POLITECNICO DI TORINO

Master's Degree in Computer Engineering
Artificial Intelligence and Data Analytics

Master's Degree Thesis

Design and development of a general purpose evolutionary algorithm fuzzer and optimizer

Supervisors

Prof. Giovanni SQUILLERO

Prof. Alberto TONDA

Candidate

Marco SACCHET

April 2024

Abstract

The aim of this thesis is to develop a comprehensive set of components for an evolutionary tool intended to serve as a foundation for a fuzzer or optimizer, providing all the necessary tools to generate and evolve solutions to a problem presented by the user, by studying self-adaptation and developing a self-adaptive evolutionary algorithm.

This algorithm differs from a non-self-adapting vanilla evolutionary algorithm for the ability to allow individuals to die of old age, the existence of an elitist subset of individuals with high fitness, the option to tweak the reward given to operators, and the ability to behave differently according to the current state of the evolution. The study also covered the auto-adaptation of the operator's selection based on previous results and sigma adaptation, following the works of Davis (1991) and De Jong (1975).

The study on operator selection shows a clear advantage in using such a technique with multi-armed bandits algorithm such as Successive Elimination [Slivkins (2019)]. This algorithm, using the reward history of each operator, performs an adaptive exploration of the space of the possible operators. It greatly reduces the number of calls to inefficient operators that fail to generate valid offspring.

The study on sigma adaptation focuses on how the algorithm is able to balance the different phases of exploration and exploitation by managing mutation's strength, introducing a user-defined temperature, and the notion of proximity related to a fitness function. The results show that implementing directly in the genetic operators the concept of mutation's strength, making them capable of acting differently based on the current state of evolution, leads to more efficient executions and more balance between exploration and exploitation phases.

Overall, this thesis showcases the process of designing, implementing, and enhancing a self-adaptive evolutionary algorithm and the benefits of this algorithm compared to a non-self-adaptive one.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction to Evolutionary Algorithm and Computation

Not all those who wander are lost,
J. R. R. Tolkien

Since the beginning of its existence on Earth, humanity has looked to nature as a source of inspiration. This habit, which has stuck with humanity throughout its existence, has not vanished in the flow of time, and it's rooted inside us still today.

Inspiration from nature encompasses every field of human knowledge, and computer science is no exception. From the very beginning, the fathers of this discipline have looked to the human brain and nature to direct their efforts in that emerging field. So, it was just a matter of time before someone pointed its attention to the very foundation of life: evolution.

When this word is pronounced, the first thing that comes to mind is a name: the one of Charles Darwin. Although the indisputable impact that the work [1] of this man has had on the current understanding of nature and biology, the theory of evolution would not be the same without the work of Wallace [2], Weismann [3], and Mendel [4] (the only one not sharing an otherwise undeniable common passion for long and untamed beard).

It is probably with these names in his mind that George E.P. Box proposed the first evolutionary approach to an industrial problem [5]. Despite some sparse publications at the end of the 1950s that lightly touched this topic, the 1960s are commonly set as the start of studies on evolutionary computation, thanks to the works of John Holland, Lawrence Fogelm, Ingo Rechenberg, and Hans-Paul Schwefel.

But what is evolutionary computation? A precise definition is yet to be found, as the exact boundaries of its applications.In general, it is a branch of computer science that studies evolutionary algorithms. They apply the same concept of natural selection as an optimization process to optimize a candidate solution to a given problem through recombination and mutations. These produce random variations: some nefarious and others beneficial, according to an objective evaluation. Only the beneficial ones are propagated. It is straightforward to see similarities between evolutionary computation and natural selection, and these parallelisms continue also in the used terminology.

1.1 Lexicon

Like natural selection, evolutionary algorithms work on a population of individuals, bringing them to life, reproducing, and eventually dying. Here is a brief summary of the lexicon used in this branch of computer science.

The term **Population** refers to a set of individuals in a certain instant of time. An **Individual** is a single candidate solution for the given problem. Each individual contains a **Genome**, the totality of genetic components that form the individual. One genetic component is a **Gene**, namely the smallest element of the individual that can be subject to modifications. The specific position of a gene inside the genome is called **Locus** (plural: Loci). The different genes allowed to position in the same locus are called **Alleles**. It is impossible to define an univocal implementation of these terms in the algorithm because they are highly dependent on the nature of the problem or on the type of algorithm used.

Every individual in a population is evaluated according to the ability to solve the given problem. The name of this score is **Fitness**, and it is computed by a **Fitness Function**. The fitness function is problem-dependent and needs to be defined alongside it. Going back to the natural world, fitness represents the ability of an individual to survive in the environment. In the case of the evolutionary algorithm, the environment is the space where the problem solution lives, whereas the ability to survive is the distance between the individual and the solution.

Unlike the real world. where every individual follows his life cycle, in this kind of algorithm evolution happens through steps called **Generation**. On each generation, new **Offspring** are born and added to the population. Then, individuals less suited to survive in the environment (i.e. those with the lowest fitness) are removed from the population, and a new generation begins.

The reproduction of an individual is performed via **Operators**, special functions that operate on the genome of one or more individuals. Reproduction can happen in different ways: sexual and asexual.

In the first case, it's called **Recombination**. Here the offspring is generated from two or more individuals called **Parents**, from whom the offspring inherits a subset of their characteristics. If this recombination happens through exchanging genetic materials, it's called **Crossover** [6].

If the reproduction is asexual, the process is called **Replication**. In this case, an individual is copied. The term **Mutation** refers to the case when the offspring copied from the parent undergoes genomics modification, resulting in a different individual from the parent.

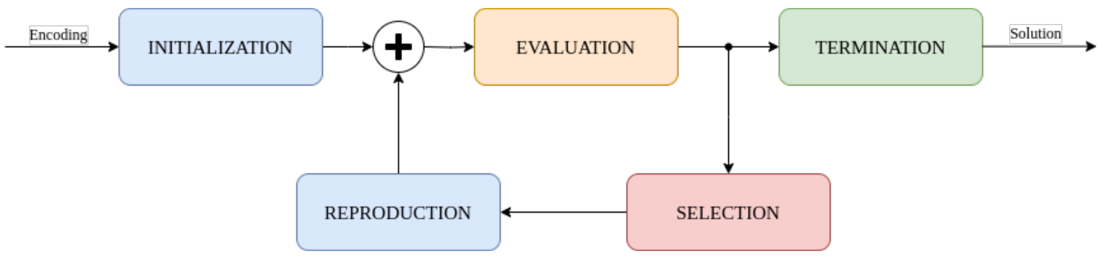


Figure 1.1: Block diagram of a generic evolutionary algorithm

1.2 Advantages

The basic idea behind the evolutionary algorithm is to develop a feasible solution to a problem without knowing a priori the best way to reach it. With this formulation, some parallelism to a random approach may arise. Nonetheless, over the past decades, EAs have shown better performance than pure random.

Broadening the view, it can be seen that evolutionary algorithms present a plethora of interesting features.

First of all, they provide an effective methodology for trying random modifications, without requiring previous ideas about the optimal solution. They present a better robustness with respect to Hill Climb algorithms; the use of a population of possible solutions, indeed, allows for the preservation of different types of solutions, reducing the risk of taking a dead-end street with an only-initially promising solution.

The possibility to implement different kinds of genetic operators, based on the specific structure of the genome, permits modifications to happen on large or small scale, with different probabilities. The existence of sexual reproduction, moreover, led to an efficient exploration of the search space, combining different characteristics from different solutions.

Finally, in more prosaic terms, evolutionary algorithms, once started, require no human intervention and offer an easy trade-off between the quality of the result and the computational load.

A non-comprehensive list of examples of use of evolutionary algorithms on practical applications could be found in [7].

1.3 Disadvantages

As everything coming from a human mind, also evolutionary algorithms are not perfect. There are some problems and difficulties embedded in the very structure of these algorithms that can drastically reduce their performances.

The fitness function has to be able to compute different fitness values for different solutions because small variations can induce large modifications in the population. High consideration must be paid in deciding which information is relevant and which can be discarded.

The problem of relevant and non-relevant information endure also with the individual itself. The way an individual is represented indirectly encodes the representation of the desired solution. So, if too much information is present inside the individual, the optimal solution may be excluded even before the algorithm starts searching for it. On the other side, if not enough information is present, the search space could increase in size, slowing down the process. The type of information stored inside the individual does not influence only the solution, but also the means to reach it. The genetic operators will act based on the solution structure.

If the problems presented until now are preventable or largely mitigateable with an accurate and thoughtful design, the premature convergence [8] problem is far more insidious and difficult to eradicate. In this case, the individuals of a population tend to share a restricted set of alleles, losing a large part of their genetic inheritance. This way all solutions tend to converge to a single point in the search space, halting evolution. If this problem occurs and no measures are taken to address it, the evolutionary algorithm will behave like an inefficient hill climber algorithm. There are no standard solutions for this problem, and workarounds need to be found on a per-case basis.

Chapter 2

Conceptual Description

There is a way out of every box,
a solution to every puzzle;
it's just a matter of finding it,
Captain Jean-Luc Picard

2.1 Byron Evolutionary Tool

The studies and development in this thesis were performed using Byron evolutionary tool. Byron, currently under development, is designed to be a Python package with all the needed tools to run an evolutionary algorithm. So it presents implementations of the concepts introduced in the previous chapter 1.1.

2.1.1 Population

The class Population contains a list of individuals belonging to that population, plus some general information regarding all of these. It presents the ability to order itself based on the individual's fitness and it exposes methods to retrieve general information about the composition of individuals genome. It also performs some end-of-the-line checks on the newly added individuals to avoid keeping invalid ones.

Thesis Development

This thesis led to the introduction of the concept of Age, that will be further discussed in the next paragraph. The population class now presents methods to manage the life-cycle of the individuals, including the possibility to die of old age. It can, in fact, not only manage the aging on a per individual basis, but also find the individuals above a certain age.

2.1.2 Individual

The class Individual is definitively more than just a genome's container. Indeed, this class contains an inner class to keep track of the individual's age and another for the individual's parents. Both of those are deeply integrated into the overall flow of execution.

The Age class keeps track of the age of the individual and also the moment of its creation. The management of the age is demanded to the Population class, but every individual is responsible to directly acting on its age.

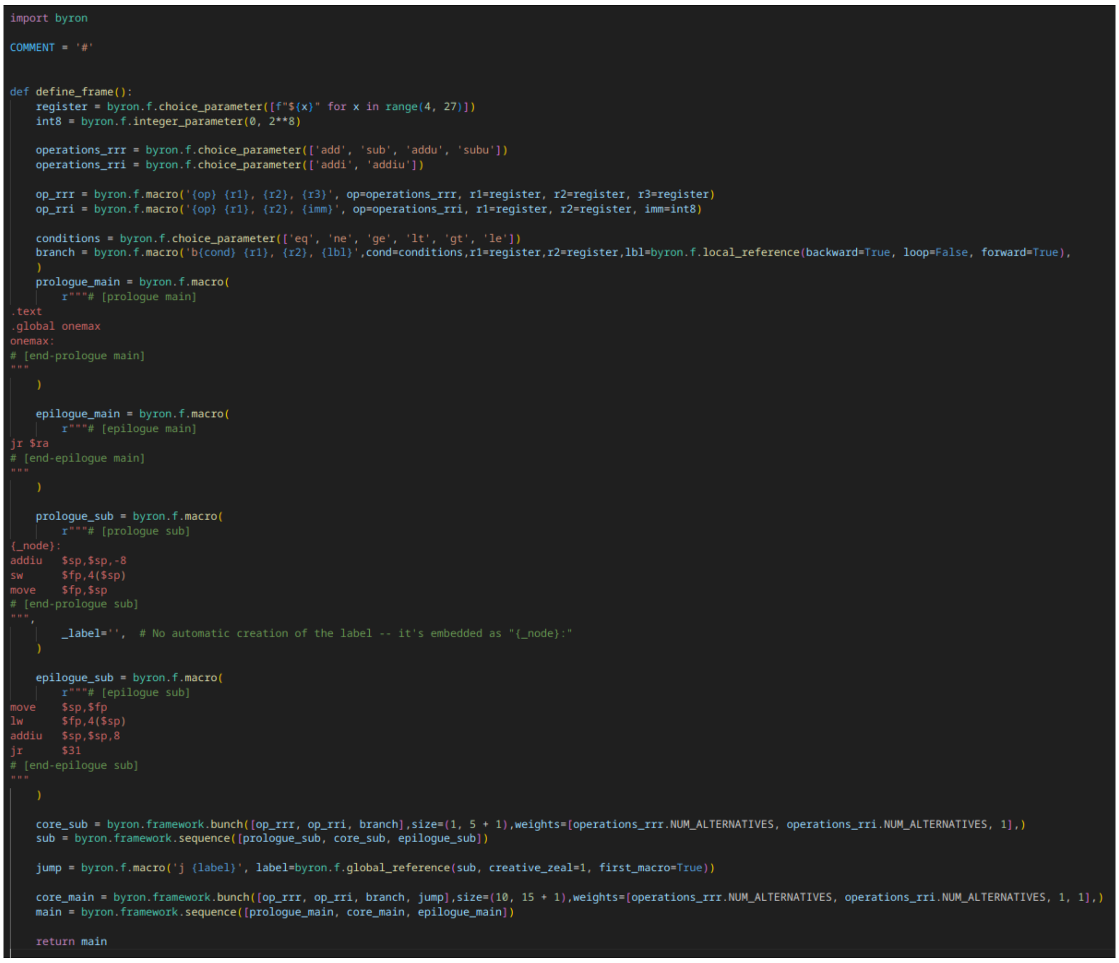
The Lineage class is used to maintain information about the individual's ancestors. This can be useful to track the evolution of a specific individual and to plot the family tree of the population.

The individual presents the ability to execute several checks on itself to find invalid components of the genome, to clone itself and to store its fitness. Moreover, it presents the ability to produce different types of representations of its genome, like as a forest or as a Linear Genetic Program (LGP).

2.1.3 Genome

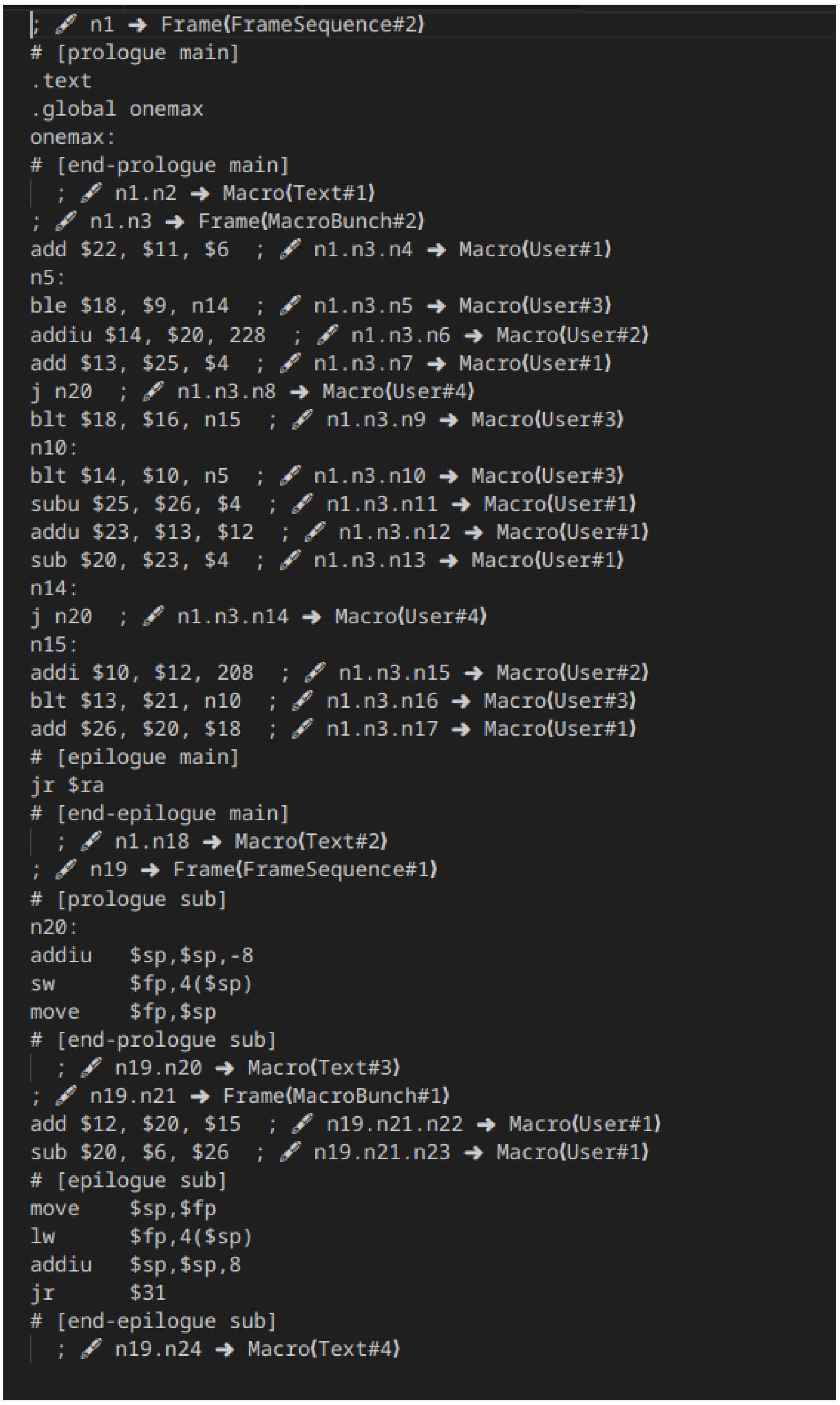
The genome is encoded as a weakly-connected multigraph. This structure is particularly useful in representing a series of computer instructions (i.e. code). The possibility to allow loops in the graph structure permits the implementation of function calls, for-loops and also recursion.

To allow a proper definition of the genes available in an individual, Byron offers a hierarchy of components. Those are parameter, macro and framework.



```python
import byron

COMMENT = '#'


def define_frame():
    register = byron.f.choice_parameter([f"${x}" for x in range(4, 27)])
    int8 = byron.f.integer_parameter(0, 2**8)

    operations_rrr = byron.f.choice_parameter(['add', 'sub', 'addu', 'subu'])
    operations_rri = byron.f.choice_parameter(['addi', 'addiu'])

    op_rrr = byron.f.macro('{op} {r1}, {r2}, {r3}', op=operations_rrr, r1=register, r2=register, r3=register)
    op_rri = byron.f.macro('{op} {r1}, {r2}, {imm}', op=operations_rri, r1=register, r2=register, imm=int8)

    conditions = byron.f.choice_parameter(['eq', 'ne', 'ge', 'lt', 'gt', 'le'])
    branch = byron.f.macro('b{cond} {r1}, {r2}, {lbl}',cond=conditions,r1=register,r2=register,lbl=byron.f.local_reference(backward=True, loop=False, forward=True),
    )
    prologue_main = byron.f.macro(
        r"""# [prologue main]
.text
.global onemax
onemax:
# [end-prologue main]
"""
    )

    epilogue_main = byron.f.macro(
        r"""# [epilogue main]
jr $ra
# [end-epilogue main]
"""
    )

    prologue_sub = byron.f.macro(
        r"""# [prologue sub]
{_node}:
addiu   $sp,$sp,-8
sw      $fp,4($sp)
move    $fp,$sp
# [end-prologue sub]
""",
        _label='',  # No automatic creation of the label -- it's embedded as "{_node}:"
    )

    epilogue_sub = byron.f.macro(
        r"""# [epilogue sub]
move    $sp,$fp
lw      $fp,4($sp)
addiu   $sp,$sp,8
jr      $31
# [end-epilogue sub]
"""
    )

    core_sub = byron.framework.bunch([op_rrr, op_rri, branch],size=(1, 5 + 1),weights=[operations_rrr.NUM_ALTERNATIVES, operations_rri.NUM_ALTERNATIVES, 1],)
    sub = byron.framework.sequence([prologue_sub, core_sub, epilogue_sub])

    jump = byron.f.macro('j {label}', label=byron.f.global_reference(sub, creative_zeal=1, first_macro=True))

    core_main = byron.framework.bunch([op_rrr, op_rri, branch, jump],size=(10, 15 + 1),weights=[operations_rrr.NUM_ALTERNATIVES, operations_rri.NUM_ALTERNATIVES, 1, 1],)
    main = byron.framework.sequence([prologue_main, core_main, epilogue_main])

    return main
```

Figure 2.1: An example of use of genes



```
; ✎ n1 → Frame(FrameSequence#2)
# [prologue main]
.text
.global onemax
onemax:
# [end-prologue main]
  ; ✎ n1.n2 → Macro(Text#1)
; ✎ n1.n3 → Frame(MacroBunch#2)
add $22, $11, $6  ; ✎ n1.n3.n4 → Macro(User#1)
n5:
ble $18, $9, n14  ; ✎ n1.n3.n5 → Macro(User#3)
addiu $14, $20, 228  ; ✎ n1.n3.n6 → Macro(User#2)
add $13, $25, $4  ; ✎ n1.n3.n7 → Macro(User#1)
j n20  ; ✎ n1.n3.n8 → Macro(User#4)
blt $18, $16, n15  ; ✎ n1.n3.n9 → Macro(User#3)
n10:
blt $14, $10, n5  ; ✎ n1.n3.n10 → Macro(User#3)
subu $25, $26, $4  ; ✎ n1.n3.n11 → Macro(User#1)
addu $23, $13, $12  ; ✎ n1.n3.n12 → Macro(User#1)
sub $20, $23, $4  ; ✎ n1.n3.n13 → Macro(User#1)
n14:
j n20  ; ✎ n1.n3.n14 → Macro(User#4)
n15:
addi $10, $12, 208  ; ✎ n1.n3.n15 → Macro(User#2)
blt $13, $21, n10  ; ✎ n1.n3.n16 → Macro(User#3)
add $26, $20, $18  ; ✎ n1.n3.n17 → Macro(User#1)
# [epilogue main]
jr $ra
# [end-epilogue main]
  ; ✎ n1.n18 → Macro(Text#2)
; ✎ n19 → Frame(FrameSequence#1)
# [prologue sub]
n20:
addiu   $sp,$sp,-8
sw      $fp,4($sp)
move    $fp,$sp
# [end-prologue sub]
  ; ✎ n19.n20 → Macro(Text#3)
; ✎ n19.n21 → Frame(MacroBunch#1)
add $12, $20, $15  ; ✎ n19.n21.n22 → Macro(User#1)
sub $20, $6, $26  ; ✎ n19.n21.n23 → Macro(User#1)
# [epilogue sub]
move    $sp,$fp
lw      $fp,4($sp)
addiu   $sp,$sp,8
jr      $31
# [end-epilogue sub]
  ; ✎ n19.n24 → Macro(Text#4)
```

Figure 2.2: A textual representation of the genes

Parameter

This is the smallest element of the genome that can be subject to modifications. Moreover, this component is the only having the ability to mutate itself without additional infrastructure. Keeping the example of code, a parameter could be the name of a variable, the plus sign in a sum, or else. There are different classes related to this component:

- integer / float parameter: it contains a number within a given range, and can mutate itself with the value range provided.

- choice parameter: it contains an element between a set of possible elements. When it mutates, a new element from that set is selected.

- array parameter: it contains a fixed length array of symbols.

- local / global structural parameter: it represents a node in the genome (in case of a local structural parameter, only the sibling [1] nodes are considered)

Macro

A macro is, by definition [9], a fragment of text with variable elements. A macro is more than a simple combination of parameters. It provides ways of encoding parameter information and means to connect different parameters in a single component with a specific syntactic through the insertion of fixed text. It can iteratively check the validity of its components. A macro could represent a single line of code, like a variable assignment, a conditional branch or else.

Frame

If parameters are single elements and macros are a way to syntactically group parameters, a frame is the component used to semantically organize macros. It can present a fixed sequence, an unordered group or a list of mutually exclusive macros.

[1]the successor nodes of the node's predecessor

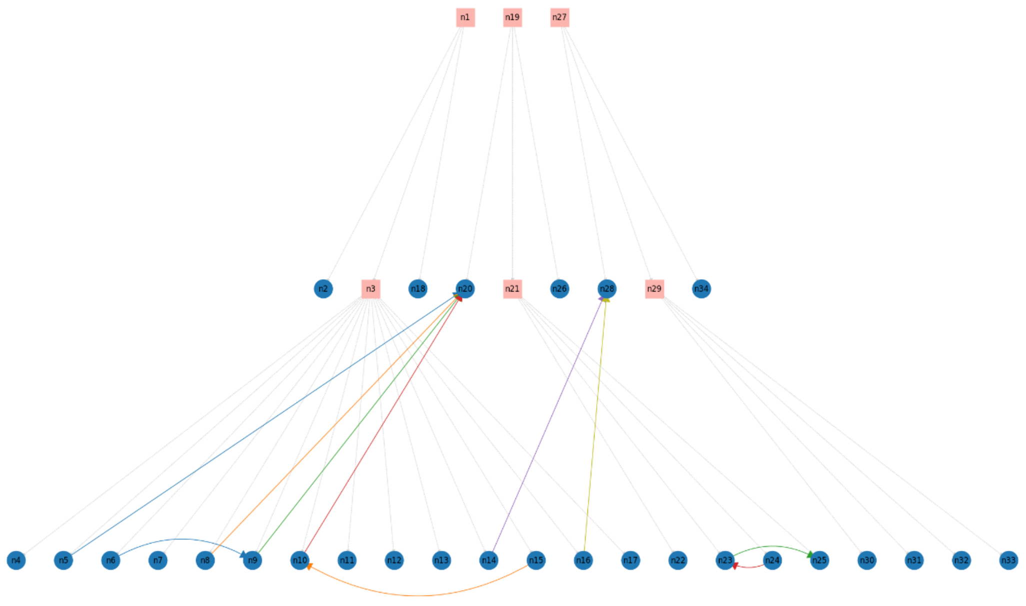


Figure 2.3: A representation of the genome as a forest

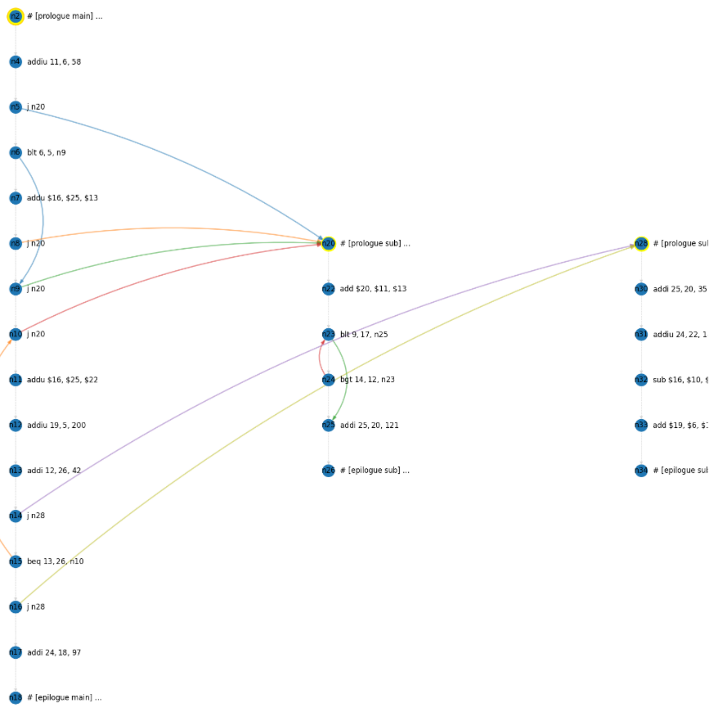


Figure 2.4: A representation of the genome as a Linear Genetic Program

Moreover, it is the only component able to contain another instance of itself. In other words, a frame can contain other frames. This causes that in the end the entire genome of an individual could be stored inside a single frame. A frame broads the context with respect to a macro. It could be an entire function, a subroutine, an exit sequence and, ultimately, the entire computer program.

2.1.4 Operators

The complex structure of an individual's genome heavily influences the complexity of operators. Operators implement both recombination and mutation, with specific implementations for Parameter, Frame and even a generic node in the graph.

Every time an operator is called, the goodness of its output is evaluated and logged. This way Byron is able to keep track of the overall performance of the operator. The monitored information are:

- Number of calls of the operator: how many time the operator was used by Byron
- Aborts: number of completely failure of the operator, i.e. every time no valid offspring were generated
- Offspring: how many valid offspring this operator created in total
- Failures: number of new individuals whose fitness was worse than the ones of their parents
- Successes: how many new individuals have a better fitness than at least one of their parents.

In Byron are implemented several genetic operators:

Single Parameter Mutation

It is the simplest operator possible: selects a parameter and makes it mutate.

Single Element Array Parameter

Given a parameter encoded as a list, it randomly selects an element of that list and mutates it.

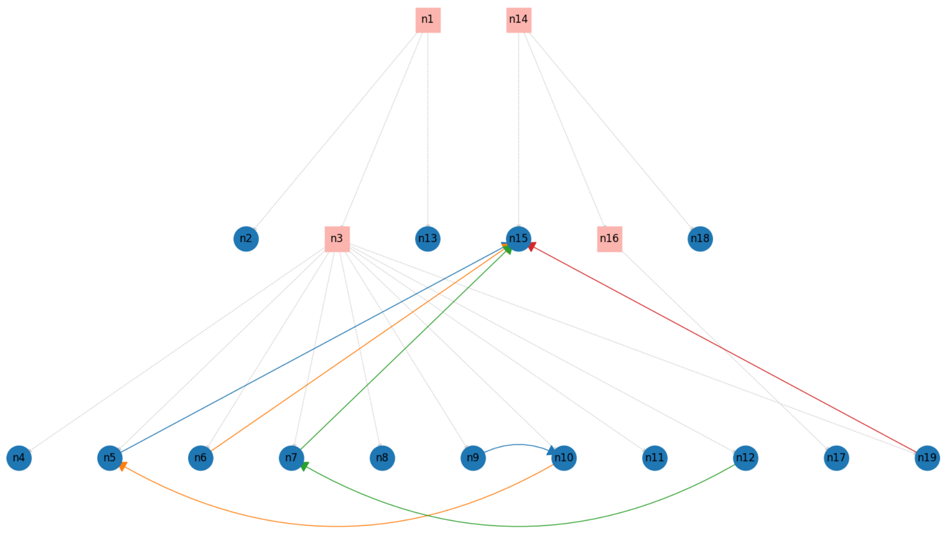
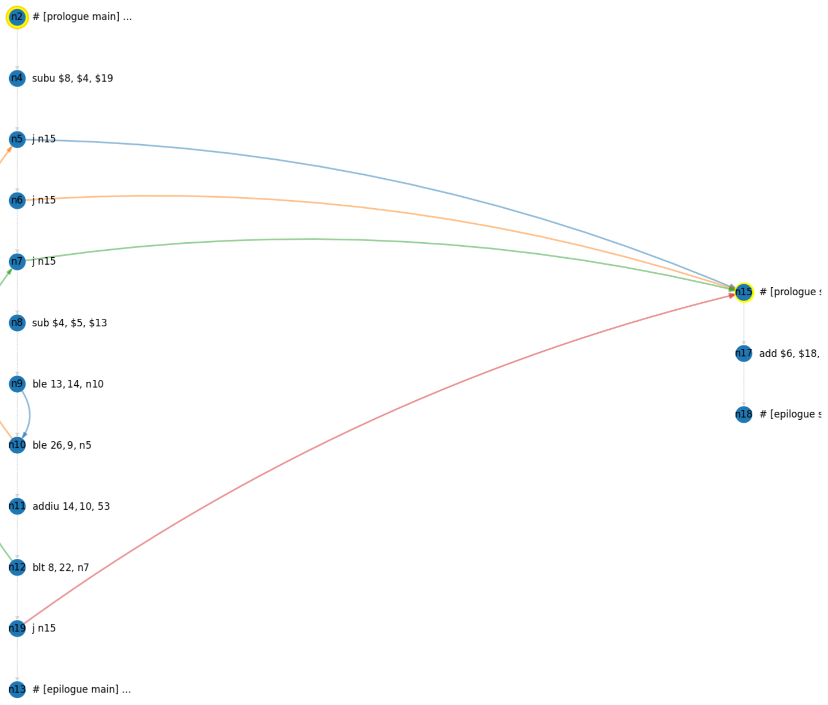


Figure 2.5: Individual's genome represented as a forest



```
n2  # [prologue main] ...
n4  subu $8, $4, $19
n5  j n15
n6  j n15
n7  j n15
n8  sub $4, $5, $13
n9  ble 13, 14, n10
n10 ble 26, 9, n5
n11 addiu 14, 10, 53
n12 blt 8, 22, n7
n19 j n15
n13 # [epilogue main] ...

n15 # [prologue s
n17 add $6, $18,
n18 # [epilogue s
```

Figure 2.6: Individual's genome represented as a Linear Genetic Program

Add Macro to Bunch

It selects a Frame node with at least one free branch, it randomly selects a macro from the pool of possible macros for that frame and add to the frame a new instance of that macro.

Remove Macro to Bunch

It randomly selects a macro from the ones belonging to a frame and removes it.

Parameter Crossover

It selects a parameter encoded as a list from both the parents and merge these into a new parameter.

Generic Node Crossover

It selects a node from both the parents and merges them in a new node.

2.1.5 Parents Selection

In 1.1 is introduced the concept of reproduction between individuals, but no information about how these individuals are selected is provided.

There are multiple way to operate this decision. The most renowned are:

Ranking Selection

In this selection individuals are ranked according to their fitness. Then, a decreasing along side the rank probability of being selected is assigned to each one of the population members. The function that map the probability to the rank is not standard, as it can be linear, exponential or else.

Fitness Proportional Selection

It is the simplest selection algorithm, in which the probability of an individual of being selected is the relationship between its fitness and the total fitness of the population.

Tournament Selection

Both of the previously discussed algorithms, require a complete knowledge of the entire population. This is not always possible or computational negligible, so it is necessary to find a different way. With this algorithm n individual are randomly selected and compared. The metric used for the comparison may vary. This way it is not required to know the entire population, but only the individuals drawn for the tournament.

In Byron Tournament Selection is implemented, using the fitness of the individual as metric.

2.1.6 Plugin

Byron is developed to be expandable according to the user's needs. For this purpose, Python's decorators [10] are used to make it possible to integrate different components, Operators for example, from the user side to the Byron core.

2.2 Self Adapting Evolutionary Algorithm

A self-adapting evolutionary algorithm is an algorithm able to modify internal parameters to adapt its adaptation mechanisms to different problems. There are several parameters inside an evolutionary algorithm that can (or that can better not) be tweaked.

2.2.1 Selection

The first parameter to consider is how the offspring (λ) are combined with the pre-existing population (μ). Two main groups of strategies exist:

- Comma strategies: (μ, λ)
- Plus strategies: $(\mu + \lambda)$

Comma

In these strategies, the previous population is discarded before the offspring are evaluated. This leads to different advantages, for example, discarding all parents could help in escaping a local optima [11]. Nonetheless, for the same principle, an optimum solution may be discarded (although not forgotten) during the process, preventing more exploration in that direction.

Plus

Here instead, the already present population and the offspring are merged and then the worst individuals are discarded. This way optimum solutions are preserved until better optima are found. On the other side, the prolonged presence of a local optima solution could cause a delay in the process of escaping that optima in order to find a better one.

Recombination and Elitism

If the best individuals are preserved no matter what their age or the population size, it is usually called elitism. In case recombination operators are used, it is used the formalism $(\mu/\rho\ [+/,]\ \lambda)$, where ρ represents the number of individuals used in the recombination operations.

Thesis Development

The algorithm developed in this thesis is by default a $(\mu/\rho + \lambda)$ without elitism, but by tweaking the maximum lifespan of an individual and the elitism parameter, the user could easily transition to a $(\mu/\rho, \lambda)$ or (μ, λ) selection.

Despite in some cases [12] varying μ and λ could produce better results, usually, it is not the case, so in the developed algorithm, those are treated as fixed parameters.

2.2.2 Temperature

This term is normally used in Selective Annealing [13], but it perfectly fits also in this context. Temperature is a parameter used to balance two different phases of the search process: Exploration and Exploitation. There is no rigorous definition of these terms, but an intuition can be given.

Exploration is usually referred to as the first phase of the search, in which individuals are scattered in the solution space, exploring undiscovered regions.

In the Exploitation phase, instead, the individuals are located near a known good solution, trying to slightly improve it.

Thesis Development

In the algorithm developed, the temperature is a trigger for the exploitation phase to start. It is used to determine, in the case of a fitness target, the proximity of the actual population to the target. If this proximity is confirmed, the temperature is used to tweak the strength of the genetic operators (more on that in the next paragraph).

2.2.3 Operator Strength

The strength of an operator is defined as the impact that a replication has with respect to the input individual. A higher strength push towards exploration, while a lower ones push towards exploitation.

Thesis Development

The impact of the mutation differs based on the type of the target:

Array Parameter

For the Array Parameter, namely the parameter coded as a list, the strength of the mutation determines how many elements of the list are subjected to mutation.

Acting on every element will result in the birth of an individual with a parameter with no direct correlation to the parent one, frustrating the idea of the algorithm. To avoid this the scale parameter was introduced. Its goal is to determine, given the strength of the operator, an upper limit of how many list elements will be changed.

Add / Remove Macro to Buch

These operators do not have an intrinsic concept of strength, because removal and addition are boolean concepts: they are done or not, and there is nothing in between. So, the strength was applied not on the action per se, but on the subject of the action. More specifically, the exploration phase corresponds to adding to the bunch a macro uniformly selected between the set of all possible macros available in the node; while the exploitation phase sees a reduction of the diversity, with a constantly decreasing quantity of macros available for the choice, reducing the number to only the most frequent ones. Conversely, the higher the lower the strength, the less frequent will be the removed macro.

2.3 Operator Selection

In an evolutionary algorithm, several genetic operators are present. How to select the most appropriate one, or how to not select the ones that perform poorly, is a non-trivial question. First of all, it is needed to define what "poorly" means and, more importantly, how to evaluate an operator.

2.3.1 Reward

The evaluation of operators in Byron heavily relies on the performance tracking seen in 2.1.4. The idea is to reward operators able to produce individuals better than their parents, but also operators able to simply produce a valid individual (although in the latter, the reward would be smaller).

On the other side, no penalties are provided for generating individuals worse than their parents or for failure in the creation of the offspring. The logic behind this is that, especially for very complex problems, generating valid individuals could not be an easy task, so even good and useful operators could fail it.

2.3.2 Regret

Having defined a reward r for the desired behaviour of the operators, it is possible to define $\mu(o_p)$ as the mean reward of o_p, where $o_p \in O$, with O being the set of available operators. Furthermore is possible to define $\mu^* = max(\, \mu(o_p) \, \forall \, o_p \in O)$ as the best mean reward.

Supposing to randomly select T times an operator from O and running it, it is possible to compute

$$R(T) \; = \; \mu^* \cdot T - \sum_{t=1}^{T} o_{p_t}$$

This quantity is known as *Regret* at time T. This compares the cumulative reward obtained by the algorithm against the reward obtained by using an optimal strategy.

2.3.3 Random

Now that it is defined a way to evaluate an operator, there is the need to decide how to select the most appropriate one. By default, Byron implements a random selection, where every operator is drawn with IID (Independent and Identically Distributed) probability. This method, however, does not consider any information about the operator's performance. The regret obtained by this selection method has been estimated to be

$$R(T) \leq \; \mathcal{O}(KT) \tag{2.1}$$

where K is the number of operators.

Since better results could be achieved using the information logged by Byron itself, in this thesis a new selection algorithm was developed.

2.3.4 Successive Elimination Algorithm

Indeed, even if the "goodness" of an operator is defined, what is not defined is a way to compare two different operators. To solve this problem a few more steps are required. Firstly, the *Confidence Radius* is defined as

$$r_t(o_p) = \sqrt{\frac{2 \log T}{n_t(o_p)}}$$

where $n_t(o_p)$, in the context of this thesis, is the number of times an operator has been selected.

Then, it is possible to define respectively *Upper* and *Lower Confidence Bound* at times t:

$$UCB_t(o_p) = \ \mu_t(o_p) + r_t(o_p)$$

$$LCB_t(o_p) = \ \mu_t(o_p) - r_t(o_p)$$

while $[LCB_t(o_p); UCB_t(o_p)]$ is called *Confidence Interval.* It represents the interval where it is assumed that the mean reward of an operator will fall. Now, having defined what rewards are expected by the operators, it is possible to evaluate if an operator performs better than another.

Having all the necessary information leads then to the definition of the algorithm:

Algorithm 1 Successive Elimination

```
All operators are active
**for** each round $t$: **do**
    try all active operator
    deactivate all operator $o'_p$ so that $\exists o_p \ : \ UCB_t(o'_p) < LCB_t(o_p)$
    **if** $t$ / $\frac{T}{4}$ has ratio == 0 **then**
        retry all operators
    **end if**
**end for**
```

with regret[2]:

$$R(T) \leq \ \mathcal{O}(\sqrt{KT \log T}) \tag{2.2}$$

Comparing 2.1 versus 2.2, it is seen in Fig 2.8 that the latter presents a much smaller regret.

This algorithm allows to reduce how many times a not appropriate operator is selected, hence reducing the waste of computational time used to create a non-valid individual that will be discarded immediately after its creation.

[2]Computation omitted in 2.1 and in 2.2 can be found, together with a more exhaustive explanation of Algorithm 1, in [14] and [15].

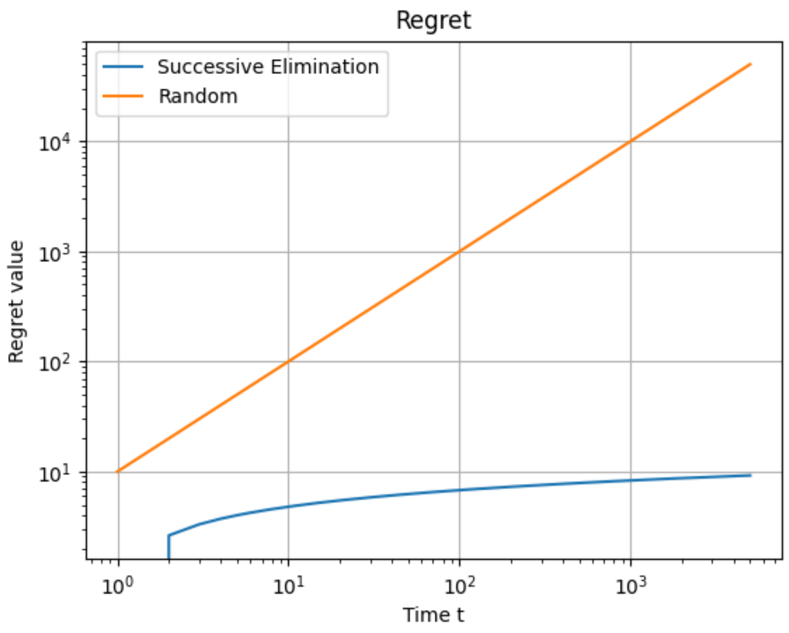


Figure 2.7: Regret of random versus Successive Elimination Algorithm at increasing time T

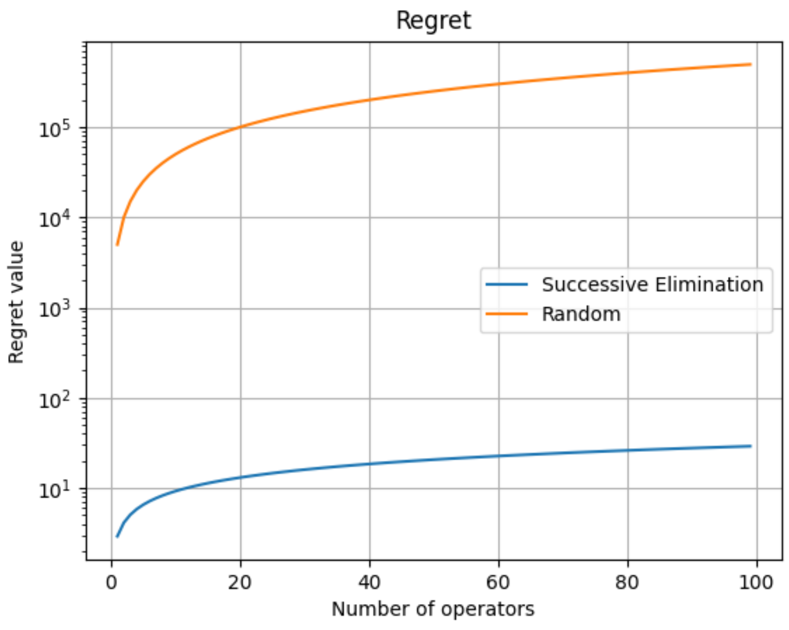


Figure 2.8: Regret of random versus Successive Elimination Algorithm at increasing number of operators

Chapter 3

Implementation

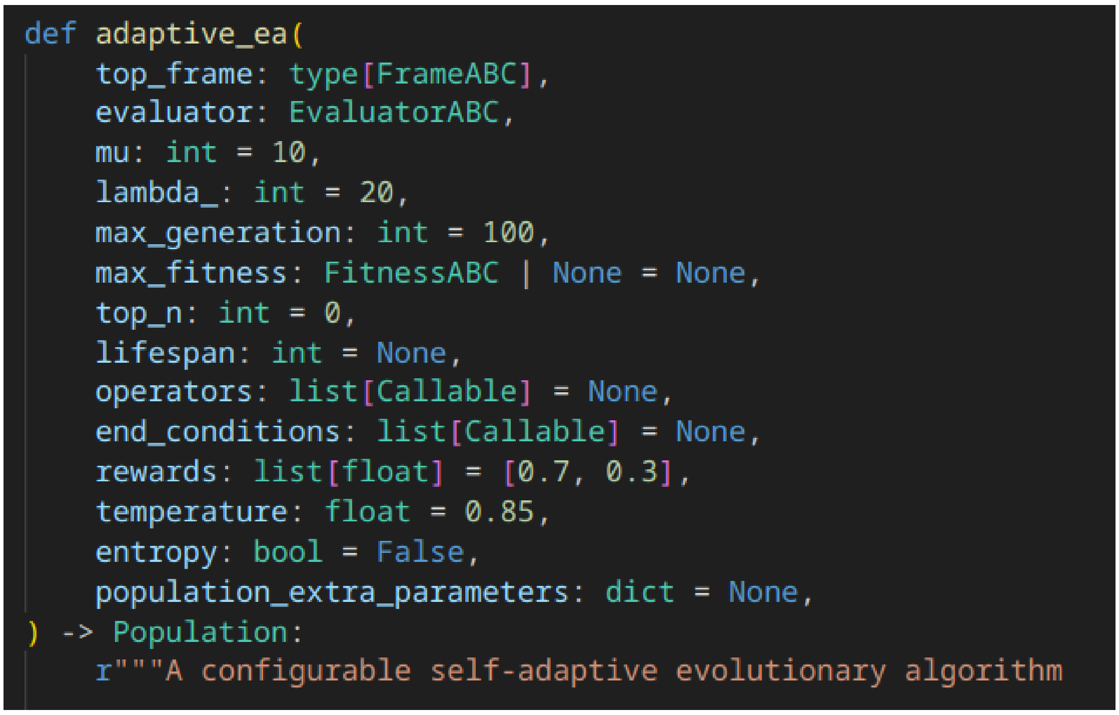
Will clean air smell any sweeter?
Will sunny days be any brighter?
Will starry nights hold any more wonder?,
Don Rosa

In chapter 2 were exposed the theory and the main ideas behind self adapting evolutionary algorithm and operators' selection, together with some conceptual description of the implementations. In this chapter, instead, will be provided the actual implementation of the above-mentioned concepts.

Differently from the conceptual description, the implementation is not a single algorithm managing everything. Instead, the implementation has led to a set of distinct functions working together synergistically.

3.1 Adaptive Ea

This function was written with the idea of being a highly customisable user-accessible evolutionary algorithm. As the user's entrance point it demands most of the adaptations and optimizations to the Estimator class (3.2), which is not designed to be directly accessed by users. Nonetheless this function directly manages the selection methods (2.2.1). Given the aim to be as customisable as possible, it exposes several parameters:



```
def adaptive_ea(
    top_frame: type[FrameABC],
    evaluator: EvaluatorABC,
    mu: int = 10,
    lambda_: int = 20,
    max_generation: int = 100,
    max_fitness: FitnessABC | None = None,
    top_n: int = 0,
    lifespan: int = None,
    operators: list[Callable] = None,
    end_conditions: list[Callable] = None,
    rewards: list[float] = [0.7, 0.3],
    temperature: float = 0.85,
    entropy: bool = False,
    population_extra_parameters: dict = None,
) -> Population:
    r"""A configurable self-adaptive evolutionary algorithm
```

Figure 3.1: Self Adapting Algorithm's internals

3.1.1 Parameters

- top_frame: this is individual's top frame. It contains all the others frames, macros and parameters of the genome. It is used to create the initial individuals.

- evaluator: it's the Byron wrapper for the user defined fitness function. The use of the evaluator allows an easy way not only to parallelise the evaluation of multiple individuals, but also a way to implement external fitness functions, such as script written in other languages, like Bash, Go or C.

- mu: the size of the population. At the end of every generation the population will be reduced of a number of individuals to equals this value.

- lambda_: the size of the generated offspring. At each generation a number of individuals equals to this value will be generated and added to the population.

- max_generation: maximum number of generations after which the algorithm stops, whether the fitness target was reached or not.

- max_fitness: the fitness target that needs to be reached. If no target is provided, the algorithm will continue to run until the max_generation limit is reached.

- top_n: this value, if set, enable elitism in the algorithm. It defines the maximum number of elite individuals, also known as champions population, allowed to exist. These individuals do not age and are guaranteed to survive as long as they belong to the elite. If top_n == mu, lifespan parameter is useless.

- lifespan: it represent the maximum age (i.e. the maximum number of generations) an individual not belonging to the elite could remains in the population before it dies. If top_n == mu every individual in the population will be part of the elite, hence it will not age, making this parameter useless.

- operators: a list of user-defined genetic operators that will be used instead of the Byron standard ones.

- end_conditions: list of functions that will be checked at each generation to stop the evolution. Functions in this list have to return a boolean in order to be checked by the algorithm. By default max_fitness (if a target is provided) and max_generation are the only checked conditions.

- reward: list of rewards for the genetic operators. If no specific values are provided bu the user, the ones selected through an hyper-parameter tuning are used.

- temperature: the temperature value to balance exploration and exploitation.

- entropy: A flag to use population entropy parameter to promote diversity in population. Currently not used.

- population_extra_parameters: additional parameters that are added to the population.

3.1.2 Algorithm

Initialisation

The stopping conditions and the population are created. The estimator is initialised with the user defined parameters and the genetic operators specific for initialisation are selected.

After the selection, the first generation of individuals is generated. Given the absence of previous data and the uniqueness of the initialisation, the operator selection is purely random. After the first batch of individuals is born, they are evaluated and the population is sorted accordingly.

Evolution

Now, until a stopping condition is reached, the evolution proceeds. Firstly the mutation strength is retrieved from the estimator. This strength will be used for every individual of the generation.

Then, the estimator is called to select a genetic operator. From the operator the needed number of parents is obtained, each of them is then selected through a tournament. After, the operator is called with the individuals selected as arguments and, if supported, the strength parameter.

When the number of offspring reaches the value of λ, the individuals creation is stopped. At this point if a lifespan was given, the individuals in the population age and those that exceed the lifespan threshold are removed. Then the offspring are added to the population, which is evaluated and sorted again. After that all the individuals with the lowest fitness that exceed the population dimension (μ), are removed and the estimator is updated.

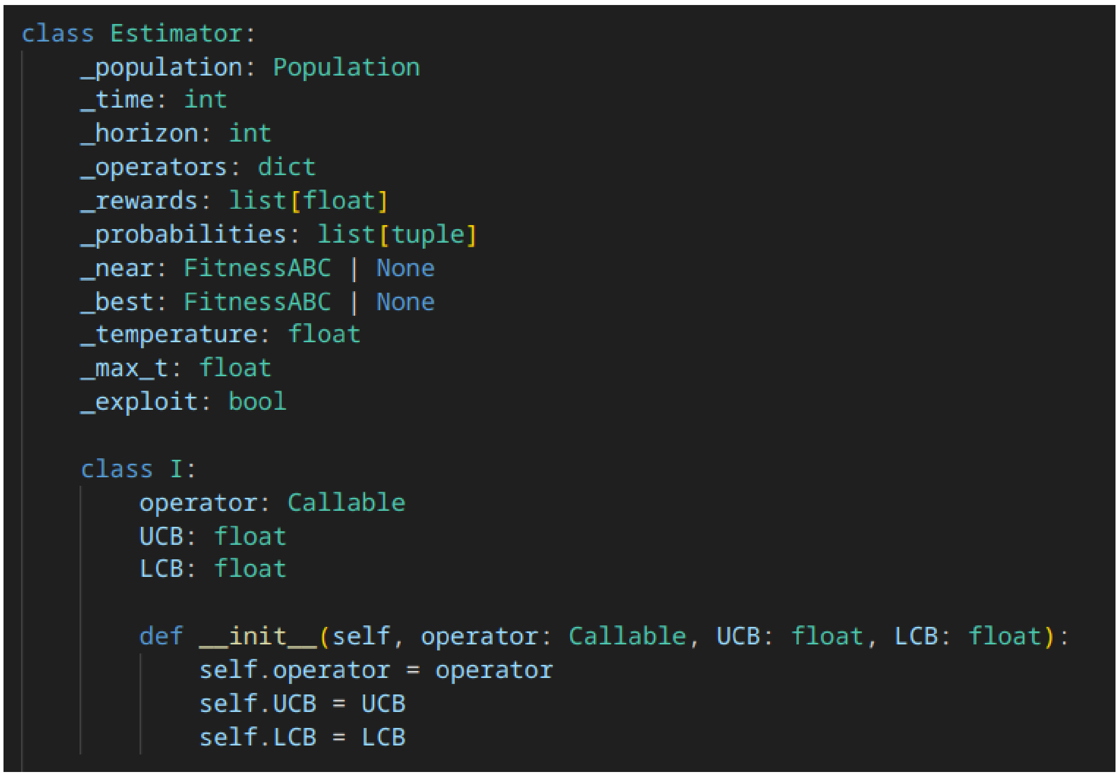
Termination

When a stopping condition is matched, the evolution stops. The algorithm returns the last evaluated population and print some statistics about the used operators.

3.2 Estimator

This class is the core of the adaptation process. It manages both the sigma adaptation (2.2.2) and the Successive Elimination algorithm (2.3.4).

3.2.1 Parameters



```
class Estimator:
    _population: Population
    _time: int
    _horizon: int
    _operators: dict
    _rewards: list[float]
    _probabilities: list[tuple]
    _near: FitnessABC | None
    _best: FitnessABC | None
    _temperature: float
    _max_t: float
    _exploit: bool

    class I:
        operator: Callable
        UCB: float
        LCB: float

        def __init__(self, operator: Callable, UCB: float, LCB: float):
            self.operator = operator
            self.UCB = UCB
            self.LCB = LCB
```

Figure 3.2: Estimator's internals

Internally it stores several information:

- _population: a reference to the population of the evolutionary algorithm
- _time: an internal parameter to keep track of the execution's generations
- _horizon: to compute the confidence radius, the maximum number of generations
- _operators: a dictionary with all the possible operators available

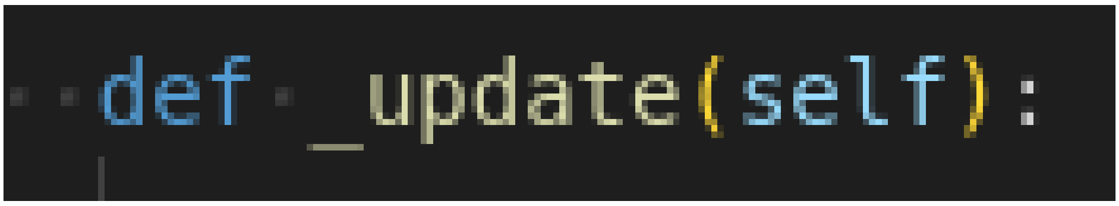
- _rewards: a list of the defined rewards for the operators
- _probabilities: a list of tuple *(operator, probability of being select)* for the selection of the operator in a given time. [1]
- _temperature: the parameter used to balance exploration and exploitation
- _max_t: the maximum temperature set by the user
- _near: the fitness threshold below which the exploitation phase begin
- _best: the best achieved fitness during the execution

It also contains the class *I*, which provides a compact way of maintaining information about operators' performances.

To interact with the Estimator there are different functions available. The idea behind these functions is to have the most minimal signature [16] possible, so that after the creation of the Estimator, no other parameters are required for the system to work. This allows this component to be interoperable to every algorithm developed for Byron and not only to 3.1

3.2.2 Update

This function manages the update of internal components and the way time is computed in the class. It is not called by an external agent, but instead is called by the Sigma (2.2.2) function, in order to keep track of the passing generations.



```
def _update(self):
```

Figure 3.3: Update function's signature

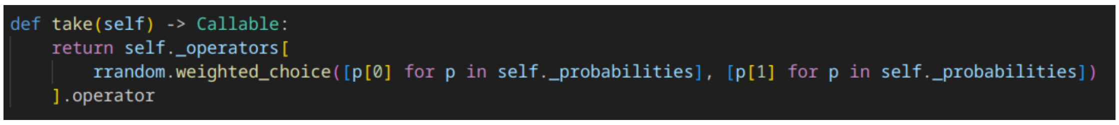
[1]currently the probability of being selected is equal for all the valid operators. The probability information is preserved for future improvements

Every time this function is executed, it computes the confidence interval for every operator. It then apply the successive elimination algorithm, ruling out from the valid operator set the ones that do not meet the requirements.

Computing every time the confidence interval, finding the max value of the lower bound, has a fundamental advantage against keeping the max from the previous iteration. That is, if at a given time the best operators start to fail for some reasons, the overall LCB margin will decrease. Thanks to this, other operators previously discarded could be considered valid again. The ratio behind this choice is that if too many failure start to appear suddenly, the active operators may be not suited anymore to the current structure of the genome, while operators that did not performed well in the past could now have a positive impact. In order to minimize the creation of invalid individuals, the first operators to be re-activated will be the ones that have the most probabilities to produce a valid individual: i.e. from the most recent deactivate ones to the less ones.

Being the function that manages time, it is also the one that, every quarter of execution, re-enables the discarded operators for a test run, in order to check if they could now behave better than when they were discarded.

3.2.3 Take

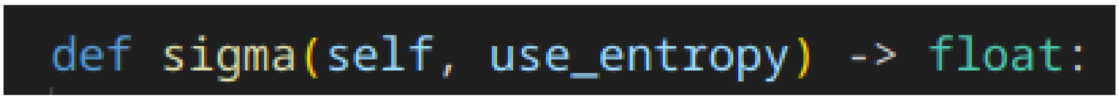


```
def take(self) -> Callable:
    return self._operators[
        rrandom.weighted_choice([p[0] for p in self._probabilities], [p[1] for p in self._probabilities])
    ].operator
```

Figure 3.4: Take function

This is a quite simply but at the same time fundamental function. In fact through this function, the algorithm could retrieve the operator to create the next individual of the population. The set from where the operator is drawn is managed by 3.2.2 and here it is simply performed a choice based on the probability of each operator. The use of this structure allows a high re-usability, because no assumption is made about the probabilities distribution or how it is computed.

3.2.4 Sigma



```
def sigma(self, use_entropy) -> float:
```

Figure 3.5: Sigma function's signature

The function sigma is the one designated to balance exploration and exploitation. To achieve so, it measure the distance between the actual fitness found in the population and the *_near* value computed at the initialization of this class. Optionally, this function could also performs a check on the Shannon's entropy [17] of the population. This allows a fine check about the diversity in the population and the ability to tweak the temperature according to the actual population. [2] The default state of the function is exploration. When the above mentioned conditions are satisfied, the function enter in the exploitation phase. In this phase it allows the Update function to tweak the temperature, slowly reducing it at every iteration, until it reach a lower limit. When this limit is reached, the temperature stops to decrease and remains partially still, with minor variations to improve the solution' search. This function is used to obtain the strength of the mutation for a genetic operator, and so it returns this value based on the actual temperature if the function is in the exploitation phase, and 1 instead.

3.3 Hyper-parameters Tuning

Due to the high customizability, a large number of parameters are needed in order to manage all the available options. Given that the user could not know all the best values for the parameters, it is needed that the default ones provided with the functions are a good trade-off between performance and versatility.

[2]Actually the entropy computation for the population is still under development, so an extra parameter in the function' signature was added to temporarily disable this check while preserving the functionality for development purpose

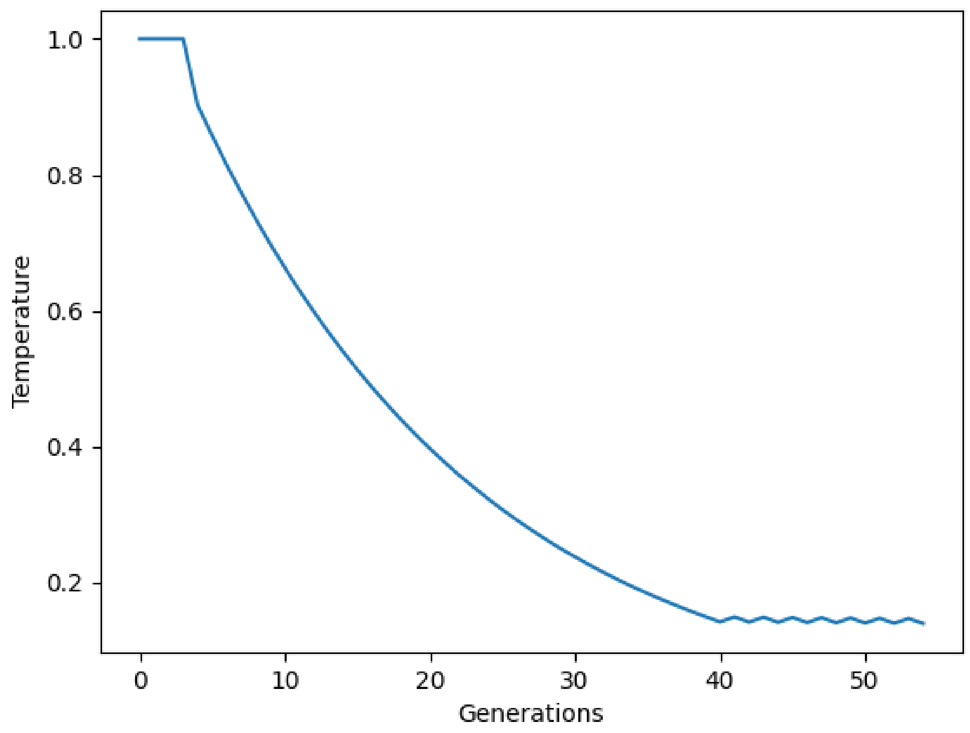


Figure 3.6: Temperature decreasing in the exploitation phase

3.3.1 Population

The first and maybe most difficult parameter (or better, in this case, parameters) to find is the population and offspring size.

Methodology

It is not trivial to define how to measure the difference in performance regarding the population size. In fact many possible metrics heavily rely on the population itself. Number of individuals or generations are not reliable metrics in this case, so it is decided to focus on the time needed for the algorithm to find the solution.

Experiments where executed on a Onemax [3] benchmark test several times and

[3]It is the most simple and popular example of evolutionary algorithm usage: the genome of

average accordingly.

As Shown in Figure 3.7, the results could be very different between themselves w.r.t. the variation of the values. Also it has to be noted that several studies as, for example, [18], showed that a large population can be unhelpful and that, in general, population size could be extremely problem dependant.

Result

With these elements in mind, it was considered a good trade-off selecting a small dimension for the population and the offspring, so that to ease the computational load. It was also decided to keep $\lambda > \mu$ in order to keep available by default Comma strategies (2.2.1).

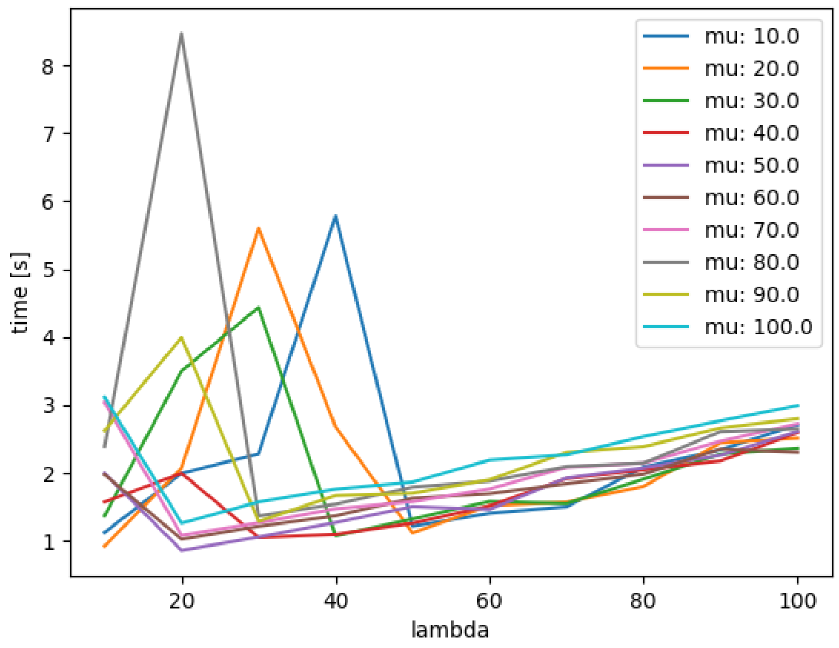


Figure 3.7: Performance according to Mu/Lambda variation

every individual it is a string of '0's and '1's randomly distributed. The fitness function is the number of '1's present in the genome

3.3.2 Rewards

The operator rewards are fundamental parameters for the operator selection phase.

Methodology

Having a fixed population and offspring dimension, the parameter choose as a metric is the number of generation needed to reach the desired fitness. This value was preferred because directly measuring single operator' stats would results in a partial view of the overall situation, while the number of generation is a cumulative metrics of all the previous values.

The rewards considered are

- Reward for an individual better than parents (success)
- Reward for a valid individual

The most important is the "Success" one, while "valid" can vary much more. With this in mind firstly the same benchmark as in 3.3.1 was run. Then, the variance between the number of generations was analyzed to find the most stable reward for success (Fig 3.9). Found this value, using Figure 3.8, a good value for the "valid" reward was selected.

Result

Again as for the population, it was not selected the best value overall but a good enough one, in order to keep the versatility of the parameters.

3.3.3 Temperature

The temperature manages the balance between exploration and exploitation

Methodology

Since this parameter does not only influences who the exploitation phase is conducted, but also when this phase begin, it is not that useful looking at metrics like the operators statistics. Since temperature will change how the solution space is explored, reducing the exploration and restring its scope, the lesser the value, the

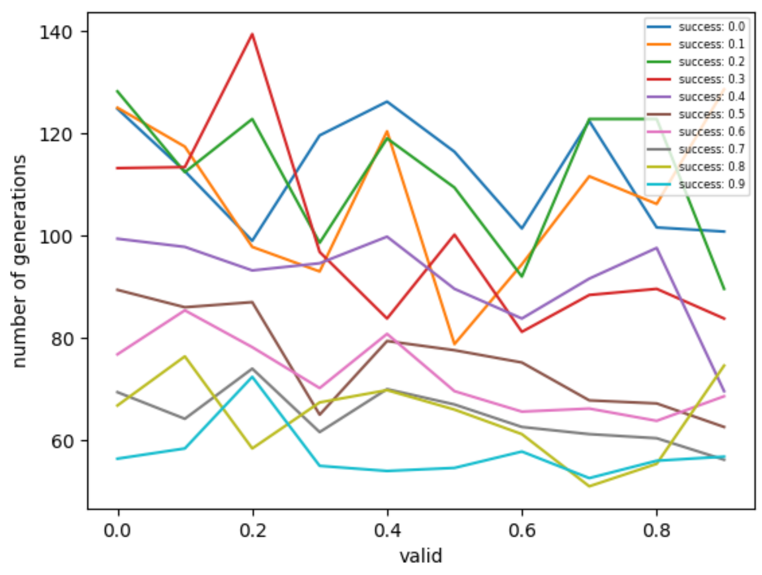


Figure 3.8: Performance according to Rewards variation

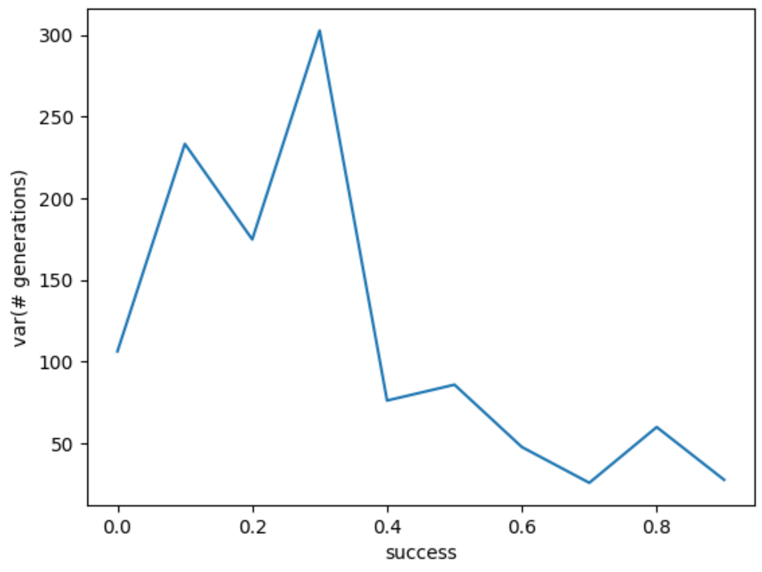


Figure 3.9: Variance of the number of generations for each rewards

smaller will be the pace of the exploration. So an obvious metric to be analyzed could be the time needed for the algorithm to complete. Nonetheless, timing the algorithm requires external resources, while there is another metric that perfectly serves for the goal: the number of generations. Indeed, if the exploitation start sooner, more generations will be needed to explore the same area of space that can be explored in the exploration phase.

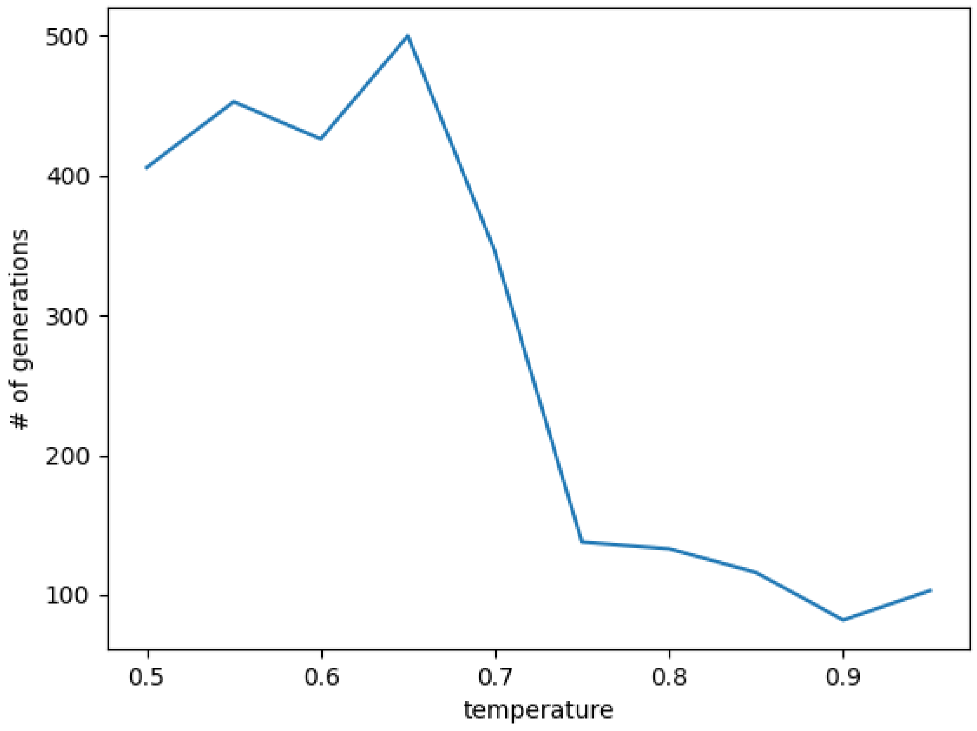


Figure 3.10: Number of generation per different temperature

Result

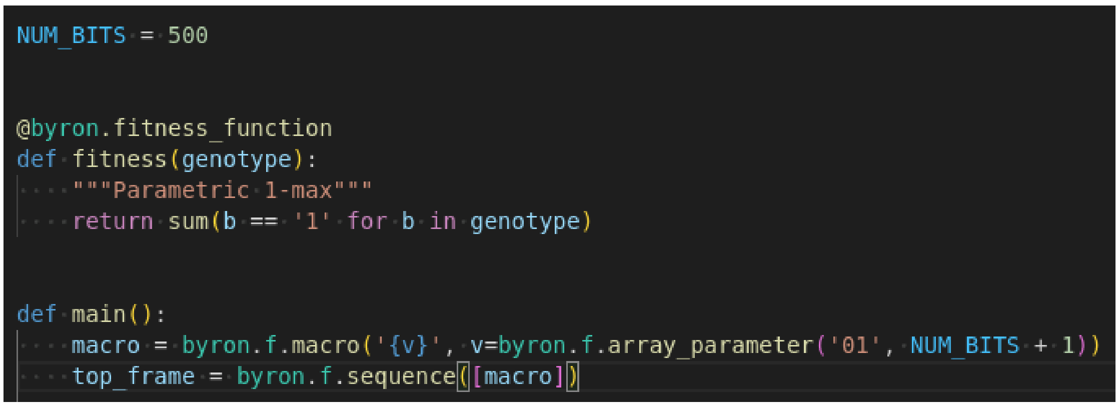
As shown in Figure 3.10, there is a huge gap of performances between 0.65 and 0.75, where a later starting of the exploitation phase lead to a more then halving number of generations. Since some problems could benefit of a earlier start of this last phase, the chosen parameter was again not the best found overall, but a good enough one.

3.4 Comparison

The already present vanilla evolutionary algorithm in Byron is little more than a demonstrator of Byron capabilities or, in other words, a basic evolutionary algorithm without self adaptation. Nonetheless could be interesting to show the difference in performance between a standard non self-adapting algorithm and the one developed in this thesis.

The test suite consists in:

3.4.1 Onemax



```python
NUM_BITS = 500


@byron.fitness_function
def fitness(genotype):
    """Parametric 1-max"""
    return sum(b == '1' for b in genotype)


def main():
    macro = byron.f.macro('{v}', v=byron.f.array_parameter('01', NUM_BITS + 1))
    top_frame = byron.f.sequence([macro])
```

Figure 3.11: Onemax problem definition

It is the most simple and popular example of evolutionary algorithm usage: the genome of every individual it is a string of ’0’s and ’1’s randomly distributed. The fitness function is the number of ’1’s that are present in the genome. In this case the string length is equal to 500. Only one type of macro and frame are used, this is both for simplicity but also to create an ad hoc environment where not every operator is useful.

It is clearly visible from Table 3.1 that the self-adapting algorithm outperform

Algorithm perfomance		
Algorithm	Individuals	Generations
vanilla_ea	63659	2834
adaptive_ea	31423	833

Table 3.1: Algorithms performances on Onemax problem

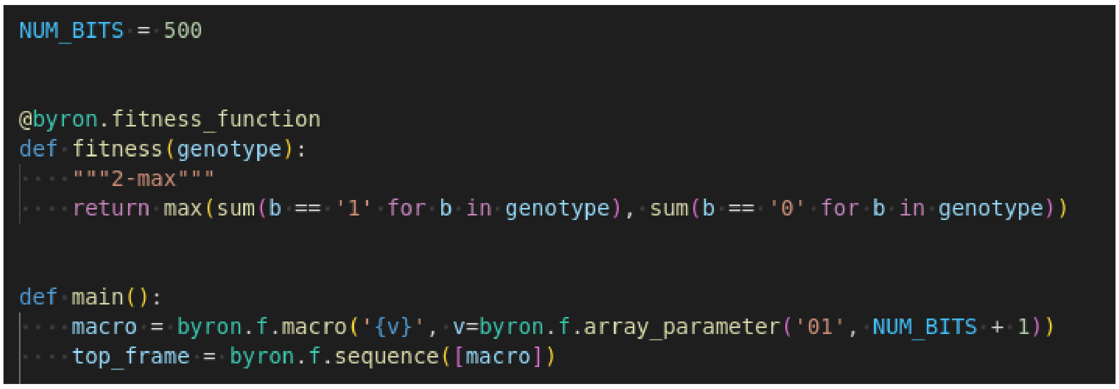
Algorithm perfomance				
Algorithm	operator	calls	abort	success / valid
vanilla_ea	random_individual (init)	10	10	/
	add_macro_to_bunch (mut)	6988	6988	/
	remove_macro_from_bunch (mut)	7084	7084	/
	single_element_array_parameter_mutation (mut)	7210	0	364 / 3244
	single_parameter_mutation (mut)	7003	0	7 / 6996
	array_parameter_uniform_crossover_choosy (xover)	7080	0	1297 / 1243
	leaf_crossover_unfussy (xover)	7203	3727	480 / 486
	node_crossover_choosy (xover)	7015	7015	/
	node_crossover_unfussy (xover)	7097	7097	/
adaptive_ea	random_individual (init)	10	10	/
	add_macro_to_bunch (mut)	179	179	/
	remove_macro_from_bunch (mut)	164	164	/
	single_element_array_parameter_mutation (mut)	5196	0	76 / 4670
	single_parameter_mutation (mut)	4350	0	4 / 4346
	array_parameter_uniform_crossover_choosy (xover)	5637	0	894 / 758
	leaf_crossover_unfussy (xover)	789	431	81 / 85
	node_crossover_choosy (xover)	179	179	/
	node_crossover_unfussy (xover)	166	166	/

Table 3.2: Performance of each operator on Onemax, with specified if it is a "Mutation", an "Initializer" or a "Crossover"

the vanilla one as per number of generations and also per number of individuals. Moreover, in Table 3.2 is shown the operator selection algorithm cutting off the operators that are not suited for this problem

3.4.2 Twomax

This problem is a slight modification of the previous. Here, instead of measuring the fitness on the number of '1's, the fitness is computed based on the maximum number of ’0’ or ’1’ in the genome. in this way, an algorithm that promote diversity is favored with respect to one that does not, since more individuals with a different genome could have similar results.



```python
NUM_BITS = 500


@byron.fitness_function
def fitness(genotype):
    """2-max"""
    return max(sum(b == '1' for b in genotype), sum(b == '0' for b in genotype))


def main():
    macro = byron.f.macro('{v}', v=byron.f.array_parameter('01', NUM_BITS + 1))
    top_frame = byron.f.sequence([macro])
```

Figure 3.12: Twomax problem definition

Algorithm perfomance		
Algorithm	Individuals	Generations
vanilla_ea	111451	5000
adaptive_ea	26290	707

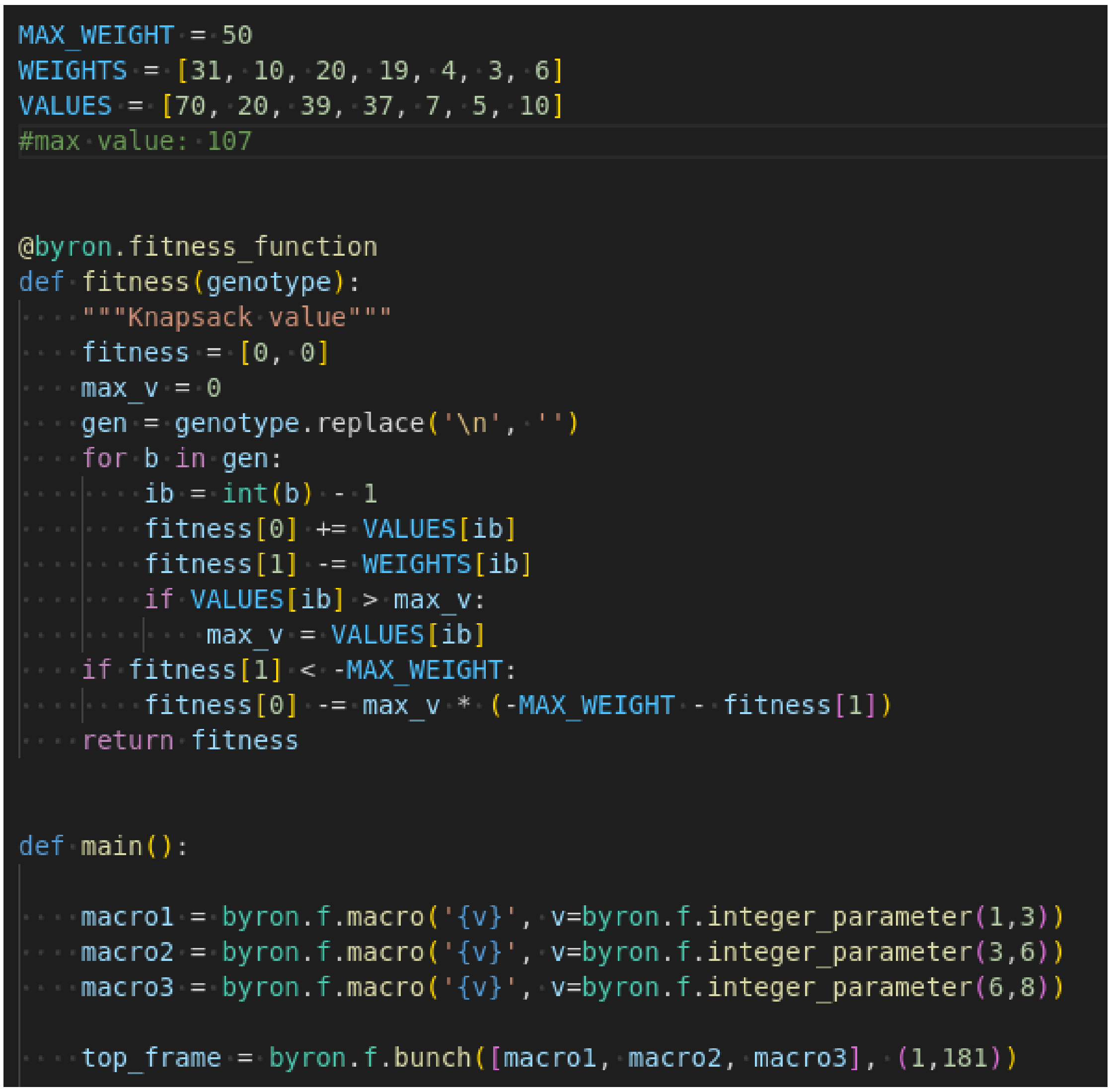
Table 3.3: Algorithms performances on Twomax problem

The difference in performance here is more than evident. The vanilla algorithm was not able to find the desired solution and run out of computational time reaching a max fitness of 495. The table with the operators performances is omitted because it simply shows the same behaviour shown in Table 3.2

3.4.3 Knapsack

Knapsack problem is a well known [19] problem in combinatorial optimization where, given a set of item with a defined weight and value, the goal is to obtain the maximum value without exceeding a maximum weight. This example is worth showing for two reasons.

The first one is the non trivial fitness function. Far from affirming that the function itself is complex, the non triviality can be found in the process of deciding how to compute the fitness, while enforcing the weight’s limit. Above all, this is an



```python
MAX_WEIGHT = 50
WEIGHTS = [31, 10, 20, 19, 4, 3, 6]
VALUES = [70, 20, 39, 37, 7, 5, 10]
#max value: 107


@byron.fitness_function
def fitness(genotype):
    """Knapsack value"""
    fitness = [0, 0]
    max_v = 0
    gen = genotype.replace('\n', '')
    for b in gen:
        ib = int(b) - 1
        fitness[0] += VALUES[ib]
        fitness[1] -= WEIGHTS[ib]
        if VALUES[ib] > max_v:
            max_v = VALUES[ib]
    if fitness[1] < -MAX_WEIGHT:
        fitness[0] -= max_v * (-MAX_WEIGHT - fitness[1])
    return fitness


def main():

    macro1 = byron.f.macro('{v}', v=byron.f.integer_parameter(1,3))
    macro2 = byron.f.macro('{v}', v=byron.f.integer_parameter(3,6))
    macro3 = byron.f.macro('{v}', v=byron.f.integer_parameter(6,8))

    top_frame = byron.f.bunch([macro1, macro2, macro3], (1,181))
```

Figure 3.13: Knapsack problem definition

example of how the user intervention and reasoning is needed in the definition of the problem.

The second and most obvious reason is to show how, changing the genome composition, it changes also the operators that can be used, as can be clearly seen comparing Table 3.2 and Table 3.5.

Algorithm perfomance		
Algorithm	Max Value	Generations
vanilla_ea	94	500
adaptive_ea	107	387

Table 3.4: Algorithms performances on Knapsack problem

Algorithm perfomance				
Algorithm	operator	calls	abort	success / valid
vanilla_ea	random_individual (init)	10	10	/
	add_macro_to_bunch (mut)	1222	0	5 / 1217
	remove_macro_from_bunch (mut)	1269	0	45 / 1224
	single_element_array_parameter_mutation (mut)	1252	1252	/
	single_parameter_mutation (mut)	1249	0	23 / 1226
	array_parameter_uniform_crossover_choosy (xover)	1286	1286	/
	leaf_crossover_unfussy (xover)	1241	972	13 / 12
	node_crossover_choosy (xover)	1230	1230	/
	node_crossover_unfussy (xover)	1251	1251	/
adaptive_ea	random_individual (init)	10	10	/
	add_macro_to_bunch (mut)	2791	0	421 / 2370
	remove_macro_from_bunch (mut)	2845	0	50 / 2795
	single_element_array_parameter_mutation (mut)	235	235	/
	single_parameter_mutation (mut)	2865	0	346 / 2519
	array_parameter_uniform_crossover_choosy (xover)	237	237	/
	leaf_crossover_unfussy (xover)	565	429	45 / 70
	node_crossover_choosy (xover)	233	233	/
	node_crossover_unfussy (xover)	229	229	/

Table 3.5: Performance of each operator on Knapsack, with specified if it is a "Mutation", an "Initializer" or a "Crossover"

3.5 Conclusion

In this thesis is shown the process and the methodology behind the development of a self-adapting evolutionary algorithm and how it presents several advantages in performance against non self-adapting ones.
Despite a more complex structure and the need to check more parameters during the execution, this algorithm has proven to be able to outperform the non self-adapting one both in the solutions found and in the computational load produced.

It is also shown the modularity of the Byron Evolutionary Tool and its versatility in tackling different types of problems.

3.5.1 Future Works

Connecting to the work conducted in this thesis, additional steps can be taken to increase the capabilities of this tool.
Since the core parts of the tool are already completed and working, the problem of tomorrow will be expand its function integrating new algorithms, operators and ways to promote diversity, while perfecting what is already present.

Bibliography

[1] Charles Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. London: Murray, 1859 (cit. on p. 1).

[2] Alfred Russel Wallace Charles Darwin. «On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life». In: *Journal of the Proceedings of the Linnean Society of London* (1858) (cit. on p. 1).

[3] August Weismann and Margaret R Thomson. *The evolution theory*. Vol. 2. E. Arnold, 1904 (cit. on p. 1).

[4] Franz Weiling. «Historical study: Johann Gregor Mendel 1822–1884». In: *American journal of medical genetics* 40.1 (1991), pp. 1–25 (cit. on p. 1).

[5] George EP Box. «Evolutionary operation: A method for increasing industrial productivity». In: *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 6.2 (1957), pp. 81–101 (cit. on p. 2).

[6] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998, p. 8 (cit. on p. 3).

[7] Ernesto Sanchez, Giovanni Squillero, and Alberto Tonda. *Industrial applications of evolutionary algorithms*. Springer, 2012 (cit. on p. 4).

[8] Kenneth Alan De Jong. *Analysis of the behavior of a class of genetic adaptive systems*. Tech. rep. 1975, pp. 48–52 (cit. on p. 5).

[9] Cad Polito. *Byron Macro Description*. 2024. URL: `https://github.com/cad-polito-it/byron/blob/bd92cc2ee07459d2cd3e53457ca2598af4db0169/byron/framework/macro.py#L64` (cit. on p. 10).

[10] Python Software Foundation. *Decorator*. 2024. URL: `https://docs.python.org/3/glossary.html#term-decorator` (cit. on p. 15).

[11] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Springer, 2015, p. 89 (cit. on p. 16).

[12] Radka Poláková, Josef Tvrdík, and Petr Bujok. «Differential evolution with adaptive mechanism of population size according to current population diversity». In: *Swarm and Evolutionary Computation* 50 (2019) (cit. on p. 16).

[13] Emile Aarts and Jan Korst. *Simulated annealing and Boltzmann machines: a stochastic approach to combinatorial optimization and neural computing*. John Wiley & Sons, Inc., 1989 (cit. on p. 16).

[14] Aleksandrs Slivkins et al. «Introduction to multi-armed bandits». In: *Foundations and Trends® in Machine Learning* 12.1-2 (2019), pp. 15–16, 19–25 (cit. on p. 20).

[15] Eyal Even-Dar, Shie Mannor, and Yishay Mansour. «PAC bounds for multi-armed bandit and Markov decision processes». In: *Computational Learning Theory: 15th Annual Conference on Computational Learning Theory, COLT 2002 Sydney, Australia, July 8–10, 2002 Proceedings 15*. Springer. 2002, pp. 255–270 (cit. on p. 20).

[16] Jiwon Seo Brett Cannon et al. *Signature Object*. 2023. URL: `https://peps.python.org/pep-0362/#signature-object` (cit. on p. 27).

[17] Claude Elwood Shannon. «A mathematical theory of communication». In: *The Bell system technical journal* 27.3 (1948), pp. 379–423 (cit. on p. 29).

[18] Tianshi Chen, Ke Tang, Guoliang Chen, and Xin Yao. «A large population size can be unhelpful in evolutionary algorithms». In: *Theoretical Computer Science* 436 (2012), pp. 54–70 (cit. on p. 31).

[19] George B Mathews. «On the partition of numbers». In: *Proceedings of the London Mathematical Society* 1.1 (1896), pp. 486–490 (cit. on p. 37).

Acknowledgements

And to you,
if you have stuck with Harry
until the very end,
J. Rowling

Reader, a heartfelt thanks if you have read until here. Forgive me, but the rest of these lines will not be written in English.

Qualsiasi lista di ringraziamenti includerebbe forse la metà delle persone da ringraziare; e mostrerei, per più della metà di loro, meno della metà dell'affetto che meritano.
Per cui, chiunque tu sia, se in questi anni ci siamo incontrati, conosciuti, apprezzati e siamo infine diventati amici: grazie. Grazie perchè se sono riuscito ad arrivare alla fine di questo percorso, è anche merito tuo.

Grazie alla mia famiglia, che mi ha sostenuto, incorraggiato e, non secondariamente, cresciuto fino ad oggi. Sarebbe inutile nonchè impossibile entrare nel dettaglio di quanto hanno fatto, e quindi non lo farò; mi limiterò ad un abbraccio letterario, perchè ogni tanto un punto vale più di mille parole.

E grazie a te, Valentina, per Tutto.