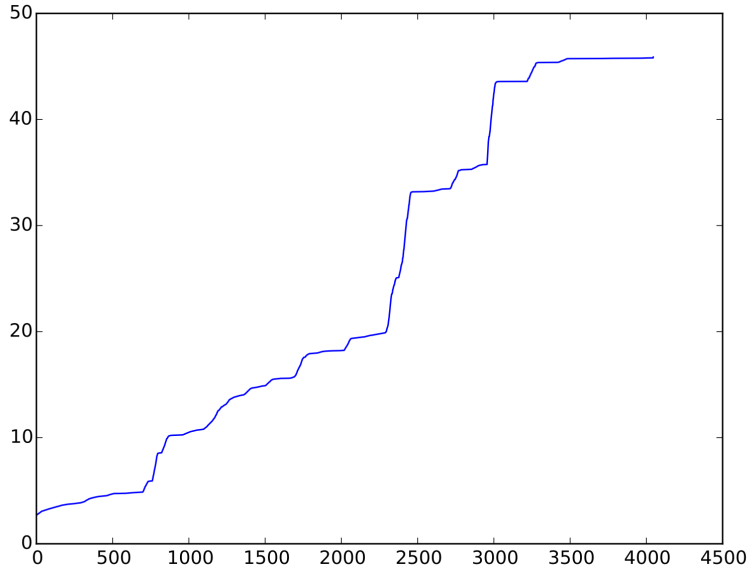


Evolution of the best individual's fitness across time

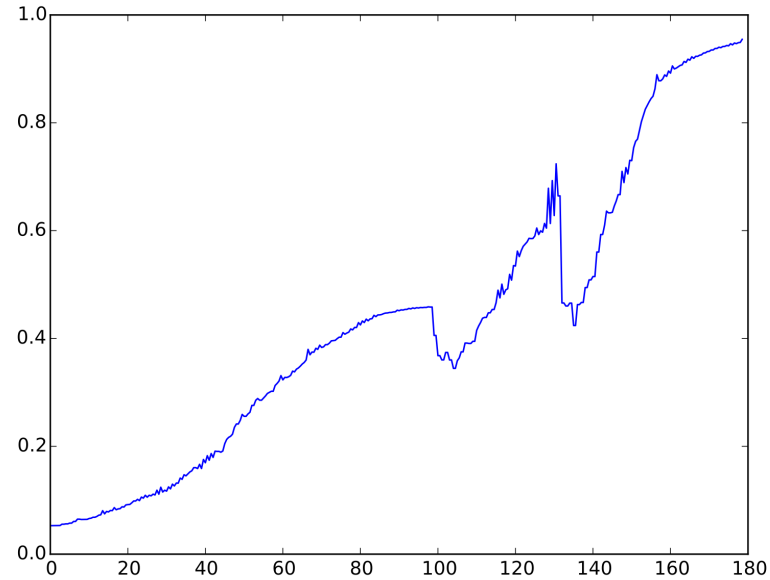
3. GA vs ES

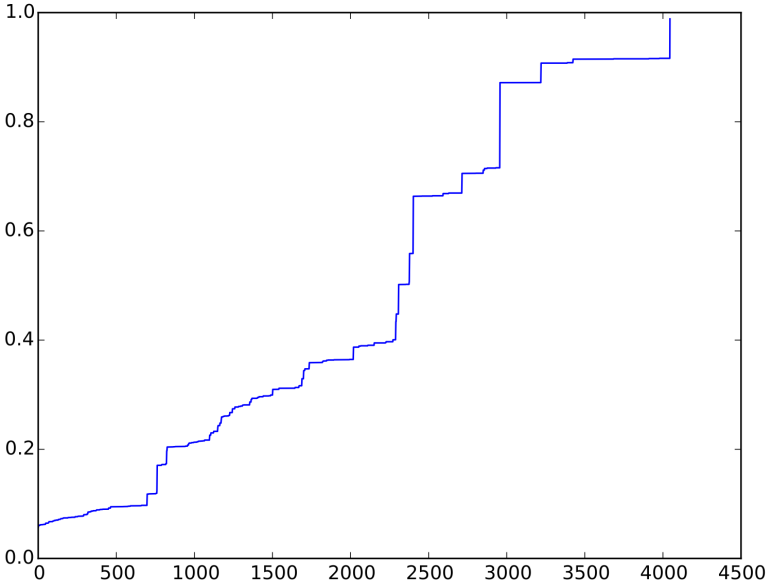
Here is a little comparison between curves from GA and ES.





Comparison on the sum of every fitnesses between ES (left) and GA (right)





Comparison on the maximal fitness in population between ES (left) and GA (right)