Master Thesis

Design and Development of a Python Package implementing a General-Purpose Evolutionary Algorithm

Candidate:
Luca Barillari

Supervised by
Giovanni Squillero
Alberto Tonda

Final Project Report for the
Master in Computer Engineering - Data Science

DAUIN
Politecnico di Torino
Italy, Turin

March 2020

Design and Development of a Python Package implementing a General-Purpose Evolutionary Algorithm

Luca Barillari

Supervised by:

Supervisor 1
Giovanni Squillero - DAUIN
Supervisor 2
Alberto Tonda - INRA

Abstract

The thesis focuses on the design and on the development of MicroGPv4, a Python package implementing a versatile, general-purpose evolutionary algorithm. It explains how it was conceived, why certain decisions were taken, and how to make the most out of it. In the first part, the document explains the basis of an Evolutionary Algorithm: how it works, which are its components and when it should be used. There are also references and comparisons to other currently available libraries implementing Genetic Algorithm in Python and Java. After a brief introduction on the history of MicroGP, it shows the structure and the main genetic operators available in the highly customizable tool. Furthermore, examples are provided to ensure a correct and simple use. Experimental tests have been performed on the One-Max problem (a typical test case used to check performances of Genetic Algorithms). These experiments have the goal of maximizing the bit count (ones or zeros) of a fixed length bit-string. Given the past of MicroGP, another experiment I devised produces a x86 assembly code implementing the One-Max problem. The thesis also included the creation of an online documentation that allows a quick and readable consultation of the source code.

Acknowledgements

I want to take a moment at the end of this journey to thank all the people who supported me through the years of my degree.

Thanks to my parents and my sister who took care of me supporting during the hardest moments of my university and non-university career.

Thanks to all people that shared with me the ups and downs of university experience. A special thanks to my friends Pietro and Gabriele for every time they came through with the help I needed.

Thanks to my colleagues with whom I did group projects: Antonio, Federico, Giuseppe, Luca and Tommaso.

Thanks to my longtime friends who have endured me so far.

Last but not the least, a great thanks to my wonderful girlfriend.. Yes she is real.

Contents

1 Introduction to Evolutionary Computation **19**
- 1.1 Problems to be solved . . . 19
  - 1.1.1 Simulation problem . . . 19
  - 1.1.2 Modeling problem . . . 20
  - 1.1.3 Optimization problem . . . 20
- 1.2 Data feeds Machine Learning . . . 20
- 1.3 Optimization algorithms . . . 21
  - 1.3.1 Global vs. Local Optimization . . . 21
- 1.4 Evolutionary Computation . . . 22
  - 1.4.1 How does a Evolutionary Algorithm work? . . . 23
  - 1.4.2 Multi-Objective optimization problems . . . 24
- 1.5 Genetic Algorithms . . . 24
  - 1.5.1 Individuals and Population . . . 24
  - 1.5.2 Evaluation and fitness . . . 25
  - 1.5.3 Mutation and Crossover . . . 25
  - 1.5.4 Parent and offspring selection . . . 26
  - 1.5.5 Termination criteria . . . 27

2 Background and history of MicroGP **30**
- 2.1 State of the art . . . 30
  - 2.1.1 ECJ - Evolutionary Computation System . . . 30
  - 2.1.2 Inspyred - Bio-inspired Algorithms in Python . . . 30
  - 2.1.3 DEAP - Distributed Evolutionary Algorithms in Python . . . 31
- 2.2 The history of MicroGP . . . 31
  - 2.2.1 The version 1 - MicroGPv1 . . . 31
  - 2.2.2 The version 2 - MicroGPv2 . . . 31
  - 2.2.3 The version 3 - MicroGPv3 . . . 31
- 2.3 Introduction to MicroGPv4 . . . 32
- 2.4 The goal . . . 32

3 Proposed solution **33**
  - 3.0.1 OneMax Problem . . . 33
- 3.1 Constraints definition . . . 33
  - 3.1.1 Parameters . . . 34
  - 3.1.2 Macro . . . 35
  - 3.1.3 Section . . . 35
  - 3.1.4 Individual object . . . 37
  - 3.1.5 Properties . . . 37
- 3.2 Darwin class . . . 38
  - 3.2.1 Operators . . . 39
  - 3.2.2 GenOperator . . . 39

3.2.3 Population . . . 39
3.2.4 Archive . . . 40
3.3 Individual Operators . . . 40
3.3.1 Initialization operator . . . 41
3.3.2 Mutation operators . . . 41
3.3.3 Crossover operators . . . 43
3.4 Fitness and evaluation . . . 46
3.4.1 Order by fitness logic . . . 46
3.4.2 Evaluator . . . 47
3.5 Evolution process . . . 47
3.5.1 Parent selection . . . 49
3.5.2 Offspring selection . . . 49

4 Experiments **50**
4.1 Introduction . . . 50
4.2 OneMax problem - Base . . . 50
4.2.1 OneMax Base version 1 . . . 50
4.2.2 OneMax Base version 2 . . . 52
4.3 OneMax problem - Assembly . . . 55

5 Conclusions and future works **60**
5.1 Future works . . . 61
5.1.1 Statistics for genetic operators selection . . . 61
5.1.2 Stopping conditions . . . 61
5.1.3 2D and 3D categorical sorted parameter . . . 61

List of Figures

1 Left: Genotype of the solution. Right: Phenotype of the solution 16
2 Darwin class evolution logic and parameters. 17
3 Genetic operators implemented for the package. 17
4 Installation of MicroGPv4 package through PIP. 17
5 Online documentation home page. 17

1.1 Schema of a Simulation Problem. 19
1.2 Schema of a Modeling Problem. 20
1.3 Schema of an Optimization Problem. 20
1.4 Schema of a Spam Detection Classifier. 21
1.5 Travel Salesman Problem schema with 4 cities and their distances. 22
1.6 Local and global minimum/maximum. 22
1.7 General schema of an Evolutionary Algorithm. 23
1.8 Example of a Pareto frontier. 24
1.9 Schema of an Individual composed by a chromosome with 9 genes. 25
1.10 Example of population in a Genetic Algorithm. 25
1.11 Single-point Crossover. 26
1.12 Double-point Crossover. 26
1.13 Mutation of three genes. 27

2.1 MicroGPv4 Logo 32

3.1 Parameter class schema. 35
3.2 Section class schema. 36
3.3 Example of a node 37
3.4 Example of parents selected to generate a new individual with Switch procedure crossover. 43
3.5 Example of new solutions produced by Switch procedure crossover. 44
3.6 Example of parents selected to generate a new individual with MacroPool OneCut crossover. 45
3.7 Example of new solutions produced by MacroPool OneCut crossover. 45
3.8 Example of parents selected to generate a new individual with MacroPool Uniform crossover. 46
3.9 Example of new solutions produced by MacroPool Uniform crossover. 46
3.10 Schema of Fitness Tuple class. 47
3.11 Schema of Fitness Tuple Multi-objective class. 47

4.1 Plot of an individual generated for OneMax Base version 1 problem. 53
4.2 Plot of an individual generated for OneMax Base version 2 problem 55
4.3 Plot of an individual generated for OneMax Assembly problem. Black edges are *next* edges and the red ones are `LocalReferences` (jump). 57

5.1 Installation of MicroGPv4 package through PIP 60

5.2 Online documentation home page 60

Listings

3.1 Constraints definition for OneMax problem v2. . . . 33
3.2 Setting up the evaluator function and the fitness type. . . . 34
3.3 Examples of `Parameters` definition. . . . 35
3.4 Create a section of name `word_sec` containing a macro (`word_macro`), it will appear once inside the individual. . . . 36
3.5 Create a section of name `sec_jmp` that contains 1 to 4 macros (`jmp1`), it will appear once inside the individual. . . . 36
3.6 Create a section with a default unique name that contains 2 to 5 macros chosen in `add, sub` and it can appear 0 to 10 times inside the individual. . . . 36
3.7 Build the main section with 3 sections, the second one is a `SubsectionsSequence` that contains 3 sections: a `SubsectionsAlternative sec2a, sec2b`, a simple section `sec_jmp` (MacroPool) a simple section containing a Macro without parameters (MacroPool) . . . 36
3.8 Instantiating a unique Frame. . . . 37
3.9 Create two cumulative (custom) builders and add a checker that tests that two sections have the same number of nodes. . . . 38
3.10 Initialize a `Darwin` object. . . . 38
3.11 Evolve and print results (`Population`). . . . 38
3.12 Print the `Archive` that contains the best eve individuals. . . . 39
3.13 Create and fill an `Operators` object to be passed to a `Darwin` object. . . . 39
3.14 Select k operators that has arity in the given range . . . 39
3.15 Build three genetic operators passing the method and the arity. . . . 39
3.16 Call the method inside the genetic operator. . . . 39
3.17 Add a set of individuals in the population . . . 40
3.18 Add a single individual in the population. . . . 40
3.19 Remove a single individual from the population. . . . 40
3.20 Remove multiple individuals (set) from the population. . . . 40
3.21 Retrieve from population an individual using tournament selection Tournament selection. . . . 40
3.22 Retrieve the entire set of individual contained in the population. . . . 40
3.23 Try to insert an individual in the archive. . . . 40
3.24 Declaration of an evaluator running a script stored in a file. . . . 47
4.1 OneMax Base version 1 - Source code. . . . 50
4.2 OneMax Base version 1 - Example log. . . . 52
4.3 OneMax Base version 2 - Source code. . . . 53
4.4 OneMax Base version 2 - Example log . . . 54
4.5 OneMax Assembly evaluator (eval.bat) . . . 55
4.6 OneMax Assembly possible script solution (phenotype). . . . 56
4.7 OneMax Assembly - Source code . . . 57
5.1 First generation genetic operators selection . . . 61
5.2 Current version of `microgp.darwin.Darwin.evolve()` method . . . 61

List of Algorithms

1 Remove node mutation 41
2 Add node mutation 42
3 Hierarchical mutation 42
4 Flat mutation 43
5 Do generation 48

Summary

Abstract

MicroGP (or μGP) is an optimizer that exploits Evolutionary Computation to find optimal solutions to hard problems. The thesis consists in the complete re-design of the toolkit and in the new implementation in Python language.

1. The goal of the thesis

MicroGP was created almost 20 years ago to generate assembly-language programs used to test microprocessors. The project has evolved in the following years improving the flexibility, usability and functionalities. C and then C++ language was used for the implementation.

The thesis philosophy is to analyze errors and shortcomings of the previous versions which lead to a complete re-design of the tool. At first, defining the individual's structure and using the evolutionary parts required an enormous effort that only an expert programmer could handle. The goal has been to build a simpler and more dynamic evolutionary tool increasing the number of users. As a result, we have selected Python language because it is easy, flexible and highly supported by its community. In order to use the tool, the user only needs to know the Python syntax to define the individuals and genetic operators.

2. Introduction to EC

Evolutionary Computation is a branch of **Computational Intelligence** that consists in iterative optimization of solutions inspired by biological evolution. The kind of algorithms called Evolutionary Algorithms (EA) starts from a bunch of randomly generated solutions and applies a set of operators leading eventually to the optimal ones. The solutions are typically represented with a certain format called *genotype* and their physical representation is called *phenotype* (see Figure 1).

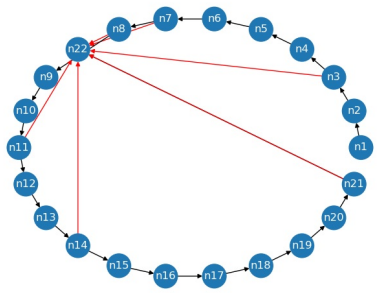
3. New Design

An individual is a directed multi-graph (Figure 1 left) in which the information is encoded in its nodes and edges (*genotype*). The **instruction library**, is used to define the individuals' structure reducing the search space through smart decisions. It is composed by **sections** that contain one or more **macros** (text with zero or more **parameters**). The class called Constraints stores the instruction library and the properties. **Properties** are boxes that can contain values and checkers that run tests on the values. The user can perform any kind of check passing user-defined functions. They are run during each validation phase of any solution.

An individual is composed by **frames** that are unique instances of sections. The solutions evaluator can be a Python method, or a script run by an external program/machine.

The evolution process has been implemented in the *Darwin* class (see Figure 2). Size of the population, initial population size, number of genetic operators used at each generation and other evolution parameters are passed during the creation of this object. A set of genetic operators is passed to *Darwin*, they can be the ones provided by the package or user defined methods. The evolution logic handles efficiently the selection of operators and manages the population keeping the best solutions in an archive that is updated at each generation.

The **parameters** are used to make




```
19      jo n22
20      shr $30, %edx
21      shl $96, %edx
22      shr $105, %edx
23      je n22
24      jz n22
25      sub %edx, %ebx
26      cmp %eax, %edx
27      jz n22
28      shr $62, %ecx
29      sub %edx, %edx
```

```
30      jno n22
31      cmp %ebx, %ecx
32      or %edx, %edx
33      and %edx, %ecx
34      and %ebx, %eax
35      shr $248, %ebx
36      shl $51, %ecx
37      jo n22
38  n22:
39      movl    %eax, -4(%ebp)
40      movl    -4(%ebp), %eax
```

Figure 1: Left: Genotype of the solution. Right: Phenotype of the solution

macros dynamic and describe some individuals genes. They can be of several types: integer, bitstring, categorical, ordered categorical, local reference or external reference. The *reference* parameters act on the edges of the graph linking nodes together. The integer parameter can take a range of values specifying minimum and maximum. During the creation of the categorical and ordered categorical parameters types the user defines a set of alternatives. The bitstring parameter generates strings of bits with the specified length.

MicroGPv4 offers eight basic **genetic operators** (see Figure 3). Three of them are **crossover operators** (or recombination operators), four are **mutation operators**. The last has the goal to **create a random individual** starting from the user-defined instruction library. The initialization differs from the other EAs libraries because usually the logic implementing the creation of the solution is detached from the set of the genetic operators.

One of the mutation operators mutates the graph inserting a new node that contains a valid macro, on the contrary, another operator removes a node from the graph. The remaining mutation operators mutate the values of the parameters inside one or more macros. The operator called switch_proc_crossover copies a particular sequence of nodes from an individual to another one. The other recombination operators swap one or more nodes between two individuals.

The library allows the use multiple types of **fitnesses**: *FitnessTuple*, *FitnessTupleMultiobj*, both inherit from a base class that handles Tuple-like fitnesses. The different selection schemes are implemented simply by overriding the "*greater than*" operator. The sorting logic of individuals with fitness type *FitnessTupleMultiobj* uses Pareto frontier to get a list of individuals ordered considering multiple values to maximize.

4. New possibilities

MicroGPv4 has been engineered to allow the creation and use of user-defined Parameters and Operators. The friendly and flexible logic implemented during the design of the top-level methods makes all this easy and immediate, a sign that the goal we had set has been achieved. Users can be divided in three levels: (1) the user knows Python syntax and uses MicroGP in a Jupyter Notebook exploiting provided parameters and genetic operators; (2) the user knows how to code in Python and can develop its own parameters and genetic operators; (3) an expert coder can re-implement its own evolution process as has been developed for *Darwin class*.

5. Experimental Evaluation

A typical experiment used to test an EA is OneMax problem. The algorithm produces a set of individuals that contain a fixed number of 0s and 1s. The goal is to increase the number of *ones* in the solution. To determine the effectiveness of MicroGP we solved the OneMax problem in three ways. The first version the simplest, the second introduces complexity in the graph structure. The most complex version generates assembly working programs that perform some

computations and return the value stored in a register. An example of a possible result produced is shown in Figure 1.

6. Current release and future works

The released version is a *pre-alpha*, meaning that the tool is fully functional but largely incomplete in terms of functionalities that will be available in the next updates. Two of them are: the introduction of customizable stopping conditions and the implementation of statistics that guide the decisions taken during the selection process of genetic operators.

Due to the fact that this version of MicroGP has been designed to be used by a broad number of users, the state of the art for distribution of FOSS (Free Open Source Software) has been used. The current version of the package has been uploaded on PyPI servers and is installable through a Python de-facto standard package-management system (see Figure 4). The documentation has been deployed with Read the Docs. It contains a precise explanation of how the library works and how to use it. All experiments studied during the development process are there listed and explained. The GitHub repository containing the source code is public.

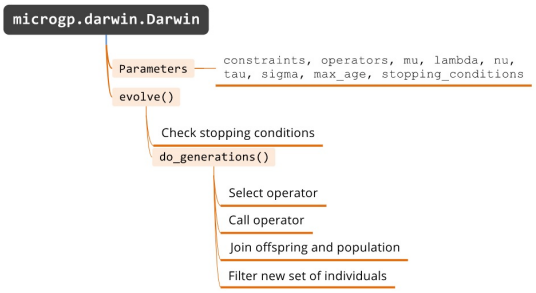


Figure 2: Darwin class evolution logic and parameters.

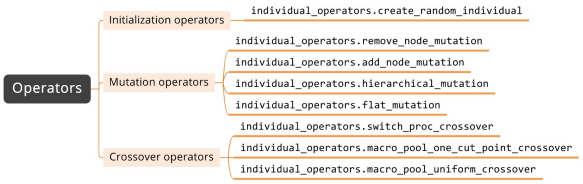


Figure 3: Genetic operators implemented for the package.

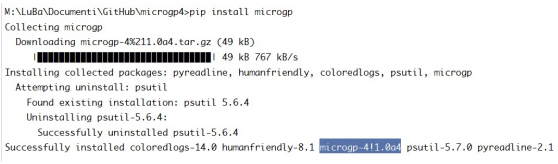


Figure 4: Installation of MicroGPv4 package through PIP.

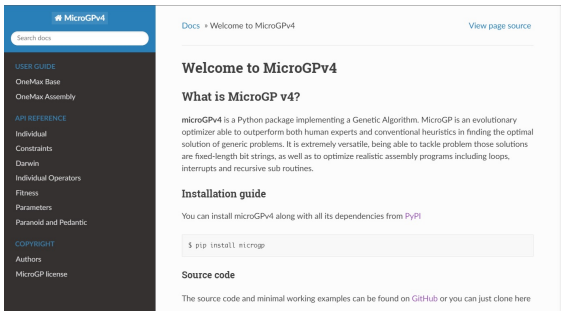


Figure 5: Online documentation home page.

Chapter 1

Introduction to Evolutionary Computation

In the last decades **Artificial Intelligence**[23] has improved lots of technologies. Major smartphone companies are introducing new dedicated chipsets[19] that help the devices to compute data and to make decisions. Some examples can be: photo editing, whether or not to unlock the device with the scanned face[17] or how to answer the questions asked to the voice assistant[31]. Nonetheless improving user experience is not the only application. There are many areas that have taken advantage thanks to the adoption of AI.

This chapter will introduce some AI techniques with a deeper focus on the **optimization algorithms**.

1.1 Problems to be solved

Artificial Intelligence can be applied to solve problems which can be described either mathematically or logically. Among those problems, AI has been used to tackle optimization, modeling, and simulation problems. Such problems can be described by three elements: **input**, **model**, and **output**. An input is a collection of raw information. The model is a mathematical function mapping raw information into a useful output. Depending on the problem, one of these elements is unknown while the others are not.

1.1.1 Simulation problem

In case of a **simulation problem**[18] (Figure 1.1) the known elements are the input and the model; the goal is to find the right output given a certain input. This means that sometimes the output is specified and the simulation must produce a more or less coherent result. Solving this problem means, in most cases, saving money[32] because it is much cheaper to simulate rather than, for example, building a physical circuit and test it.

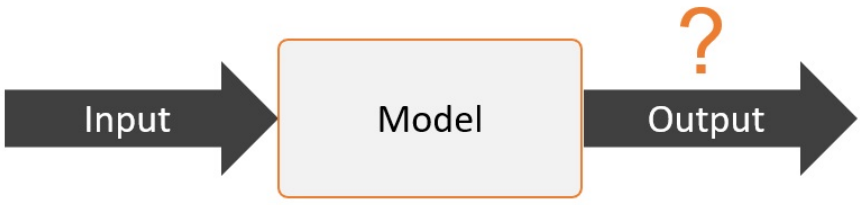


Figure 1.1: Schema of a Simulation Problem.

1.1.2 Modeling problem

A modeling problem is the process of developing a system using mathematics and logic[2]. In this kind of problems the missing element is the model (Figure 1.2). In this case the solution to the modeling problem is a model that accurately maps input data to the output space.

Figure 1.2: Schema of a Modeling Problem.

1.1.3 Optimization problem

An **optimization problem** is the selection of optimal solutions from a set of available alternatives. The model is known, the output is specified (or sometimes known) and the unknown element is the input (Figure 1.3).

An illustrative example can be represented by the shortest path problem. In this case the goal is to find shortest itinerary from a starting point to a pre-defined destination. Each possible sequence of instructions used to reach the target is called *feasible solution*. The goal is to find the optimal solution among the feasible alternatives. This problem can be solved either to minimize the distance travel or the time required to reach the final destination.

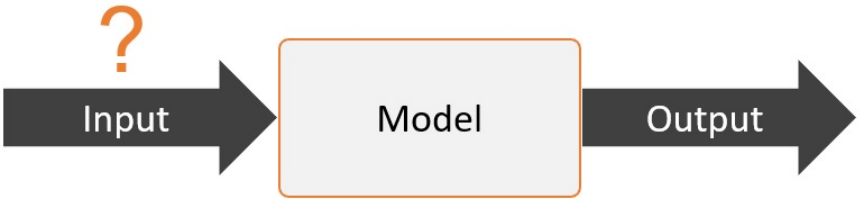


Figure 1.3: Schema of an Optimization Problem.

Optimization problems can be used to optimize the parameters of a modeling problem (i.e. Artificial Neural Network) with the goal of maximizing the model accuracy.

1.2 Data feeds Machine Learning

One of the most popular Artificial Intelligence sub-class is **Machine Learning** (ML)[1]. "Machine learning is the idea that there are generic algorithms that can tell you something interesting about a set of data without you having to write any custom code specific to the problem. Instead of writing code, you feed data to the generic algorithm and it builds its own logic based on the data."[15].

Machine learning can be used to solve modeling problems. For instance, it can be applied to the spam detection problem. The algorithm optimize the *model* (a.k.a. *Classifier*)

parameters with the objective of predicting if an e-mail (*input*) should be considered as spam/not-spam (*output*). See Figure 1.4.

In contrast to other mathematical approaches, the accuracy of ML usually increases with the amount of data used to feed the algorithms.

ML has been used to classify the e-mails as spam or not spam[28], to improve search engine results[4], to improve cybersecurity analyzing data and predicting intrusions or malware[11][14], and many more innovative purposes.

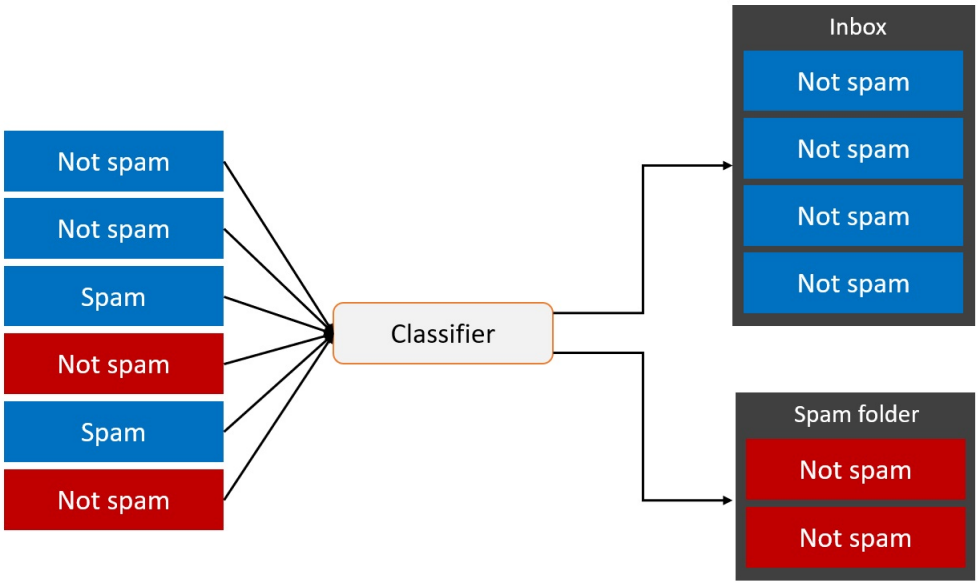


Figure 1.4: Schema of a Spam Detection Classifier.

1.3 Optimization algorithms

Considering the shortest path problem previously described, a navigation system can be driven by an optimization algorithm in order to find the fastest route from a staring point to an other.

This problem is also known as the *Travelling Salesman Problem (TSP)*[13]. The salesman has to visit a set of cities and turn back home. The cities are placed at a certain distance from each other. The objective is to choose the ordered sequence of cities that allows to get smallest possible covered distance. The Figure 1.5 is the abstract representation of the TSP problem: four cities (A, B, C, D), linked by weighted edges (distances).

This model has several applications, for example, the selection of the best route of a network packet travelling from a server to another one[16], reducing the delivery time and speeding up the network.

1.3.1 Global vs. Local Optimization

The challenge, for an optimization algorithm, is to choose the solution good enough to solve the considered problem. Sometimes, the program execution could last too much in terms of running time and then a solution that is not the best one can be considered as the final one. Consequently, the solution can be of two types: **globally optimal**[33] or, more often, **locally optimal**. If the objective is to minimize an error, then the algorithm tries to find the global/local minimum, on the contrary if the objective is to maximize the score, for

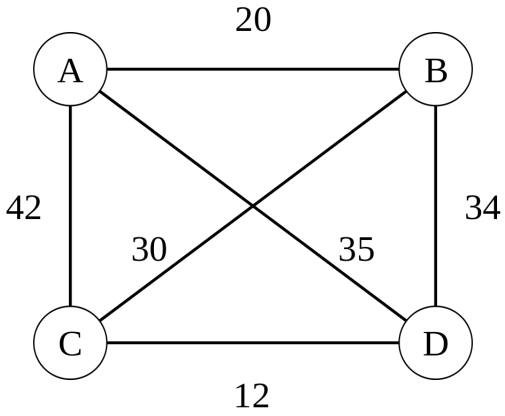


Figure 1.5: Travel Salesman Problem schema with 4 cities and their distances.

example, a game score, accordingly we have to consider the global/local maximum values. The problem requirements could be satisfied with a solution that is not the best ever but achieves a sufficient quality. Termination criteria will be discussed in the next pages.

In Figure 1.6 we have on *x-axis* the parameter that represents the solution and on *y-axis* the quality of the solution. It is important to notice that there could be several local minimum/maximum points and only one global minimum/maximum.

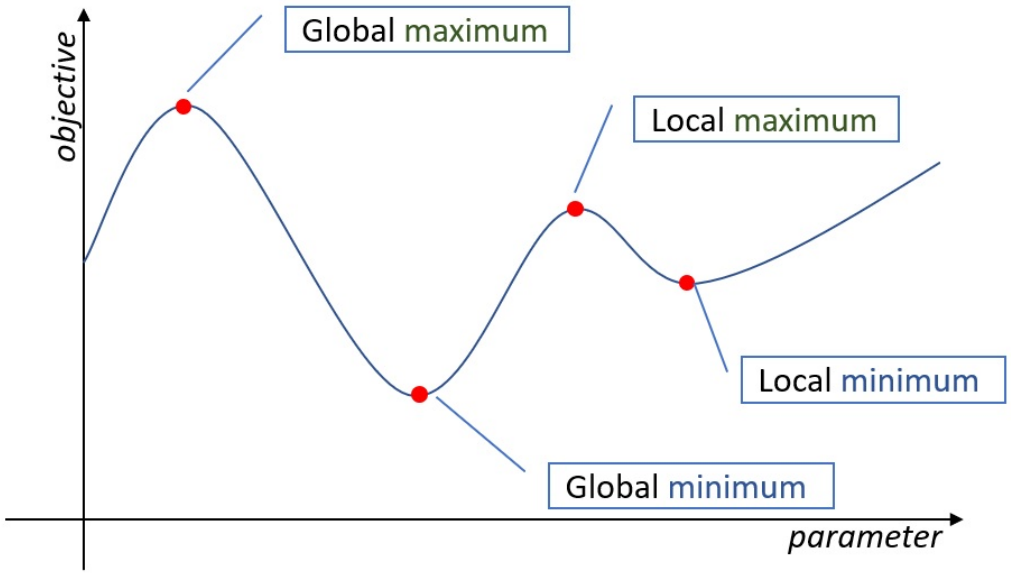


Figure 1.6: Local and global minimum/maximum.

1.4 Evolutionary Computation

Evolutionary Computation[12] is an area within Evolutionary Intelligence which takes inspiration from natural evolution process described by the *Darwin's evolutionary theory*[25].

"*A given environment is filled with a population of individuals that strive for survival and reproduction. The fitness of these individuals is determined by the environment, and relates to how well they succeed in achieving their goals*"[12]. The environment represents the problem to be solved with its output that is described as the goal of minimizing or maximizing a number. This number is correlated to a **fitness** describing a quality. The **individuals** are the candidate solutions (*inputs*) that could or not satisfy the output constraints. In other words this is a method that allows to optimize solutions based on the notion *survival of the fittest* in which the individuals with higher fitness have more chances to be kept in the population of the next generation.

1.4.1 How does a Evolutionary Algorithm work?

The algorithm behind the Evolution Computation is known as **Evolutionary Algorithm** (EA) and can be described with three main characteristics[12]:

1. **Population-Based**: Evolutionary algorithms manipulate individuals trying to get better solutions. The set of the current solutions is called population.

2. **Fitness-Oriented**: A function, called **fitness function**, determines whether a solution is better than another one, through a value called **fitness score**. Each individual (or solution) has one fitness score that can be equal to the one of another individual, despite the differences between the two individuals.

3. **Variation-Driven**: Mechanisms allow to generate variations in individuals letting them be very different from each other. The solutions thus created are then evaluated by the fitness function giving a better or worse score then the previous one.

An EA is divided into several phases. The **Initialization phase** consists in a random generation of individual with determined characteristics depending on the problem. The **Evaluation phase** allows to determine the score (fitness) of each individual that composes a population. The next step is the **Parent selection phase** in which individuals are chosen to create offspring through recombination and mutation. The offspring is then evaluated and selected in the **Survivor selection phase**. The new population is evaluated to determine whether to continue with the next generation or **terminate** the process. This algorithm can be represented with the schema in Figure 1.7.

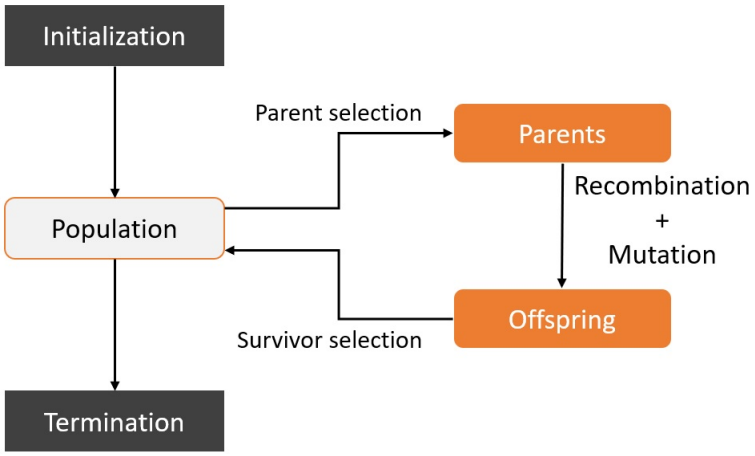


Figure 1.7: General schema of an Evolutionary Algorithm.

1.4.2 Multi-Objective optimization problems

Real problems have several variables that can influence the results in a more or less important way. Therefore, it is necessary to use algorithms that take into account more than one feature during the maximization of the solution score. These algorithms solve Multi-Objective Optimization problems[5].

Genetic Algorithms (GA)[10] are evolutionary algorithms that perform well with this kind of problems (see Section 1.5). They allows to customize the fitness functions and to use methods that promote solution diversity keeping in the population several very different individuals.

An individual is called **nondominated** (or Pareto optimal) if "*if none of the objective functions can be improved in value without degrading some of the other objective values*"[1].

In Figure 1.8 we can see the *Pareto frontier*[22] (red line) defined by the set of Pareto optimal solutions. Consider that the smaller is the value better is the solution. The points on the top-right of the Pareto frontier are individuals dominated by the optimal solutions.

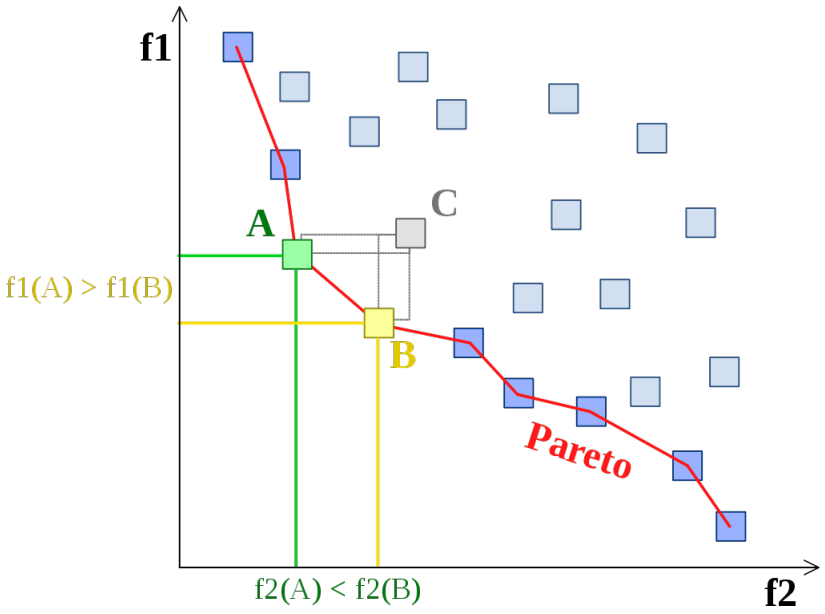


Figure 1.8: Example of a Pareto frontier.

1.5 Genetic Algorithms

A Genetic Algorithm (GA) is a sub-class of Evolutionary Algorithms that has a big random component in it. It is based on Darwin's theory[25] of evolution because the process of solution improvement acts with little and slow changes.

1.5.1 Individuals and Population

Each solution that is generated by a GA is called **individual** (Figure 1.9). It is described by a sequence of features called **genes** that altogether form a **chromosome**. Each chromosome defines how the solution is codified, it can have a variable number of genes and each gene can take any value (**allele**). For instance, an integer number (42), a real number (5.62), a Boolean/binary value (True = 1 or False = 0).

[1]Multi-objective optimization: https://en.wikipedia.org/wiki/Multi-objective_optimization

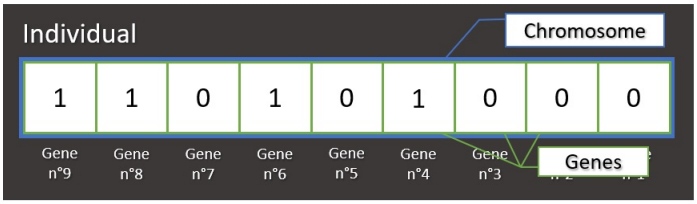


Figure 1.9: Schema of an Individual composed by a chromosome with 9 genes.

The set of individuals is called **population** (Figure 1.10) whose size is defined as **population size** (*popsize*).

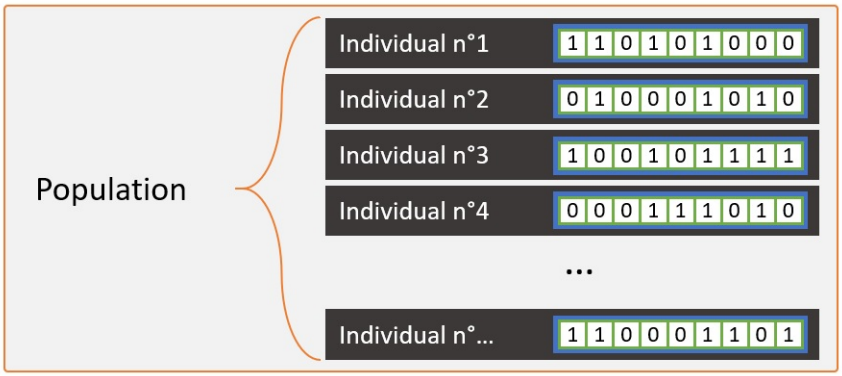


Figure 1.10: Example of population in a Genetic Algorithm.

A **Genotype** is a sequence of genes representing a chromosome, while the physical representation of the chromosome is called **Phenotype**. For example, considering the Figure 1.9, if the genotype of an individual is the bit-string `110101000` its phenotype could be the decimal representation `424`.

1.5.2 Evaluation and fitness

Each individual can be associated with a value called **fitness value** produced by the **fitness function** on the basis of the final goal and the chromosome (see Subsection 1.4.2). The phase in which the fitness values are computed is called **evaluation phase** (see Section 1.4.1).

1.5.3 Mutation and Crossover

The first population is created randomly during the first **generation**, to produce a new population, modifications have to be applied to the individuals. The operators that perform the variation are called **Variation Operators** and they are: **crossover (or recombination) operator** and **mutation operator**.

Crossover

The **crossover** consists of the creation of one or more individuals that take some genes from an individual and some others from another. The number of genes taken from each parent and the point of the chromosome from which they are taken is random. This works in a similar way to the natural reproduction in which some characteristics are taken from

the father and others from the mother of a new born. The amount of new individuals thus produced, varies from the problem specifications and its whole is called **offspring**.

In Figure 1.11 you can see an example of a **single-point crossover** (or one-point cut crossover) in which a point in the chromosomes is chosen and the new individuals will be composed by a combination of parents.

The same behavior can be exploited by choosing two points instead of one (see Figure 1.12).

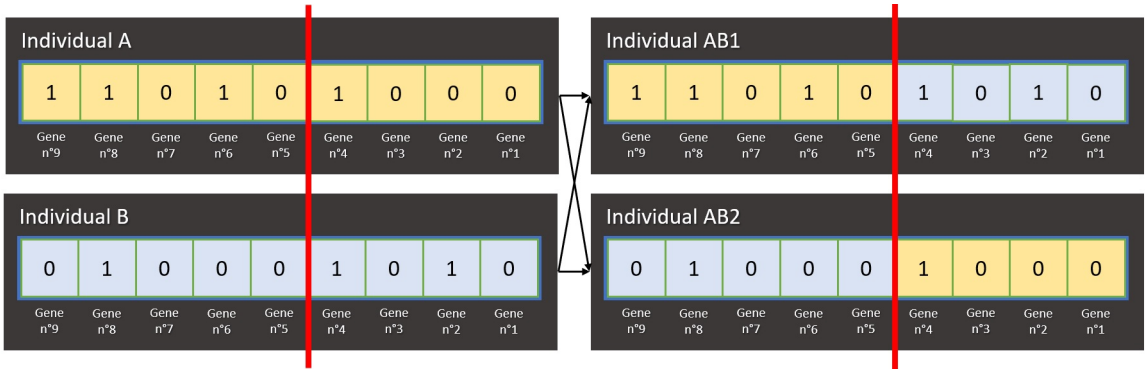


Figure 1.11: Single-point Crossover.

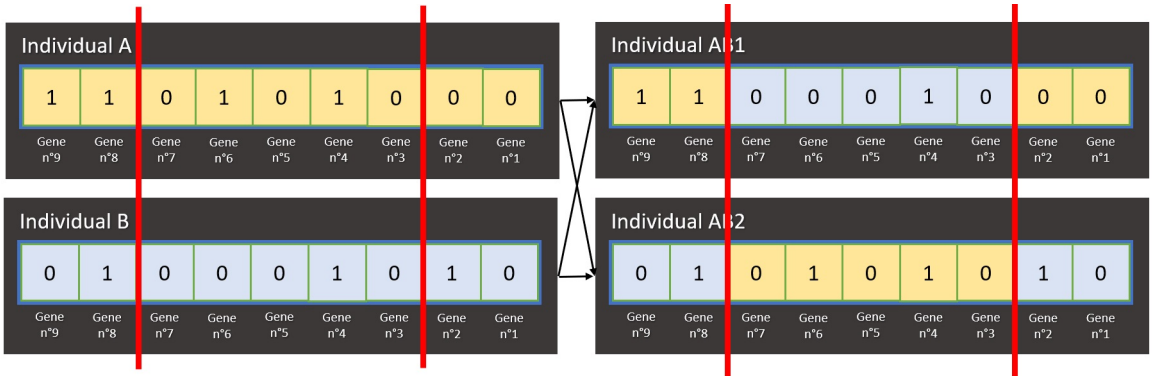


Figure 1.12: Double-point Crossover.

Mutation

The individuals thus created are then modified by the **mutation operator**. This operator chooses randomly one or more genes and, with a given probability, mutates them assigning random values (alleles). For instance the algorithm decides that the genes 3, 4 and 9 will be mutated producing a new individual (see Figure 1.13).

1.5.4 Parent and offspring selection

The individuals that are selected for the crossover and mutation are called **parents**, they are chosen in the **parent selection** phase (see Section 1.4.1) based on the fitness score. Usually the higher is the score, the more likely it is to be chosen. The set of the selected individuals is called **mating pool**; in some cases the parents can be more than two. The selection of individuals with high fitness scores allows to create new solutions that will probably have high score too.

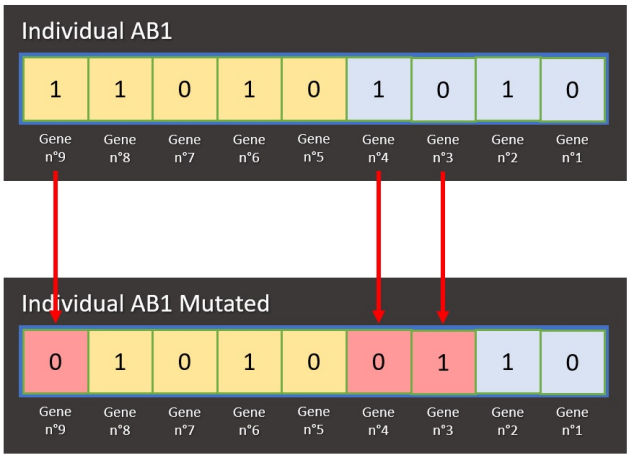


Figure 1.13: Mutation of three genes.

After the mutation the offspring can't be simply joined to the population, here is where the **offspring selection** takes place. There are several techniques of selection, one of the simplest is the one that joins the offspring to the population, rank it and keep only the *popsize* best individuals, removing the worst ones preventing them from reproducing.

In Table 1.1 on the top-left there is the tabular representation of the population after the recombination and mutation phase. On the bottom is shown the population and the offspring ranked together. In the last table (top-right) the new population after the selection that has removed the worst individuals

It is trivial to come to conclusion that after a certain number of generations the population will contain solutions with a fitness score higher than the fitness score of the first randomly generated population.

1.5.5 Termination criteria

The evolution keeps running until one or more **stopping conditions** occur. Stopping conditions depend on the goal of the use case, some of them could be:

- Limited amount of run time;
- Limited number of generations run by the algorithm;
- Limited number of evaluations of the fitness score;
- Slow changes in individuals among generations;
- Fitness score good enough.

	Individual	Fitness score
1	A	10
2	B	23
3	C	22
4	D	14
5	E	3
6	F	44

Individual	Fitness score
F	**44**
B	**23**
C	22
D	14
A	10
E	3

	Individual	Fitness score
1	F	44
2	FB1	39
3	B	23
4	C	22
5	FB2	15
6	D	14
	A	10
	E	3

Table 1.1: Select the best *popsize* individuals after ranking them by fitness score.

Chapter 2

Background and history of MicroGP

2.1 State of the art

Online there are several libraries that implement Evolutionary Programming (or Genetic Programming), this is a clear sign that there is a great interest in solving problems through an optimizer. There are different use cases such as industrial, scholars and research. In this chapter we give a brief introduction to three of the most used EC frameworks.

2.1.1 ECJ - Evolutionary Computation System

ECJ [1] is a java library for implementing Evolutionary Computation developed at George Mason University's ECLab (Evolutionary Computation Laboratory). It was presented at GECCO 2019 with the paper *ECJ at 20: Toward a General Metaheuristics Toolkit* [29] and it received a grant from *NFS* (National Science Foundation) for improving and enhancing its features.

The library has a basic GUI that can be used for loading, running jobs, managing parameters and plot statistics; it is multi-threading ready and the solutions can be coded also with tree and vector representations. There is the possibility to use multiple operators and it can hook for other multi-objective optimization methods. There is also a user manual online [2].

2.1.2 Inspyred - Bio-inspired Algorithms in Python

Developed by taking into account the book "Evolutionary Computation: A Unified Approach."[9], **Ispyred** is a Python package that uses bio-inspired algorithms. Because of the variety of problem that a Genetic Algoritm can solve, the Ispyred developers design their code with a central principle: keep separate the problem specific components from the algorithm-specific components. This approach allows to make algorithms as flexible and generic as possible. The library is composed in two main components: problem-specific components and algorithm-specific evolutionary components. More information are available on the website[3]

[1] ECJ: https://www.parabon.com/dev-center/origin/ecj/index.html
[2] ECJ Manual: https://cs.gmu.edu/ eclab/projects/ecj/manual.pdf
[3] Inspyred documentation: https://pythonhosted.org/inspyred/overview.html

2.1.3 DEAP - Distributed Evolutionary Algorithms in Python

Inspyred is not the only EC project implemented in Python, another example is **DEAP (Distributed Evolutionary Algorithms in Python)**. Its strength is the ease with which you can define the structures used for the solutions. The main difference with the other libraries is the ability to create types with the specifics given by the user, that will also have to select the most suitable operator. There is also the possibility to build your own operator. The documentation is available in "*Read the Docs*" format[4]

2.2 The history of MicroGP

As you can imagine the MicroGPv4 is not the first version, several versions in several programming languages have been developed, improving the capabilities and introducing new features and higher flexibility.

2.2.1 The version 1 - MicroGPv1

After almost a year of studies, the first version of MicroGP was created, it was the 2001 and the goal was to generate *real* assembly programs through an evolutionary algorithm. The code was composed of hundreds of C lines and some scripts with the purpose of testing feasibility of the idea (see *Efficient machine-code test-program induction*[7]).

The algorithm produced an individual in the form of directed acyclic graphs that were modified by changing the topology and modifying the parameters present inside the nodes. The final program was then built to be evaluated, starting from the graph.

2.2.2 The version 2 - MicroGPv2

The new version uses directed multi-graphs to encode the structure of the individuals and allows to operate with specific microprocessors thanks to the ability of loading the list of the parametric code fragments (*macros*) from a file. The core logic was completely changed in order to embrace the Genetic Programming paradigm (hence the GP in the name). The programming language remains C and its development has been supported by Intel through a grant named "GP Based Test Program generation".

Some examples of test program generation for design validation of microprocessors can be read in the paper *MicroGP — An Evolutionary Assembly Program Generator*[30]. A subsequent re-design lead to a re-implementation in C++.

The tool has been used by engineers for the test and verification of small microprocessors, it is also possible to scale up tackling of a real Pentium 4 (see *Automatic test programs generation driven by internal performance counters*[21]). The first machine-written programs able to become *King of the Hill* in all core-war competitions was created by MicroGPv2 (see *Efficient machine-code test-program induction*[8]). Thanks to its flexibility it has been used to design bayesian networks, create mathematical functions represented as three, integer and combinatorial optimization, real-value parameter optimization.

2.2.3 The version 3 - MicroGPv3

Started in 2006 and released in different versions spread from 2010 to 2016, the MicroGPv3 brought a complete change of paradigm. The new goal was the generalization of the library which allows the users finding solutions to a wider range of applications.

[4]DEAP documentation: https://deap.readthedocs.io/en/master/index.html

A description of the version released in 2010 (called *Bluebell*) can be found in the book *Evolutionary Optimization: the microGP toolkit*[27]

2.3 Introduction to MicroGPv4

Figure 2.1: MicroGPv4 Logo

MicroGP is an optimizer that adopts Genetic Programming to find solutions to hard problems. Starting from a bunch of randomly generated solutions, it applies a set of already provided operators to the solutions changing their characteristics. The solutions that has closer fitness score to the ideal solution are kept. The process repeats itself until an optimal solution is found. This technique allows to explore the search space efficiently.

2.4 The goal

As mentioned above, there are several releases of MicroGP, the last one is the "v4" and for the first time Python language has been used for the implementation.

Looking on the web, there is a big variety of libraries that implement evolutionary algorithms, it is therefore clear the need for this kind of tools. The main uses are: industrial, research and teaching. The v2 and v3 has been widely used for industrial purposes, but a new target has been taken into account with the new version, it has been developed also for research and teaching purpose. The package can be easily installed and used looking at the examples inside it.

MicroGP allows to keep detached the evaluation of the individual and its generation. This is an advantage for **industrial uses** because a company can execute the evaluation on its machines without giving any industrial information to the developers in charge of building the others evolutionary parts.

Individuals can be built with a very high structuring. With the other EC libraries the possibilities of the solutions structures are restricted, for example with DEAP you can define any kind of phenotype but it isn't structured. With MicroGP I can build a procedure that enters and exits from the protected mode of a Intel Pentium; this is impossible with the other tools.

Chapter 3

Proposed solution

The aim of this chapter is to explain to the reader how MicroGPv4 package has been designed, in order to ensure an easy and correct use of the tool.

As previously said the programming language used is Python v3.7[1]. The individual is composed by a graph managed by *NetworkX v2.3*[2], the log is colored by *coloredlogs v4*[3] and the plots of the solutions require *Matplotlib v3.1*[4].

3.0.1 OneMax Problem

We will consider the **OneMax problem**[6] as an example of a simple problem solvable with Genetic Algorithms. Given a sequence of N random bits composing a bitstring (i.e. `10001010`), the fitness score is given based on the number of *ones* present in it (higher is better). The algorithm must generate a random set of individuals (strings of bits), evolve them till they will contain only *ones*.

3.1 Constraints definition

MicroGP allows an amazing flexibility in building the possible solutions. The structure of an individual can be described through **sections** (`microgp.constraints.Section`) that contain one or more **macros** (`microgp.macro.Macro`). A **macro** is a fragment of text with zero or more variable **parameters** (`microgp.parameter.Parameter`).

The first thing to do is to define the **constraints** that all solutions must follow. The `microgp.constraints.Constraints` class contains the set of macros and sections of which an individual is composed. Futhermore, you can define a set of properties (`microgp.properties.Properties`) that will execute tests on the structure of the solution. The fitness score will be computed by an **evaluator** stored as attribute of the Constraint object.

The backbone of an individual is called **instruction library**. It must be defined in this order: parameters, macros, sections and lastly the main section. Here is an example of a the instruction library for the One Max problem (version 2) in which the individual is composed by 8 macros with, each one, one parameter that can take two values (`0` or `1`):

```
1 # Define a parameter of type ugp.parameter.Categorical that can take two values: 0
      or 1
```

[1]Python: https://www.python.org/
[2]NextowkX: https://networkx.github.io/
[3]coloredlogs - PyPI: https://pypi.org/project/coloredlogs/
[4]Matplotlib: https://matplotlib.org/

```
2 bit = ugp.make_parameter(ugp.parameter.Categorical, alternatives=[0, 1])
3 # Define a macro that contains a parameter of type ugp.parameter.Categorical
4 word_macro = ugp.Macro("{bit}", {'bit': bit})
5 # Create a section containing a macro that will appear 8 times
6 word_section = ugp.make_section(word_macro, size=(8, 8), name='word_sec')
7 # Create a constraints library
8 library = ugp.Constraints()
9 library['main'] = ["Here is the bitstring", word_section, ""]
```

Listing 3.1: Constraints definition for OneMax problem v2.

After that, the fitness type is chosen and the function that will evaluate the solutions is set (See Listing 3.2). For more information about the evaluation function see Subsection 3.4.2.

```
1 # Define the evaluator and the fitness type
2 def my_script(filename: str):
3     with open(filename) as file:  # Use file to refer to the file object
4         data = file.read()
5         count = data.count('1')
6     return list(str(count))
7
8 library.evaluator = ugp.fitness.make_evaluator(script=my_script, fitness_type=ugp.
      fitness.Lexicographic)
```

Listing 3.2: Setting up the evaluator function and the fitness type.

3.1.1 Parameters

The *parameters* are used to make the macros dynamic. For example you can build a parameter of type `microgp.parameter.categorical.Categorical` passing some values (alternatives); the macro that contains this kind of *parameter* will take the given values and the resulting text of the macro will depends on the values taken by the assigned *parameters*.

The method `microgp.parameter.helpers.make_parameter` allows to build a parameter providing the parameter class type and the needed attributes (See Listing 3.1 line 2).

There are several types of parameters (Figure 3.1):

- `microgp.parameter.categorical.Categorical`, it can assume one value among the specified unsorted alternatives;
- `microgp.parameter.categorical.CategoricalSorted`, it can assume one value among the specified sorted alternatives;
- `microgp.parameter.bitstring.Bitstring`, it can assume a fixed-length bitstring value
- `microgp.parameter.integer.Integer`, it can assume an integer value between a specified minimum and maximum;
- `microgp.parameter.reference.LocalReference`, it is used to reference a Node *connected* through (undirected) "next" edges;
- `microgp.parameter.reference.ExternalReference`, it is used to reference a Node *non-connected* through (undirected) "next" edges;
- `microgp.parameter.special.Information`, dummy Parameter whose purpose is to insert information about the state of the system into the individual.

Here is an example of `Parameters` definition:

```
registers = ugp.make_parameter(ugp.parameter.Categorical, alternatives=['ax', 'bx', 'cx', 'dx'])
cat_sor = ugp.make_parameter(ugp.parameter.CategoricalSorted, alternatives=['e', 'f', 'g', 'h', 'i', 'l'])
bitstring8 = ugp.make_parameter(ugp.parameter.Bitstring, len_=8)
uint256 = ugp.make_parameter(ugp.parameter.Integer, min=0, max=256)
int256 = ugp.make_parameter(ugp.parameter.Integer, min=-128, max=128)
ref_fwd = ugp.make_parameter(ugp.parameter.LocalReference,
                             allow_self=False,
                             allow_forward=True,
                             allow_backward=False,
                             frames_up=1)
ref_bcw = ugp.make_parameter(ugp.parameter.LocalReference,
                             allow_self=False,
                             allow_forward=False,
                             allow_backward=True,
                             frames_up=1)
proc1 = ugp.make_parameter(ugp.parameter.ExternalReference, section_name='proc1', min=5, max=5)
```

Listing 3.3: Examples of `Parameters` definition.

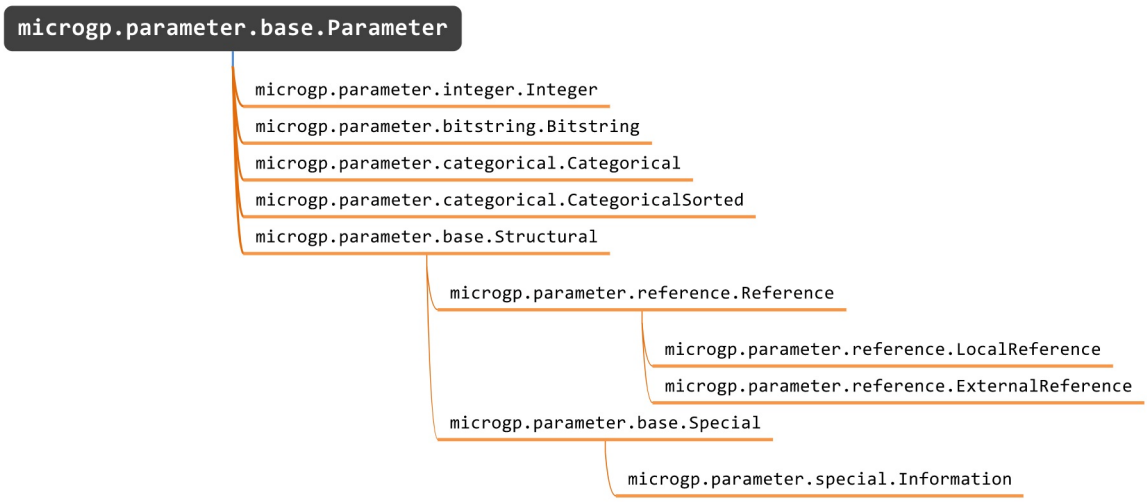


Figure 3.1: Parameter class schema.

3.1.2 Macro

A macro is made of a string that contains zero or more previously built *parameters*. The macro builder receives a string, a dictionary containing the names of the parameters and their types (classes, not objects) (See Listing 3.1 line 4).

3.1.3 Section

A section (Figure 3.1) is composed by one or more macros, a set or a list of other subsections:

- `microgp.constraints.MacroPool`: pool of macros;
- `microgp.constraints.RootSection` (root section of an individual)
- `microgp.constraints.SubsectionsSequence`: sequence of sections;

- `microgp.constraints.SubsectionsAlternative`: sequence of sections that can be alternatively chosen.

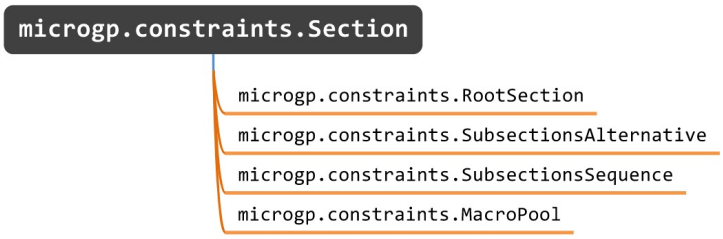


Figure 3.2: Section class schema.

Each section can be built by the `microgp.constraints.make_section` method specifying the pool of macros contained (one or more) (`section_definition`), the name of the section (`name`), the number of macros that the section can contain (`size`), how many times the section can appear inside an individual (`instances`) and how to translate a node into string `label_format`.

The first parameter (`section_definition`) can be a macro, a list or a set of macros, a string, a list or a set of sections; if a list is passed to the method, then a `SubsectionsSequence` will be created, otherwise if the passed object is a set, the method will create a `SubsectionsAlternative` containing the given set of macros/sections.

Examples:

```
word_section = ugp.make_section(word_macro, size=(1, 1), name='word_sec')
```

Listing 3.4: Create a section of name `word_sec` containing a macro (`word_macro`), it will appear once inside the individual.

```
sec_jmp = ugp.make_section(jmp1, size=(1, 4), name='sec_jmp')
```

Listing 3.5: Create a section of name `sec_jmp` that contains 1 to 4 macros (`jmp1`), it will appear once inside the individual.

```
generic_math = ugp.make_section({add, sub}, size=(2, 5), instances=(0, 10))
```

Listing 3.6: Create a section with a default unique name that contains 2 to 5 macros chosen in `add, sub` and it can appear 0 to 10 times inside the individual.

```
library['main'] = [
    'Prologue...'
    [{sec2a, sec2b}, sec_jmp, '; this is a comment'],
    'Epilogue...'
]
```

Listing 3.7: Build the main section with 3 sections, the second one is a `SubsectionsSequence` that contains 3 sections: a `SubsectionsAlternative` `sec2a, sec2b`, a simple section `sec_jmp` (MacroPool) a simple section containing a Macro without parameters (MacroPool)

3.1.4 Individual object

The `microgp.individual.Individual` object represents a possible solution, the `Individual.graph` attribute is a NetworkX[5] MultiDiGraph[6], the nodes, inside it, are identified by their unique identifier (`microgp.node.NodeID` object that inherits from `int`). Each **node** (Figure 3.3) contains:

- a `microgp.macro.Macro` object;
- a dictionary that contains `microgp.parameter.Parameter` as values;
- a tuple of `microgp.common_data_structures.Frame`.

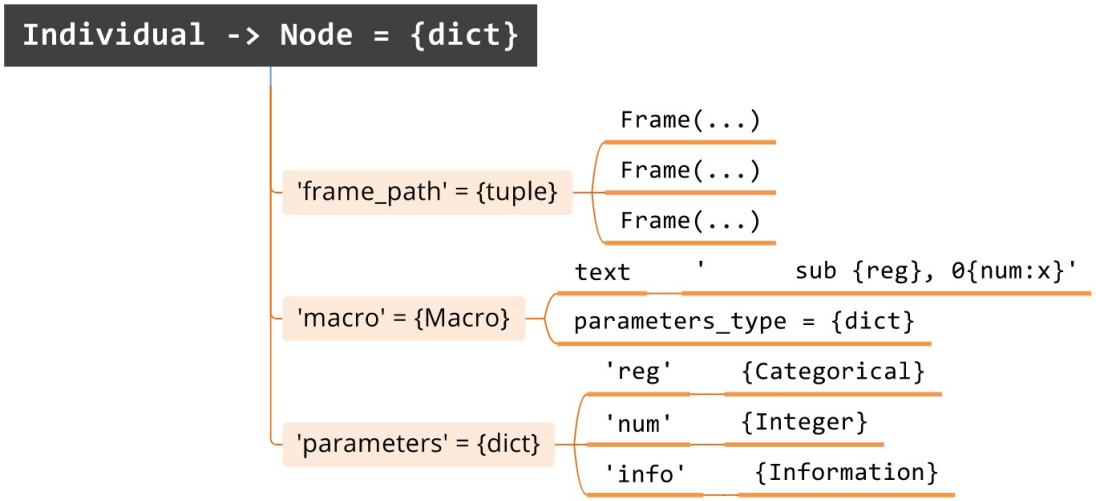


Figure 3.3: Example of a node

A **Frame** is a unique instance of a section, its name is unique and can be generated by the `microgp.individual.get_unique_frame_name()` method in the following way:

```
1 new_unique_frame = Frame(individual.get_unique_frame_name(section), section)
```

Listing 3.8: Instantiating a unique Frame.

3.1.5 Properties

Properties are boxes that can contain values and checkers that run tests on the values. The testers can return `True` or `False`. Values in Properties can be customized, for instance a value can be the number of macros in a certain section and can be set a checker on it that checks that this values doesn't exceed a certain threshold.

Builders can be:

- `custom_builders`: customizable by the user;
- `default_builders`: builders provided with MicroGP package.

Another distinction:

- `base_builders`: I can set a certain value;

[5]NetworkX: https://networkx.github.io/

[6]NetworkX MultiDiGraph: https://networkx.github.io/documentation/networkx-1.9.1/reference/classes.multigraph.html

- `cumulative_builders`: I can set a value and it can be added up recursively going through the frame tree.

Properties are used to check if a frame (i.e. the portion of the individual implementing a given section) is valid. First, all functions registered as 'values builders' are called, then all functions registered as 'check' are evaluated; if all succeeded, then True is turned.

'Values' are divided in 'custom' and 'base'. User's builders build custom ones. Values can be retrieved through property 'values' (i.e. `microgp.properties.Properties.custom_values()`) that merge the two, alternatively they can be retrieved through properties 'base_values'and 'custom_values'.

'Values builders' are functions returning a dictionary of values {'value_name': value} that is added to the current value-bag. Values cannot be shadowed.

'Checks' are called when the value bag is complete and get getting all values as parameters, i.e. 'check(**values)'.

Here is an example:

```
sec2a.properties.add_cumulative_builder(lambda num_nodes, **v: {'sec2a': num_nodes
    })
sec2b.properties.add_cumulative_builder(lambda **v: {'sec2b': v['num_nodes']})
library.global_properties.add_check(lambda sec2a, sec2b, **v: sec2a == sec2b)
```

Listing 3.9: Create two cumulative (custom) builders and add a checker that tests that two sections have the same number of nodes.

3.2 Darwin class

This class manages the evolution, stores the genetic operators, the population, and the archive. You can set some evolution parameters (*lambda, tau, nu, sigma, mu*) and a list of stopping conditions (in the future releases). A `microgp.population.Population` and a `microgp.population.Archive` objects are also initialized.

```
mu = 10
nu = 20
sigma = 0.2
lambda_ = 7
max_age = 10
darwin = ugp.Darwin(
    constraints=library,
    operators=operators,
    mu=mu,
    nu=nu,
    lambda_=lambda_,
    sigma=sigma,
    max_age=max_age)
```

Listing 3.10: Initialize a `Darwin` object.

```
darwin.evolve()
logging.bare("This is the population:")
for individual in darwin.population:
    msg = 'Printing individual ' + individual.id
    ugp.print_individual(individual, msg=msg, plot=True)
    ugp.logging.bare(individual.fitness)
```

Listing 3.11: Evolve and print results (`Population`).

```
logging.bare("These are the best ever individuals:")
ugp.print_individual(darwin.archive)
```

Listing 3.12: Print the `Archive` that contains the best eve individuals.

3.2.1 Operators

This class wraps all genetic operators, manages statistics (in the future versions), and GenOperator selection. The selection is made on the basis of the arity and the success and failure statistics of the operator.

```
operators = ugp.Operators()
init_op1 = ugp.GenOperator(ugp.create_random_individual, 0)
operators += init_op1
mutation_op1 = ugp.GenOperator(ugp.remove_node_mutation, 1)
operators += mutation_op1
crossover_op1 = ugp.GenOperator(ugp.switch_proc_crossover, 2)
operators += crossover_op1
crossover_op2 = ugp.GenOperator(ugp.five_individuals_crossover, 5)
operators += crossover_op2
```

Listing 3.13: Create and fill an `Operators` object to be passed to a `Darwin` object.

```
selected_operators = operators.select(max_arity=0, k=10)
selected_operators = operators.select(min_arity=1, max_arity=2 k=20)
```

Listing 3.14: Select k operators that has arity in the given range

3.2.2 GenOperator

Wrapper of a method that implements the algorithm manipulating or building one or more individuals. This class will also manage (in the future versions) the statistics applied to the assigned method.

The method wrapped in the GenOperator must have `**kwargs` in its parameters.

```
init_op1 = ugp.GenOperator(ugp.create_random_individual, 0)
mutation_op1 = ugp.GenOperator(ugp.remove_node_mutation, 1)
crossover_op1 = ugp.GenOperator(ugp.switch_proc_crossover, 2)
crossover_op2 = ugp.GenOperator(ugp.five_individuals_crossover, 5)
```

Listing 3.15: Build three genetic operators passing the method and the arity.

```
selected_crossover_genoperator(individual1, individual2)
selected_mutation_genoperator(individual, sigma=0.7, constraints=constraints))
individuals = tuple(ind1, ind2, ind3, ind4, ind5)
kwargs = {'param1': var1, 'param2': var2, 'param3': [a, b, c, d, e]}
selected_crossover_5_individuals(*individuals, kwargs)
```

Listing 3.16: Call the method inside the genetic operator.

3.2.3 Population

This class contains the set of individuals composing the **population** and manages the selection, insertion, removal of the individuals based on the age or whether their phenotype is already in the population (future versions). A non-valid individual can't be inserted.

```
darwin.population += set(list_of_individuals)
```

Listing 3.17: Add a set of individuals in the population

```
darwin.population += set(individual_A)
darwin.population += set(individual_B)
```

Listing 3.18: Add a single individual in the population.

```
darwin.population -= set(individual_A)
```

Listing 3.19: Remove a single individual from the population.

```
darwin.population = darwin.population - set(individual_A)
```

Listing 3.20: Remove multiple individuals (set) from the population.

```
selected_individual = darwin.tournament_selection(tau)
```

Listing 3.21: Retrieve from population an individual using tournament selection Tournament selection.

```
population = darwin.population.individuals
```

Listing 3.22: Retrieve the entire set of individual contained in the population.

3.2.4 Archive

This class manages the set of individuals not dominated by all other individuals currently or previously contained in the `microgp.darwin.Darwin._population`.

```
self._archive += individual
```

Listing 3.23: Try to insert an individual in the archive.

The individual will be inserted only if it is not dominated by all individual already inside the archive. If it is not dominated then the individuals that just became dominated are removed from the archive.

3.3 Individual Operators

MicroGPv4 package offers some basic variation operators that can be found in `microgp.individual_operators`. Three of them are **crossover operators**, four are **mutation operators** and one has the goal to **create a random individual**. As you can see during the Dawin object declaration, the methods that are listed in this module can be passed to the constructor of a GenOperator an then added to the list of operators used by the `microgp.darwin.Darwin` object.

3.3.1 Initialization operator

The method **Create random individual** creates a random individual. Individuals are created starting from section main. The new individual is eventually checked using `is_valid`. If invalid it is discarded and the generation process starts over. The function eventually gives up after a number of tries and rises an exception.

3.3.2 Mutation operators

This kind of operator change one or more genes that describe an individual. The intensity of the change and the probability that it takes place depend on *sigma*. This is a parameter specified during the creation of the `Darwin` object and it can assume values in [0, 1].

The **Remove node mutation** method tries to remove a node taken from the possible set of nodes in the individual. The removal could fail because of the minimum number of nodes that the individual must contain. This method returns a modified copy of the passed individual leaving it unchanged (see Pseudocode 1).

Algorithm 1 Remove node mutation

```
1:  procedure REMOVE_NODE_MUTATION
2:      new_individual ← copy of original_individual
3:      do
4:          if num of nodes in new_individual <= 2 then: return Empty list
5:          shrinkable_frames ← (list of frames that have min_size) <= (num of
    nodes in frame) - 1
6:          if shrinkable_frames is empty then return Empty list
7:          chosen_frame ← choose a random frame in shrinkable_frames
8:          node_to_remove ← choose a random node in chosen_frames
9:          if node_to_remove == entry_point of new_individual then
10:             entry_point ← next node of node_to_remove
11:         remove node_to_remove from new_individual
12:         for all nodes of the 'next-chain' in which node_to_remove was contained
    do
13:             if current_node has a parameter of type LocalReference with des-
    tination = node_to_remove then
14:                 mutate the parameter
15:     while random number > sigma
16:     finalize new_individual
17:     if new_individual is not valid then return Empty list
18:     elsereturn List with the new_individual
```

The **Add node mutation** inserts a new node in the individual graph. An insertion of a new node could fail because of there are no valid targets for the node that contains a `LocalReference`. See Pseudocode 2).

The **Hierarchical mutation** builds a list of all parameters contained in all nodes then choose one of them and mutate it. See Pseudocode 3.

The **Flat mutation** builds a list of all parameters contained in all nodes then choose one of them and mutate it. This mutation differs fron the previous one in the probability with which the `Parameter` is chosen. See Pseudocode 4.

Algorithm 2 Add node mutation

```
1: procedure ADD_NODE_MUTATION
2:     new_individual ← copy of original_individual
3:     do
4:         if num of nodes in new_individual <= 2 then: return Empty list
5:         expandable_frames ← (list of frames that have max_size) >= (num of nodes in frame) + 1
6:         if expandable_frames is empty then return Empty list
7:         chosen_frame ← choose a random frame in expandable_frames
8:         chosen_parent_node ← choose a random node in chosen_frame
9:         chosen_macro ← choose a random macro in the MacroPool of chosen_parent_node
10:        node_to_insert ← insert in new_individual a new node with parent = chosen_parent_node
11:        initialize macros (chosen_macro) of node_to_insert
12:    while random number > sigma
13:    finalize new_individual
14:    if new_individual is not valid then return Empty list
15:    elsereturn List with the new_individual
```

Algorithm 3 Hierarchical mutation

```
1: procedure HIERARCHICAL_MUTATION
2:     new_individual ← copy of original_individual
3:     do
4:         while True do
5:             chosen_node ← choose a random node among all nodes in new_individual
6:             for all parameters in chosen_node do
7:                 if current_parameter is not of type Information then
8:                     candidate_parameters ← candidate_parameters + current_parameter
9:             if candidate_parameters is not empty then
10:                chosen_parameter ← choose a random parameter in candidate_parameters
11:                mutate the chosen_parameter
12:                break
13:    while random number > sigma
14:    finalize new_individual
15:    if new_individual is not valid then return Empty list
16:    elsereturn List with the new_individual
```

Algorithm 4 Flat mutation

```
1: procedure FLAT_MUTATION
2:     new_individual ← copy of original_individual
3:     do
4:         for all nodes in new_individual do
5:             for all parameters in current_node do
6:                 if current_parameter is not of type Information then
7:                     candidate_parameters ← candidate_parameters + current_parameter
8:         if candidate_parameters is empty then return Empty list
9:         chosen_parameter ← choose a random parameter in candidate_parameters
10:        mutate the chosen_parameter
11:    while random number > sigma
12:    finalize new_individual
13:    if new_individual is not valid then return Empty list
14:    elsereturn List with the new_individual
```

3.3.3 Crossover operators

Switch procedure crossover: let's consider a sequence of nodes connected through edges with label= 'next', we will call this sequence *next-chain*. This operator selects the next-chains belonging to the common sections between the two parents (excluding the main section) and link a random node with `ExternalReference` parameter in it, to the selected *next-chain* after copying it from the source to the destination individual. The word 'proc' in method name refers to the *next-chains* that are not 'main' sections. The destination individual is a copy of a parent chosen randomly and then modified with the new cloned *next-chain*.

In Figure 3.4 you can see two possible parents individuals that are used to generate a new solution; the characters in the circles indicates generic macros. In Figure 3.5 there are some examples of what this crossover can produce; in *red* the parts that don't derive from the original copied individual.

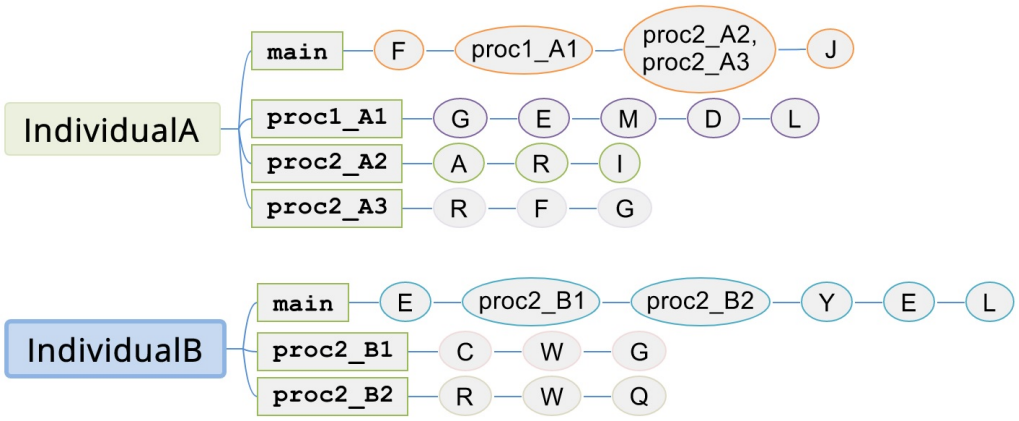


Figure 3.4: Example of parents selected to generate a new individual with Switch procedure crossover.

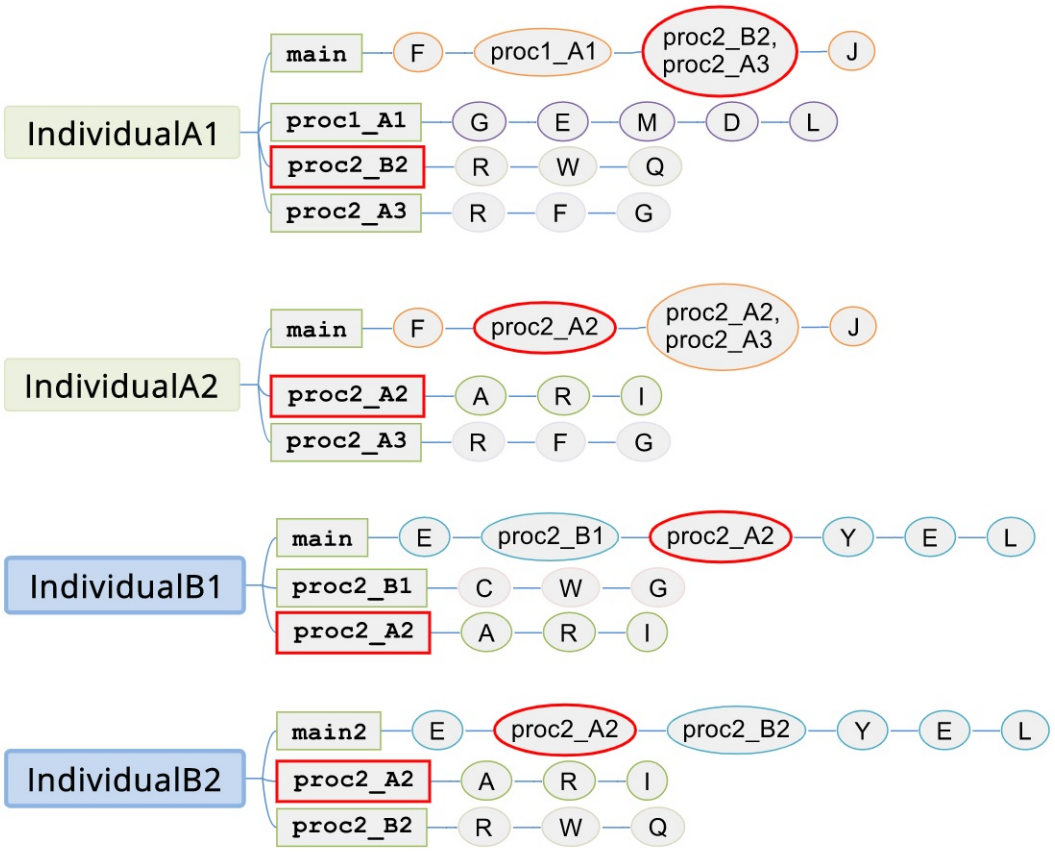


Figure 3.5: Example of new solutions produced by Switch procedure crossover.

MacroPool OneCut crossover (see Subsection 1.5.3) builds two lists of MacroPools in *parentA* and *parentB* belonging to common sections, chooses one element for each list and chooses one node (called *cut_node*). *parentA* and *parentB* are cloned and subsequently modified (in *individualC*, *individualD*); each individual will have the chosen MacroPool with an half copied from the other individual. Examples:

- *parentA* has a MacroPool with nodes: [A, B, C, D, E];
- *parentB* has a MacroPool with nodes: [F, G, H, I, L];
- the chosen *cut_node* are B and G;
- *individualC* will have a MacroPool with nodes: [A, B, H, I, L];
- *individualD* will have a MacroPool with nodes: [F, G, C, D, E].

If the chosen MacroPools has a different number of nodes, then two cut point will be chosen:

- *parentA* has a MacroPool with nodes: [A, B, C, D, E, F, G];
- *parentB* has a MacroPool with nodes: [H, I, L];
- the chosen *cut_node* are F and H;
- will have a MacroPool with nodes: [A, B, C, D, E, F, I, L];
- *individualD* will have a MacroPool with nodes: [H, G];

In Figure 3.6 you can see two possible parents individuals that are used to generate a new solution, the green area underlines the nodes in a MacroPool, the red "|" indicates the point chosen for the cut. In Figure 3.7 there is an example of what this crossover can produce; the circles with *red* borders are the nodes that have been copied from the other individual.

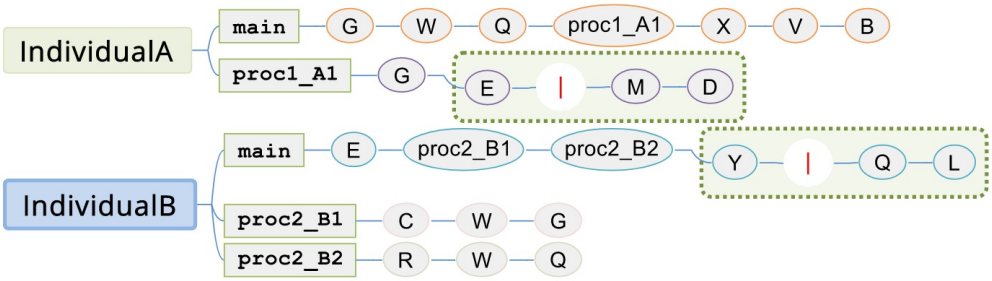


Figure 3.6: Example of parents selected to generate a new individual with MacroPool OneCut crossover.

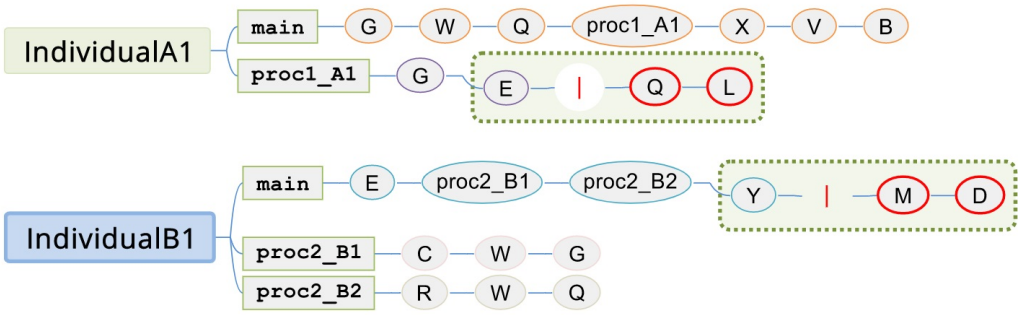


Figure 3.7: Example of new solutions produced by MacroPool OneCut crossover.

MacroPool Uniform crossover: builds two lists of MacroPools in *parentA* and *parentB* belonging to common sections, chooses one element for each list. *parentA* and *parentB* are cloned and subsequently modified (in *individualC* , *individualD*); each individual will have the chosen MacroPool swapped with that of the other. Examples:

- *parentA* has a MacroPool with nodes: [A, B, C, D, E];
- *parentB* has a MacroPool with nodes: [F, G, H, I, L];
- *individualC* will have a MacroPool with nodes: [F, G, C, D, E];
- *individualD* will have a MacroPool with nodes: [A, B, H, I, L].

If the chosen MacroPools has a different number of nodes:

- *parentA* has a MacroPool with nodes: [A, B, C, D, E, F, G];
- *parentB* has a MacroPool with nodes: [H, I, L];
- *individualC* will have a MacroPool with nodes: [H, I, L];
- *individualD* will have a MacroPool with nodes: [A, B, C, D, E, F, G].

In Figure 3.8 you can see two possible parents individuals that are used to generate a new solution, the green area underlines the nodes in a MacroPool. In Figure 3.9 there is an example of what this crossover can produce; the circles with *red* borders are the nodes that have been copied from the other individual.

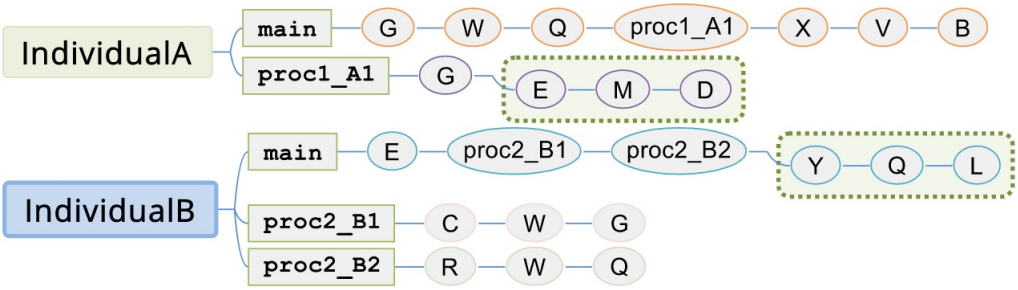


Figure 3.8: Example of parents selected to generate a new individual with MacroPool Uniform crossover.

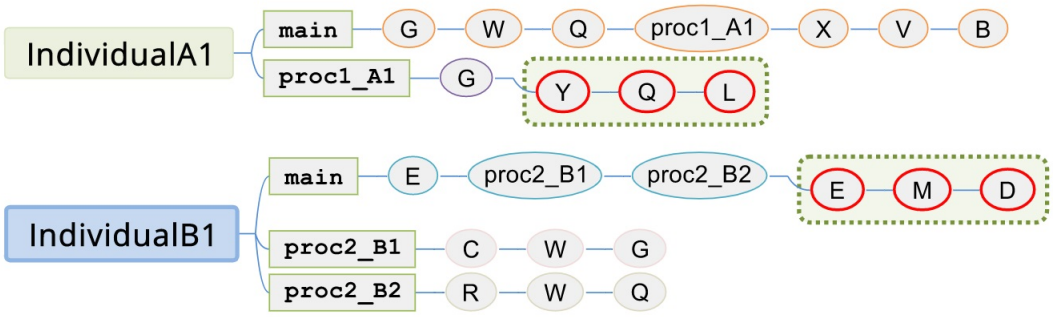


Figure 3.9: Example of new solutions produced by MacroPool Uniform crossover.

3.4 Fitness and evaluation

The library allows the use two main types of fitnesses (FitnessTuple, FitnessTupleMultiobj), both inherit from the `microgp.fitness.Base` class.

`microgp.fitness.Base` is the base class for handling Tuple-like fitnesses from scripts or functions. The different selection schemes are implemented simply by overriding the $>$ (greater than) operator.

Please note that, according to Spencer's 'Survival of the Fittest'[24], the bigger the better. Thus we are **maximizing** the fitness and not minimizing a mathematical function.

Fitness Tuple is the fitness used for single objective purposes (see Figure 3.10). **Fitness Tuple Multi-objective** is the fitness used for multiple objective purposes (see Figure 3.11).

3.4.1 Order by fitness logic

A method named `order_by_fitness(individuals_pool)` is used to get a list of individuals ordered by their correspondent fitness. This function calls the sort of the fitness, following the object-oriented paradigm. The sort method of *FitnessTuple* objects is a simple Python sort with reverse order, while the *FitnessTupleMultiobj* object sorting method uses the Pareto frontier[22] to determine which individuals dominate the others.

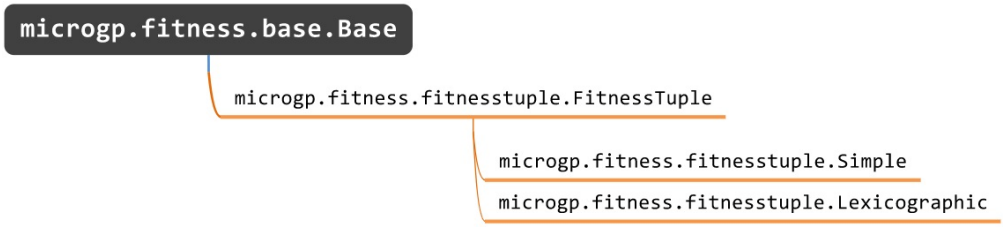


Figure 3.10: Schema of Fitness Tuple class.

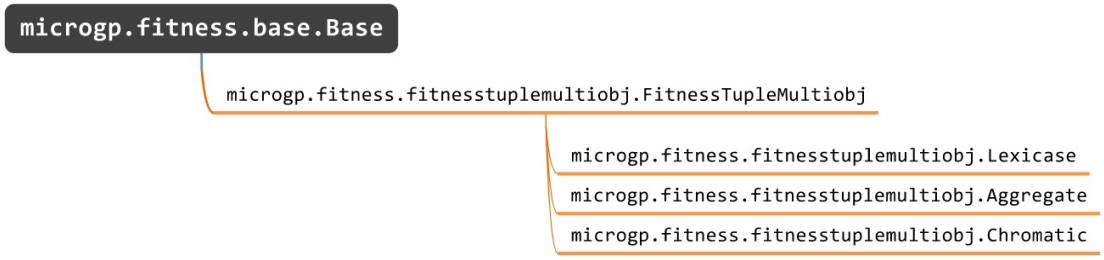


Figure 3.11: Schema of Fitness Tuple Multi-objective class.

3.4.2 Evaluator

As previously explained, the *evaluator* is declared during the construction of *Constraints*. The method `microgp.fitness.evaluator.make_evaluator(script, fitness_type, num_elements)` can get a script that is a callable method or a string indicating the path of the file that contains the script. During the declaration of the evaluator you can specify the type of fitness passing the class type. The last parameter is the number of relevant element in the script output; the remaining are considered "comment". If `num_elements=None` then the all output is part of the fitness.

Here is an example of evaluator that runs a script from the bash:

```
1 if sys.platform != "win32":
2     script = "./evaluator.sh"
3 else:
4     script = "evaluator.bat"
5 library.evaluator = ugp.fitness.make_evaluator(evaluator=script, fitness_type=ugp.
      fitness.Lexicographic)
```

Listing 3.24: Declaration of an evaluator running a script stored in a file.

3.5 Evolution process

The evolution process is managed by `microgp.darwin.Darwin` class through the `do_generation()` method that is called till one or more stopping conditions returns `True`. In the first release of MicroGP stopping condition have not yet been implemented, therefore a *while-loop* limits to 10 the number of generations.

A list of operators of length *lambda* are selected and used to generate an offspring that will be filtered and joined to the population. The population is also shrunk in order to keep only *mu* individuals in it.

You can see the pseudocode of `microgp.darwin.Darwin.do_generation()` here 5.

Algorithm 5 Do generation

```
1: procedure DO_GENERATION
2:     if population is empty then:
3:         selected_operators ← get from operators a GenOperator with max arity = 0
4:     else
5:         selected_operators ← get from operators a GenOperator with min arity = 1
6:     for operator in operators do
7:         if arity of operator > 0 then
8:             original_individuals ← select individuals from population with selection pressure = tau
9:         temporary_offspring ← list of new individuals generated by calling operator
10:        final_offspring ← filter not valid individuals
11:        update operator statistics based on the number of valid individuals produces
12:        whole_offspring = whole_offspring + final_offspring
13:    population ← population + whole_offspring
14:    filter population by age
15:    remove clones from population
16:    sort and keep only mu individuals in population
17:    update archive
18:    grow old each individual in population
```

3.5.1 Parent selection

The selection of individuals passed to the variation operators is handled by the `microgp.population.Population` object through the method `Population.select(tau)`. It exploits the tournament selection[3], which runs several tournaments among a few individuals and returns the best one based on their fitness score.

tau is a `microgp.darwin.Darwin` parameter that specifies the **selection pressure**. The higher it is, the higher is the number of tournament participants, thus weak individuals are less likely to be selected.

3.5.2 Offspring selection

Several filters are applied to the offspring before join it with the population:

- `None` and *not valid* individuals are removed;
- individuals with age greater than the *maximum age* specified in `Darwin` class are removed;
- individuals with the same phenotype (in the future versions).

This filtered resulting offspring is added to the current population, ordered by fitness and only the best *mu* individuals are kept inside the population.

Chapter 4

Experiments

4.1 Introduction

MicroGPv4 comes with three experiments that can be also used as examples of the operating logic. The experiments are solutions to the OneMax problem already discussed in Subsection 3.0.1.

4.2 OneMax problem - Base

As any library running Genetic Algorithms, MicroGPv4 can be tested for solving the basic OneMax problem. Two alternatives can be used: the individual is composed by 8 categorical parameters or 1 bitstring parameter.

4.2.1 OneMax Base version 1

With reference to Listing 4.1, in version one an individual is composed by a `word_section` (line 29) which contains a single macro (`word_macro` line 27) with a parameter (*bit*) of type `microgp.parameter.bitstring.Bitstring` of length 8 bits (line 25). The *main* section contains a simple prologue. The evaluator in both versions is a Python method (`my_script` line 37) returning an `int` value that is the sum of '1' in the individual's phenotype.

Here is the code:

```
1  import argparse
2  import sys
3
4  import microgp as ugp
5  from microgp.utils import logging
6
7  if __name__ == "__main__":
8      ugp.banner()
9      parser = argparse.ArgumentParser()
10     parser.add_argument("-v", "--verbose", action="count", default=0, help="
       increase log verbosity")
11     parser.add_argument("-d", "--debug", action="store_const", dest="verbose",
       const=2,
12                         help="log debug messages (same as -vv)")
13     args = parser.parse_args()
14     if args.verbose == 0:
15         ugp.logging.DefaultLogger.setLevel(level=ugp.logging.INFO)
16     elif args.verbose == 1:
17         ugp.logging.DefaultLogger.setLevel(level=ugp.logging.VERBOSE)
18     elif args.verbose > 1:
19         ugp.logging.DefaultLogger.setLevel(level=ugp.logging.DEBUG)
20         ugp.logging.debug("Verbose level set to DEBUG")
21     ugp.logging.cpu_info("Program started")
22
```

```
23
24      # Define a parameter of type ugp.parameter.Bitstring and length = 8
25      word8 = ugp.make_parameter(ugp.parameter.Bitstring, len_=8)
26      # Define a macro that contains a parameter of type ugp.parameter.Bitstring
27      word_macro = ugp.Macro("{word8}", {'word8': word8})
28      # Create a section containing a macro
29      word_section = ugp.make_section(word_macro, size=(1, 1), name='word_sec')
30
31      # Create a constraints library
32      library = ugp.Constraints()
33      # Define the sections in the library
34      library['main'] = ["Bitstring:", word_section]
35
36      # Define the evaluator method and the fitness type
37      def my_script(data: str):
38          count = data.count('1')
39          return list(str(count))
40
41      library.evaluator = ugp.fitness.make_evaluator(evaluator=my_script,
        fitness_type=ugp.fitness.Lexicographic)
42
43      # Create a list of operators with their aritiy
44      operators = ugp.Operators()
45      # Add initialization operators
46      operators += ugp.GenOperator(ugp.create_random_individual, 0)
47      # Add mutation operators
48      operators += ugp.GenOperator(ugp.hierarchical_mutation, 1)
49      operators += ugp.GenOperator(ugp.flat_mutation, 1)
50
51      # Create the object that will manage the evolution
52      mu = 10
53      nu = 20
54      sigma = 0.7
55      lambda_ = 7
56      max_age = 10
57
58      darwin = ugp.Darwin(
59          constraints=library,
60          operators=operators,
61          mu=mu,
62          nu=nu,
63          lambda_=lambda_,
64          sigma=sigma,
65          max_age=max_age,
66      )
67
68      # Evolve and print individuals in population
69      darwin.evolve()
70      logging.bare("This is the final population:")
71      for individual in darwin.population:
72          msg = f"Solution {str(individual.id)} "
73          ugp.print_individual(individual, msg=msg, plot=True)
74          ugp.logging.bare(f"Fitness: {individual.fitness}")
75          ugp.logging.bare("")
76
77      # Print best individuals
78      ugp.print_individual(darwin.archive.individuals, msg="These are the best ever
        individuals:", plot=True)
79
80      ugp.logging.cpu_info("Program completed")
81      sys.exit(0)
```

Listing 4.1: OneMax Base version 1 - Source code.

As you can see in Listing 4.1 the only useful variation operators are the Hierarchical mutation and Flat mutation (lines 48 and 49). No crossover operators are needed because there is only one node that contains mutable parameters.

In Figure 4.1 is shown the plof of a solution. It has two nodes: prologue and a macro containing a *bitstring parameter*.

Here is a possible log showing the results (Listing 4.2):

```
This is the final population:
Solution 49
Bitstring:
01011111
Fitness: Lexicographic(6)

Solution 71
Bitstring:
01110111
Fitness: Lexicographic(6)

Solution 65
Bitstring:
10111111
Fitness: Lexicographic(7)

Solution 41
Bitstring:
11111110
Fitness: Lexicographic(7)

Solution 60
Bitstring:
11110111
Fitness: Lexicographic(7)

Solution 59
Bitstring:
11101111
Fitness: Lexicographic(7)

Solution 40
Bitstring:
11011101
Fitness: Lexicographic(6)

Solution 56
Bitstring:
10111101
Fitness: Lexicographic(6)

Solution 64
Bitstring:
11011011
Fitness: Lexicographic(6)

Solution 39
Bitstring:
01111111
Fitness: Lexicographic(7)

These are the best ever individuals:
Bitstring:
01111111
Bitstring:
10111111
Bitstring:
11111110
Bitstring:
11110111
Bitstring:
11101111
```

Listing 4.2: OneMax Base version 1 - Example log.

4.2.2 OneMax Base version 2

With reference to Listing 4.3, in version two an individual is composed by a `word_section` (line 35) which contains exactly 8 macros (`word_macro` line 33) with a parameter of type

Figure 4.1: Plot of an individual generated for OneMax Base version 1 problem.

`microgp.parameter.categorical.Categorical` (line 31) that can assume as value: *1* or *0*. The main section contains a simple prologue and epilogue. The evaluator is the same already seen in version 1.

Here is the code:

```
1  import argparse
2  import sys
3
4  import microgp as ugp
5  from microgp.utils import logging
6
7  if __name__ == "__main__":
8      ugp.banner()
9
10     parser = argparse.ArgumentParser()
11     parser.add_argument("-v", "--verbose", action="count", default=0,
12                         help="increase log verbosity")
13     parser.add_argument("-d", "--debug", action="store_const", dest="verbose",
14                         const=2, help="log debug messages (same as -vv)")
15     args = parser.parse_args()
16
17     if args.verbose == 0:
18         ugp.logging.DefaultLogger.setLevel(level=ugp.logging.INFO)
19     elif args.verbose == 1:
20         ugp.logging.DefaultLogger.setLevel(level=ugp.logging.VERBOSE)
21     elif args.verbose > 1:
22         ugp.logging.DefaultLogger.setLevel(level=ugp.logging.DEBUG)
23         ugp.logging.debug("Verbose level set to DEBUG")
24
25     ugp.logging.cpu_info("Program started")
26
27     # Delete old solutions
28     ugp.delete_solutions()
29
30     # Define a parameter of type ugp.parameter.Categorical that can take two values:
         0 or 1
31     bit = ugp.make_parameter(ugp.parameter.Categorical, alternatives=[0, 1])
32     # Define a macro that contains a parameter of type ugp.parameter.Categorical
33     word_macro = ugp.Macro("{bit}", {'bit': bit})
34     # Create a section containing 8 macros
35     word_section = ugp.make_section(word_macro, size=(8, 8), name='word_sec')
36
37     # Create a constraints library
38     library = ugp.Constraints()
39     library['main'] = ["Here is the bitstring", word_section, ""]
40
41     # Define the evaluator and the fitness type
42     def my_script(filename: str):
43         with open(filename) as file:
44             data = file.read()
45             count = data.count('1')
46         return list(str(count))
```

```
47
48      library.evaluator = ugp.fitness.make_evaluator(script=my_script, fitness_type=
            ugp.fitness.Lexicographic)
49
50      # Create a list of operators with their arities
51      operators = ugp.Operators()
52      # Add initialization operators
53      operators += ugp.GenOperator(ugp.create_random_individual, 0)
54      # Add mutation operators
55      operators += ugp.GenOperator(ugp.hierarchical_mutation, 1)
56      operators += ugp.GenOperator(ugp.flat_mutation, 1)
57      # Add crossover operators
58      operators += ugp.GenOperator(ugp.macro_pool_one_cut_point_crossover, 2)
59      operators += ugp.GenOperator(ugp.macro_pool_uniform_crossover, 2)
60
61      # Create the object that will manage the evolution
62      mu = 10
63      nu = 20
64      sigma = 0.7
65      lambda_ = 7
66      max_age = 10
67
68      for _ in range(1):
69          darwin = ugp.Darwin(
70              constraints=library,
71              operators=operators,
72              mu=mu,
73              nu=nu,
74              lambda_=lambda_,
75              sigma=sigma,
76              max_age=max_age,
77          )
78
79          # Evolve
80          darwin.evolve()
81          logging.bare("This is the final population:")
82          for individual in darwin.population:
83              ugp.print_individual(individual, plot=True)
84              ugp.logging.bare(individual.fitness)
85              ugp.logging.bare("")
86
87          # Print best individuals
88          logging.bare("These are the best ever individuals:")
89          ugp.print_individual(darwin.archive)
90
91          ugp.delete_solutions()
92
93      ugp.logging.cpu_info("Program completed")
94      sys.exit(0)
```

Listing 4.3: OneMax Base version 2 - Source code.

As you can see in Listing 4.3 the only useful variation operators are the Hierarchical mutation, Flat mutation, MacroPool OneCut crossover and MacroPool Uniform crossover. The crossover operators applies variations to the graph structure because each node contains a macro with one parameter.

Here is a possible log (Listing 4.4):

```
1       This is the final population:
2       Solution 21
3       Bitstring:
4       0
5       1
6       1
7       1
8       0
9       1
10      1
11      1
12      Fitness: Lexicographic(6)
```

```
13
14    ...
15
16      Solution 44
17      Bitstring:
18      1
19      1
20      1
21      1
22      1
23      1
24      1
25      1
26      Fitness: Lexicographic(8)
27
28      These are the best ever individuals:
29      Bitstring:
30      1
31      1
32      1
33      1
34      1
35      1
36      1
37      1
```

Listing 4.4: OneMax Base version 2 - Example log

In Figure 4.2 is represented the graph of a solution. There are 9 nodes: the first is the prologue and the other nodes contains macros with a `Categorical` parameter.

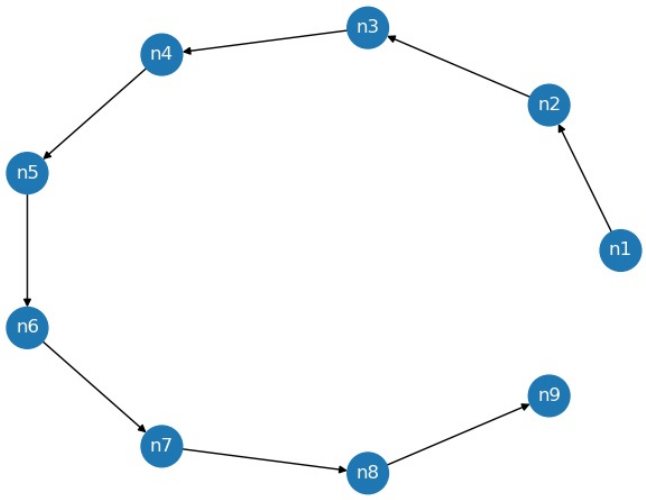


Figure 4.2: Plot of an individual generated for OneMax Base version 2 problem

4.3 OneMax problem - Assembly

The following code (Listing 4.7) produces assembly code that can be run on x86 processors. The goal is to generate an assembly script that writes in `eax` a binary number with as much as *ones* (`1`) as possible.

The evaluator is a `.bat` (Listing 4.5) file that generates an `.exe` file in charge of call a script that counts the number of ones in the returned integer value.

```
1       @echo off
```

```
2
3       rem comment
4
5       del a.exe
6       gcc main.o %1
7
8       if exist a.exe (
9           .\a.exe
10      ) else (
11          echo -1
12      )
```

Listing 4.5: OneMax Assembly evaluator (eval.bat)

A possible solution (**phenotype**) could be the one in the Listing 4.6

```
        .file   "solution.c"
        .text
        .globl  _darwin
        .def    _darwin;        .scl    2;      .type   32;     .
           endef
_darwin:
LFB17:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5

        movl    $-31312, %eax
        movl    $25598, %ebx
        movl    $-24861, %ecx
        movl    $-19236, %edx

        sub %ebx, %edx
        shl $216, %ecx
        jnz n9
        jnc n23
        cmp %ecx, %ecx
        shl $207, %edx
n9:
        jc n22
        xor %ebx, %eax
        jnz n28
        xor %eax, %ebx
        sub %edx, %edx
        jno n15
n15:
        jz n28
        shr $229, %ebx
        sub %ebx, %eax
        jc n23
        cmp %edx, %ebx
        and %ebx, %ecx
        shl $186, %eax
```

```
n22:
    cmp %eax, %edx
n23:
    jnz n29
    jz n29
    jmp n28
    jc n29
    shl $143, %ecx
n28:
    or %ebx, %eax
n29:
    movl    %eax, -4(%ebp)
    movl    -4(%ebp), %eax
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
LFE17:
   .ident  "GCC: (MinGW.org GCC-8.2.0-5) 8.2.0"

Fitness score: Lexicographic(29)
```

Listing 4.6: OneMax Assembly possible script solution (phenotype).

The correspondent graph plot is:

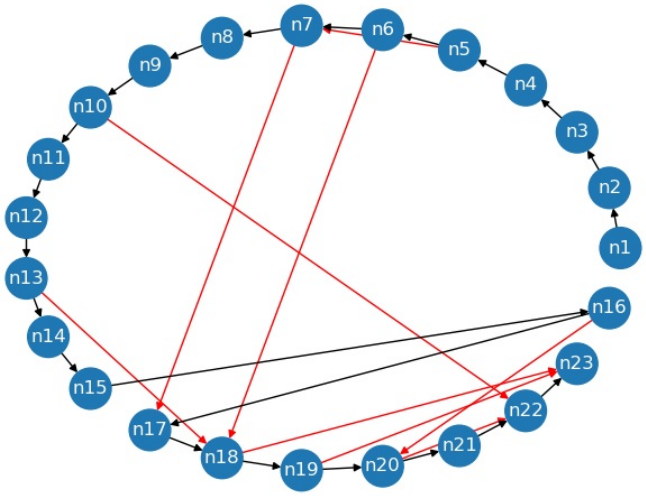


Figure 4.3: Plot of an individual generated for OneMax Assembly problem. Black edges are *next* edges and the red ones are `LocalReferences` (jump).

```
1  import argparse
2  import sys
3
4  import microgp as ugp
5  from microgp.utils import logging
6
7  if __name__ == "__main__":
8      ugp.banner()
```

```
    parser = argparse.ArgumentParser()
    parser.add_argument("-v", "--verbose", action="count", default=0, help="
    increase log verbosity")
    parser.add_argument("-d", "--debug", action="store_const", dest="verbose",
    const=2,
                        help="log debug messages (same as -vv)")
    args = parser.parse_args()
    if args.verbose == 0:
        ugp.logging.DefaultLogger.setLevel(level=ugp.logging.INFO)
    elif args.verbose == 1:
        ugp.logging.DefaultLogger.setLevel(level=ugp.logging.VERBOSE)
    elif args.verbose > 1:
        ugp.logging.DefaultLogger.setLevel(level=ugp.logging.DEBUG)
        ugp.logging.debug("Verbose level set to DEBUG")
    ugp.logging.cpu_info("Program started")

    # Define parameters
    reg_alternatives = ['%eax', '%ebx', '%ecx', '%edx']
    reg_param = ugp.make_parameter(ugp.parameter.Categorical, alternatives=
    reg_alternatives)
    instr_alternatives = ['add', 'sub', 'and', 'or', 'xor', 'cmp']
    instr_param = ugp.make_parameter(ugp.parameter.Categorical, alternatives=
    instr_alternatives)
    shift_alternatives = ['shr', 'shl']
    shift_param = ugp.make_parameter(ugp.parameter.Categorical, alternatives=
    shift_alternatives)
    jmp_alternatives = ['ja', 'jz', 'jnz', 'je', 'jne', 'jc', 'jnc', 'jo', 'jno', '
    jmp']
    jmp_instructions = ugp.make_parameter(ugp.parameter.Categorical, alternatives=
    jmp_alternatives)
    integer = ugp.make_parameter(ugp.parameter.Integer, min=-32768, max=32767)
    int8 = ugp.make_parameter(ugp.parameter.Integer, min=0, max=256)
    jmp_target = ugp.make_parameter(ugp.parameter.LocalReference,
                                    allow_self=False,
                                    allow_forward=True,
                                    allow_backward=False,
                                    frames_up=0)

    # Define the macros
    jmp1 = ugp.Macro("    {jmp_instr} {jmp_ref}", {'jmp_instr': jmp_instructions, '
    jmp_ref': jmp_target})
    instr_op_macro = ugp.Macro("    {instr} {regS}, {regD}",{'instr': instr_param,
    'regS': reg_param, 'regD': reg_param})
    shift_op_macro = ugp.Macro("    {shift} ${int8}, {regD}", {'shift': shift_param
    , 'int8': int8, 'regD': reg_param})
    branch_macro = ugp.Macro("{branch} {jmp}", {'branch': jmp_instructions, 'jmp':
    jmp_target})
    prologue_macro = ugp.Macro('    .file   "solution.c"\n' +
                               '    .text\n' +
                               '    .globl  _darwin\n' +
                               '    .def    _darwin;        .scl    2;      .type
     32;     .endef\n' +
                               '_darwin:\n' +
                               'LFB17:\n' +
                               '    .cfi_startproc\n' +
                               '    pushl   %ebp\n' +
                               '    .cfi_def_cfa_offset 8\n' +
                               '    .cfi_offset 5, -8\n' +
                               '    movl    %esp, %ebp\n' +
                               '    .cfi_def_cfa_register 5\n')
    init_macro = ugp.Macro("    movl ${int_a}, %eax\n" +
                           "    movl ${int_b}, %ebx\n" +
                           "    movl ${int_c}, %ecx\n" +
                           "    movl ${int_d}, %edx\n",
                           {'int_a': integer, 'int_b': integer, 'int_c': integer, '
    int_d': integer})
    epilogue_macro = ugp.Macro(
        '    movl    %eax, -4(%ebp)\n' +
        '    movl    -4(%ebp), %eax\n' +
        '    leave\n' +
        '    .cfi_restore 5\n' +
        '    .cfi_def_cfa 4, 4\n' +
        '    ret\n' +
```

```
        '    .cfi_endproc\n' +
        'LFE17:\n' +
        '    .ident  "GCC: (MinGW.org GCC-8.2.0-5) 8.2.0"\n')

    # Define section
    sec1 = ugp.make_section({jmp1, instr_op_macro, shift_op_macro}, size=(1, 50))

    # Create a constraints library
    library = ugp.Constraints(file_name="solution{id}.s")
    library['main'] = [prologue_macro, init_macro, sec1, epilogue_macro]

    # Define the evaluator script and the fitness type
    if sys.platform != "win32":
        exit(-1)
    else:
        script = "eval.bat"
    library.evaluator = ugp.fitness.make_evaluator(evaluator=script, fitness_type=
    ugp.fitness.Lexicographic)

    # Create a list of operators with their arity
    operators = ugp.Operators()
    # Add initialization operators
    operators += ugp.GenOperator(ugp.create_random_individual, 0)
    # Add mutation operators
    operators += ugp.GenOperator(ugp.hierarchical_mutation, 1)
    operators += ugp.GenOperator(ugp.flat_mutation, 1)
    operators += ugp.GenOperator(ugp.add_node_mutation, 1)
    operators += ugp.GenOperator(ugp.remove_node_mutation, 1)
    # Add crossover operators
    operators += ugp.GenOperator(ugp.macro_pool_one_cut_point_crossover, 2)
    operators += ugp.GenOperator(ugp.macro_pool_uniform_crossover, 2)

    # Create the object that will manage the evolution
    mu = 10
    nu = 20
    sigma = 0.7
    lambda_ = 7
    max_age = 10

    darwin = ugp.Darwin(
        constraints=library,
        operators=operators,
        mu=mu,
        nu=nu,
        lambda_=lambda_,
        sigma=sigma,
        max_age=max_age,
    )

    # Evolve
    darwin.evolve()

    # Print best individuals
    logging.bare("These are the best ever individuals:")
    best_individuals = darwin.archive.individuals
    ugp.print_individual(best_individuals, plot=True, score=True)

    ugp.logging.cpu_info("Program completed")
    sys.exit(0)
```

Listing 4.7: OneMax Assembly - Source code

The script syntax has been built to work with Windows 10, 64-bit, for *GCC: (MinGW.org GCC-8.2.0-5) 8.2.0*[1].

[1]MinGW: http://www.mingw.org/

Chapter 5

Conclusions and future works

The re-design and re-implementation of previous MicroGP versions in Python language let to a huge improvement. New users can now be reached through easy and flexible high level methods and a completely new logic. A basic user can rapidly build its own project downloading MicroGPv4 package simply writing `pip install microgp` in its command line as shown in Figure 5.1.

```
M:\LuBa\Documenti\GitHub\microgp4>pip install microgp
Collecting microgp
  Downloading microgp-4%211.0a4.tar.gz (49 kB)
     |████████████████████████████████| 49 kB 767 kB/s
Installing collected packages: pyreadline, humanfriendly, coloredlogs, psutil, microgp
  Attempting uninstall: psutil
    Found existing installation: psutil 5.6.4
    Uninstalling psutil-5.6.4:
      Successfully uninstalled psutil-5.6.4
Successfully installed coloredlogs-14.0 humanfriendly-8.1 microgp-4!1.0a4 psutil-5.7.0 pyreadline-2.1
```

Figure 5.1: Installation of MicroGPv4 package through PIP

A documentation with examples and guides is available online[1] (Figure 5.2).

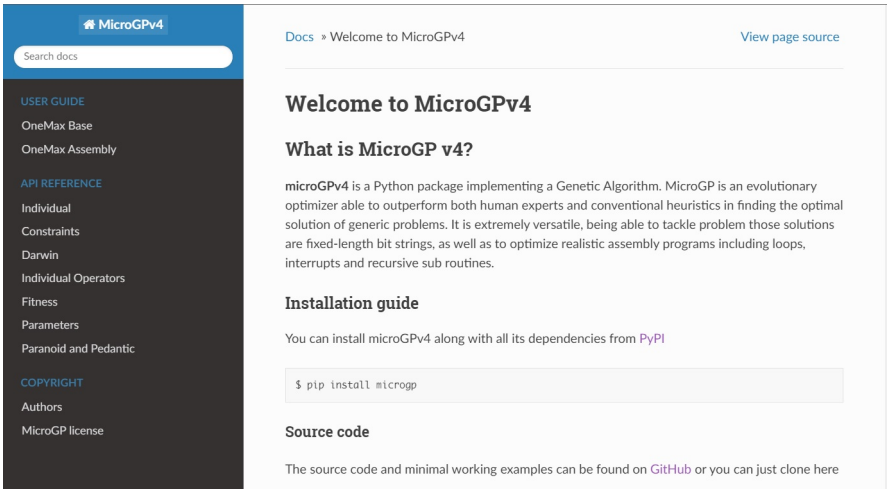


Figure 5.2: Online documentation home page

[1] Welcome to MicroGPv4: https://microgp4.readthedocs.io/

5.1 Future works

The released version is a *pre-alpha*, meaning that the tool is fully functional but largely incomplete in terms of functionalities will be available in the next updates.

5.1.1 Statistics for genetic operators selection

The Darwin class, as first step of a generation, uses Operators class to retrieve a bunch of genetic operators. The operators selection process simply returns a set randomly chosen methods depending on the *arity* range passed by Darwin object. The arity indicates the number of individuals on which the genetic operator works on.

For example, during the initialization phase only the operators with `arity=0` are selected and then run, allowing the creation of a bunch of new random individuals (see Listing 5.1.

```
1 if len(self._population) == 0:
2     # Example: generate 10.000 (nu) individuals and then keep only the best 10 (mu)
       individuals
3     selected_operators = self._operators.select(max_arity=0, k=self._nu)
```

Listing 5.1: First generation genetic operators selection

The proposed improvement consists in tracking the efficiency of the operators and selecting with higher probability the best ones. The quality of a genetic operator can be described as the percentage of successes (number of times in which it returned a valid individual over the times it has been called). The method `microgp.operators.Operators.select()` will implement a logic solving a typical multi-armed bandit problem[20].

5.1.2 Stopping conditions

By now, the only way for the evolution process to stop is if the number of generations have reached a fixed value (Listing 5.2. In the future releases the Darwin object will receive a bunch of stopping conditions[26] that will be verified at each generation. If one or more stopping conditions verify the evolution process is immediately terminated.

```
1 def evolve(self) -> None:
2     while self._generation < 20:
3         self.do_generation()
```

Listing 5.2: Current version of `microgp.darwin.Darwin.evolve()` method

5.1.3 2D and 3D categorical sorted parameter

The current **Categorical Sorted** parameter (`microgp.parameter.categorical.CategoricalSorted`) can be used for one dimensional ordered values. When a mutation of this parameter takes place depending on the strength of the *sigma* parameter some values have higher probability to be set. Usually, with a low *sigma* value, the alternatives closer to the current one are selected.

We think that this logic can be useful for 2D and 3D spatial problems in which the closeness is computed with more variables.

Bibliography

[1] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[2] Edward A Bender. *An introduction to mathematical modeling*. Courier Corporation, 2012.

[3] Tobias Blickle and Lothar Thiele. "A Mathematical Analysis of Tournament Selection." In: *ICGA*. Vol. 95. Citeseer. 1995, pp. 9–15.

[4] Justin Boyan, Dayne Freitag, and Thorsten Joachims. "A machine learning architecture for optimizing web search engines". In: *AAAI Workshop on Internet Based Information Systems*. 1996, pp. 1–8.

[5] Jürgen Branke et al. *Multiobjective optimization: Interactive and evolutionary approaches*. Vol. 5252. Springer Science & Business Media, 2008.

[6] Chihyung Derrick Cheng and Alexander Kosorukoff. "Interactive one-max problem allows to compare the performance of interactive and human-based genetic algorithms". In: *Genetic and evolutionary computation conference*. Springer. 2004, pp. 983–993.

[7] F Corno et al. "Efficient machine-code test-program induction". In: *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*. Vol. 2. IEEE. 2002, pp. 1486–1491.

[8] Fulvio Corno, Ernesto Sánchez, and Giovanni Squillero. "Evolving assembly programs: how games help microprocessor validation". In: *IEEE Transactions on Evolutionary Computation* 9.6 (2005), pp. 695–706.

[9] Kenneth De Jong. "Evolutionary computation: a unified approach". In: *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*. 2016, pp. 185–199.

[10] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*. Vol. 16. John Wiley & Sons, 2001.

[11] Sumeet Dua and Xian Du. *Data mining and machine learning in cybersecurity*. CRC press, 2016.

[12] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*. Vol. 53. Springer, 2003.

[13] Merrill M Flood. "The traveling-salesman problem". In: *Operations research* 4.1 (1956), pp. 61–75.

[14] James B Fraley and James Cannady. "The promise of machine learning in cybersecurity". In: *SoutheastCon 2017*. IEEE. 2017, pp. 1–6.

[15] Adam Geitgey. "Machine learning is fun". In: *Medium. Retrieved from https://medium.com/@ ageitgey/machine-learning-is-fun-80ea3ec3c471* (2014).

[16] F Guerriero et al. “A biobjective optimization model for routing in mobile ad hoc networks”. In: *Applied Mathematical Modelling* 33.3 (2009), pp. 1493–1512.

[17] Abdenour Hadid et al. “Face and eye detection for person authentication in mobile phones”. In: *2007 First ACM/IEEE International Conference on Distributed Smart Cameras*. IEEE. 2007, pp. 101–108.

[18] Dieter W Heermann. “Computer-simulation methods”. In: *Computer Simulation Methods in Theoretical Physics*. Springer, 1990, pp. 8–12.

[19] Francisco Jeronimo. “Mobile AI and the Future of Intelligent Devices”. In: *2017 White Paper IDC* (2017).

[20] Volodymyr Kuleshov and Doina Precup. “Algorithms for multi-armed bandit problems”. In: *arXiv preprint arXiv:1402.6028* (2014).

[21] W Lindsay et al. “Automatic test programs generation driven by internal performance counters”. In: *Fifth International Workshop on Microprocessor Test and Verification (MTV'04)*. IEEE. 2004, pp. 8–13.

[22] Patrick Ngatchou, Anahita Zarei, and A El-Sharkawi. “Pareto multi objective optimization”. In: *Proceedings of the 13th International Conference on, Intelligent Systems Application to Power Systems*. IEEE. 2005, pp. 84–91.

[23] Nils J Nilsson. *Principles of artificial intelligence*. Morgan Kaufmann, 2014.

[24] John Offer. “From ‘natural selection’to ‘survival of the fittest’: On the significance of Spencer's refashioning of Darwin in the 1860s”. In: *Journal of Classical Sociology* 14.2 (2014), pp. 156–177.

[25] Dov Ospovat. *The development of Darwin's theory: Natural history, natural theology, and natural selection, 1838-1859*. Cambridge University Press, 1995.

[26] Martín Safe et al. “On stopping criteria for genetic algorithms”. In: *Brazilian Symposium on Artificial Intelligence*. Springer. 2004, pp. 405–413.

[27] Ernesto Sanchez, Massimiliano Schillaci, and Giovanni Squillero. *Evolutionary Optimization: the μGP toolkit*. Springer Science & Business Media, 2011.

[28] Minoru Sasaki and Hiroyuki Shinnou. “Spam detection using text clustering”. In: *2005 International Conference on Cyberworlds (CW'05)*. IEEE. 2005, 4–pp.

[29] Eric O Scott and Sean Luke. “ECJ at 20: toward a general metaheuristics toolkit”. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2019, pp. 1391–1398.

[30] Giovanni Squillero. “MicroGP—an evolutionary assembly program generator”. In: *Genetic Programming and Evolvable Machines* 6.3 (2005), pp. 247–263.

[31] George Alexis Terry et al. *Systems and methods for using natural language instructions with an ai assistant associated with machine learning conversations*. US Patent App. 16/228,717. 2019.

[32] Matthias Thiel, Roland Schulz, and Peter Gmilkowsky. “Simulation-based production control in the semiconductor industry”. In: *1998 Winter Simulation Conference. Proceedings (Cat. No. 98CH36274)*. Vol. 2. IEEE. 1998, pp. 1029–1033.

[33] Aimo Törn and Antanas Žilinskas. *Global optimization*. Vol. 350. Springer, 1989.