Politecnico di Torino

Collegio di Ingegneria Informatica, Cinema e Meccatronica

Master Thesis

Design and development of a portable, general-purpose evolutionary optimizer

Advisors:
Squillero Giovanni
Tonda Alberto Paolo

Candidate:
Malfatto Nicolò

April 2018

"Nothing is created, nothing is destroyed, everything is transformed"
Antoine-Laurent de Lavoisier

Index

0. Summary

The thesis describes the design and the development of the core of a portable, general-purpose evolutionary optimizer. While the long-term goal of the project is the complete re-engineerization of the 10-year old toolkit called MicroGP[1], the activity of this thesis focused on the thorough analysis of the previous tool, the identification of its strengths and weakness; eventually followed by the definition of the new architecture, and the implementation of its key elements.

MicroGP is a generic tool that can optimize the solution of hard problems. It can be used to solve classical tasks, such as creation of test programs for pre- and post-silicon validation, design of Bayesian networks, creation of mathematical functions represented as trees, integer and combinatorial optimization and many more. But, as it is able to handle complex structures, the solution itself may be in the form of a program in an artificial language. To maximize applicability, the solution space is defined in external files, and the procedure to evaluate the candidate solution is provided by the user.

While the final goal is the same, the new version is quite different: the whole toolkit has been re-implemented from scratch in a new language, cleaning up and simplifying the design; most of the limitations have been removed, while usability and maintainability have been greatly improved.

In more details, the language switched from C++ to Python, the description of the search space switched from XML to JSON. The first change was made because more and more users are using it, Python allows to write complex code and data structure in easier way, it does not have to be compiled so the resulting program is portable on any operative system, thanks to all these features the community is very active, so there are lots of library. The latter was made because JSON has a lighter syntax that should be less confusing for someone that wants to simply use the program to find an optimal solution for a hard problem.

[1] https://github.com/squillero/microgp3

The development part was concentrated to the creation of the core of the MicroGP4. In the actual state the toolkit is an instruction randomizer, because the evolutionary motor is missing.

In particular the developed parts are:

- The set of classes, methods and functions that are responsible for reading information from the JSON. But reading the files is not the only thing done regarding the input phase, it was discussed and decided how data should encode in JSON format, in such a way to be clearer and less confusing for the user.
- The internal representation of an individual and all the data structures that serves to represent, manage and check the solution. The individual is represented as an extension of multidigraph, to check its formal correctness it has a method which executed a list of constraint functions that are dynamically injectables, but they must follow a given syntax. In fact, there is a meta description of how every element should be created; the existing classes and functions can be seen as example to use the various system.
- The dump of the individual in to a real solution. Because of it is not represented by a conventional graph, ad hoc version of the standard visit was developed to both implement the depth-first and breadth-first search.
- The basic operation, such as the individual cloning and mutation of its internal parameters.

This thesis has been developed under the joint supervision of Prof. Giovanni Squillero from Politecnico di Torino and Dr. Alberto Tonda from Institut National de la Recherche Agronomique.

1. Evolutionary Computation Introduction

Evolutionary Computation is a subfield of artificial intelligence and it consists in to optimize solution through iteration and recombination of solution and this behavior is inspired by biological evolution. It was born during the sixties guided by the idea of automating problem solving.

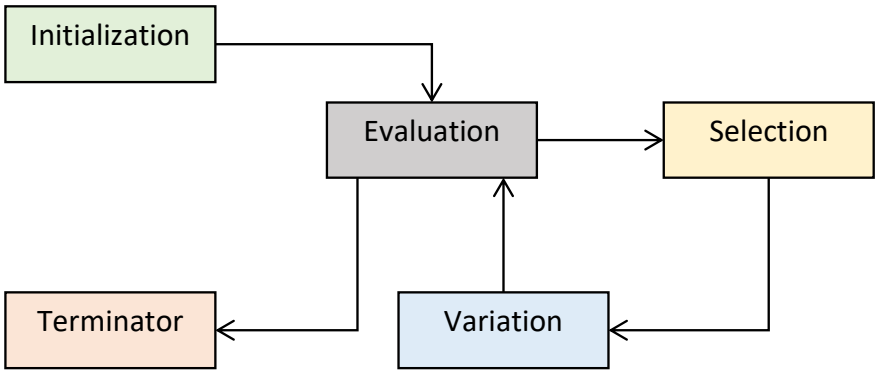

Figure 1 Evolutionary Algorithm logic scheme

1.1 Evolutionary Algorithms

Evolutionary Algorithms are inspired by biological evolution and it shares with it some of the keywords and concepts and under this name many algorithms are grouped.

Evolution is the biological theory that animals and plants have their origin in other types, and that the distinguishable differences are due to modifications in successive generations.

The process is not random, but it is based on random variations, some of those are rejected and others are conserved according to an objective evaluation, in fact only the

changes, that are beneficial to the individual, are transmitted into the next generations. This process had been called by Darwin “natural selection”.

Natural selection is an optimization process, because it generates always better individual to survive in the environment, using only random variation without the ability to design intelligent modifications. The same logic could apply in order to solve difficult problems in an efficient way.

As mentioned earlier the keywords of Evolutionary Algorithms mimic the terminology of the biology. In fact, a single candidate solution is called “individual” and the set of all individuals that exist in a certain time is called “population”.

Evolution proceeds through discrete steps called “generations”; in each of this step the population is first expanded, to mimic the process of breeding, then collapse mimicking the process of struggling for survival. Some Evolutionary Algorithms do not store a set of distinct individuals and evolution is represented through the variation of the statistical parameters that describe the population, this approach has the advantage to use less resources during the computational phases.

The ability of an individual to solve the target problem is called “fitness” and it is measured by the “fitness function”. Fitness value influences the probability of a solution to propagate its characteristic to the next generation. There are some approaches in which an individual dies in the golden age, others in which it survives in the population until replaced by a fitter one and others ones that is a combination of the previous two.

The word “genome” denotes the whole genetic material of the organism and the term “genes” refers to the functional unit of inheritance, or, operatively, the smallest fragment of the genome that may be modified in the evolution process.

Genes are positioned in the genome at specific positions called "loci", while the alternative genes that may occur at a given locus are called "alleles". The position of each gene inside a solution is really important, because it can influence the fitness value in a significant way.

In biology two fundamental concepts are: “genotype” and “phenotype”. The first one is all the genetic constitution of an organism, the second one is the set of observable properties that are produced by the interaction between the genotype and the environment. In several implementations the single numerical value representing the fitness of an individual is sometimes assimilated to its phenotype.

Evolutionary Algorithms generate offspring implementing sexual and asexual reproduction. The first one is called “recombination”, it involves two or more participants, it implies the possibility for the offspring to inherit different characteristics from different parents and when recombination is achieved through an exchange of genetic material between the parents, it is often called crossover.

Asexual reproduction is also called “replication”, when individual is copied without any changes, while, it takes the name of “mutation”, if the copy is not exactly as the original one, but some genes are changed.

All the operators used in reproduction are called “evolutionary operators”, or “genetic operators” because they act at the genotypical level. Almost no Evolutionary Algorithm takes gender into consideration, because individuals do not have distinct reproductive roles.

Evolutionary Algorithms provides an effective methodology for trying random modifications, where no preconceived idea about the optimal solution is required. They are more robust than pure hill climbing, because of they are based on a population and also due to the fact that both small and large modifications are possible, but with different probabilities. No human intervention is required when Evolutionary Algorithms run and they are quite simple to set up. Finally, it’s easy to trade-off between computational resources and quality of the results.

But unfortunately, there are some problems which can dramatically decrease the performance of these algorithm family. The fitness function must be able to create different values for each different solution, because small variation must trigger different survival in the population and, due to the fact that fitness is a synthetic measure, it is fundamental to carefully consider which information has to be kept and which has to be removed.

For example, if an Evolutionary Algorithm is used to implement and “automatic test-pattern generator”, the fitness cannot only be the attained “fault coverage”. If this value is the same in two individuals, for the algorithm they are equivalent; but if the first is able to detect also some random-resistance, hard-to-test fault and the second is not able to detect these aspects, only the characteristics of the first one must be inherited by the next generation.

Another key aspect of the design and implementation of an Evolutionary Algorithm is the choice of the representation of the individual, because it must encode some structural information about the desired solution. Having too much information inside the individual may penalize the discover of the optimal solution, on the contrary having to few information may increase the dimension of the search-space, slowing down the evolutionary process. The individual representation influences the genetic operators and their behavior, because they work with the characteristics of the solution.

The real main problem of Evolutionary Algorithms is the “premature convergence”: after several generations, all individuals, or most of them, are very similar and the population converges into a single point in the search-space, as if this were not enough, all the genetic operator became quite ineffective causing the almost stop of the evolutionary process. In such a condition the program behaves like an overloaded, inefficient random-mutation hillclimber. Unfortunately, there is not a universal solution for this problem, but for each implementation an appropriate solution must be applied.

1.2 Industrial Application

The first use of Evolutionary Algorithms in an industrial environment dates back to the beginning of 1980, and these techniques were initially meanly used to optimize numeric efficient.

In the 1990s, Evolutionary Algorithms eventually gathered some recognition in the CAD community, because the methods used at that times were unable to scale with the technologies and to handle complex circuits and Evolutionary Algorithms were seen as promising alternative.

In fact, researchers proposed methodologies based on Evolutionary Algorithms to solve several NP-hard problems, such as placement, floorplanning and routing, however they require more complex encoding of the individuals and the design of ad-hoc genetic operators.

But despite these aspects, they found a fertile ground into optimize and mapping FPGA, “logic synthesis” and decision-diagrams optimization, like BDD, OBDD, KFDD and many others. And they started also to appear in some key CAD conferences. Thanks to the initial success of using this new approach, they have been used more often and to solve more and more problems.

Evolutionary Algorithms were also used for simulation-based design validation, however design the fitness function involves sever problems. This is caused by the fact that a bug is either caught or silent, without intermediate values, while evolution requires the majority of the individuals in the population to have different fitness values. To solve this issue, the fitness functions can be based on the coverage-metric figures obtained when simulating the individuals.

In later works, Evolutionary Algorithms are used to design more complex elements, like non-standard built-in self-test structures.

1.3 MicroGP General description

MicroGP is an optimizer that can be used to find eventually the optimal solution to a hard problem. It can be theoretically solved any hard problems thanks to the fact that language description (in which the solutions must be represented), evolutionary motor and solutions evaluator are three separate blocks.

In terms of efficiency and solution quality it is capable to overcome human expert and conventional heuristic.

It starts generating random solution for the given problem, then it refines them in an iterative way. The core of the program is its heuristic algorithm, which is used, with some internal information, to efficiently explore the search space and eventually find the optimal solution.

The original purpose of the project was to create assembly language programs for testing micro-processors, from hear the part of its name “macro”. After there it was used to solve various problems; for example, creation of test programs for pre- and post-silicon validation, design of Bayesian networks, creation of mathematical functions represented as trees, integer and combinatorial optimization and many more.

The second part of its name derive from “Genetic Programming” (GP), because it an evolutionary algorithm. For each step of the process a population of different solution is considered, new individuals are generated from two or more old ones (sexual reproduction) or by one (asexual reproduction). The probability of reproduction of the individual, which ones are selected to generate the new population, is not completely random, because best solutions have better chance to reproduce and pass their good quality to the new individuals.

Eventually all the good quality (genes) collapse to only one individual generating the optimal solution.

After ten years of developing, the structure of the MicroGP is careful designed and it has pretty distinguished logic scheme, thanks to its peculiar characteristic.

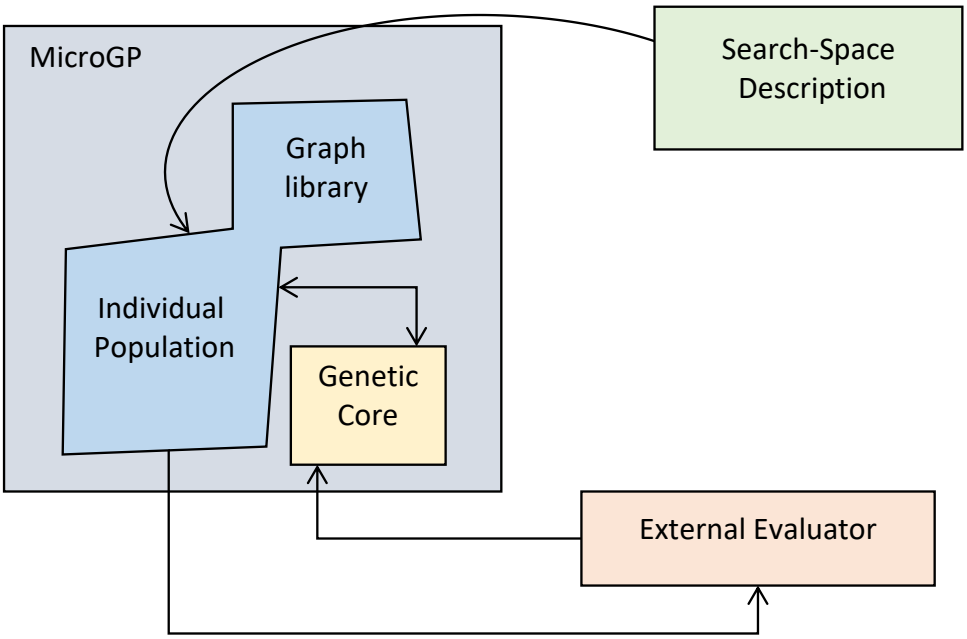

Figure 2 High level MicroGP logic blocks

The Search-Space Description is not built in the MicroGP, but the toolkit is able to create a population reading the structural information and Search-Space Description from an external source. The individuals must have the more general description, so they are built on a graph structure.

MicroGP cannot assign fitness value to its population, but an external evaluator is needed. This characteristic, that could be initially seen as a disadvantage, and the ability to get the description of the search-space from external sources are the key aspect that allows the toolkit to be as general as possible.

The logic section which perform all the genetic operation (genetic core) is not merged inside the population, but it must be seen as block puts beside the Population.

The current version of the MicroGP is the third, it is written in C++ and the Search-Space Description is provided through XML files. It is a robust, efficient and well written program, that can be used in several ambits and it has already successfully solved several problems. Unfortunately, this implementation has some problems that make its diffusion difficult.

Currently MicroGP4 is under construction and this new version of the optimization toolkit will hopefully solve all the problems that afflict the current version. The innovations consist in a new design of the general structure and of each internal element. The technologies involved are changed as well: Python is the choice for

developing the MicroGP4 itself, while JSON is chosen for encoding the information concerning the Search-Space Description and constraints.

2. MicroGP initial background

To better understand why the idea of creating a new MicroGP was considered and came true, it is necessary to analyze the peculiar characteristic of the entire project, focusing on merits and defects. To create a better MicroGP is necessary to know its characteristics in detail.

2.1 MicroGP

MicroGP was born to test microprocessor using Genetic Programming (GP).

Its name is the combination of its two fundamental characteristics, "µ" or micro (due to print issue) and GP from *Genetic Programming*.

The very first version was codenamed "Chicken Pox", because the isolation caused of that allowed Prof. Giovanni Squillero to write most of the library in a week.

MicroGP (ugp2) was an evolutionary tool, written again in C by Prof. Squillero, created for generating assembly programs and optimized for a specific microprocessor. It used the result given by an external evaluator and some internal information to efficiency explore the search space.

In this version macros, a parametric code fragments, were introduced and used.

It was developed in 2002 and maintained since 2006 and distributed, like the previous version, under the terms of the GNU General Public License.

MicroGP3 (ugp3 – µGP^3) is a complete rewritten of the toolkit in C++ and the current stable version. Its realization was supervised by Prof. Giovanni Squillero and Dr. Alberto Tonda and developed by about ten people and this tool is the result of around ten years of development.

In this version the programs became a genetic optimizer and not only a tool to perform test.

Another import change was to make it independent from the programming language generated, in other words it is designed to generate any language and it is possible thanks to the fact that the language constraints and syntax are given to the program externally and they are not built in.

2.1.1 Clearly Separated Blocks

At a macroscopic level it is possible to subdivide the toolkit in distinct and several blocks, this is a completely new concept in evolutionary computation. The main blocks of MicroGP are:

- *Search-Space Description*:
  A formal specification, where all the constraints and syntax of the search space are specified.
- *MicroGP*:
  The evolutionary motor. All the information read from the xml file are used to generate the wanted language and, thanks to evolutionary algorithms, a population of solutions (source code programs) are generated.
- *The evaluator*:
  an external part is necessary to evaluate the propose solutions, as in the oldest version. This is necessary to reach independence from the given problem and the space solutions. This aspect is the key feature of the whole project.

This specific design allows to solve any hard problems, even if they are not a real program, because what the program generates is plain text follow constrains and a given words dictionary.

If the evaluator gives back a consistent fitness for solution, and the problem is completely encoding in to the toolkit input files, there is (theoretically) no problem for which the MicroGP cannot find an optimal solution.

The Evolutionary motor is not able to calculate the fitness, because it does not know the problem specification and it could be used in an enterprise environment, despite MicroGP is under GNU General Public License. Fitness is calculating at genotype level and the toolkit work only at phenotype level, it maps this tow layer together only before to give the solution to the evaluator.

An evolutionary program works as a close loop: a population is transformed in a new one by changing and recombined its component individual and when a new individual

is created a fitness value is assigned to it. The new population is than modified and recombined again to obtain another one. The process will go on in this way until stopped by some external or internal causes.

But as mention earlier MicroGP cannot calculate any fitness and this is not a deficiency, but the feature that allow the evolutionary motor to be as general as possible, generating and optimizing solutions in any existing language for any problem.

In fact, this decomposition is the key to achieve versatility. So, the MicroGP approach is first to generate an individual at the initial phase or from other ones, then transformed into the object that represents and finally for each individual gets a fitness value.

2.1.2 How to Generate all Programming Languages

Sometimes more a thing is general and more difficult is to realize and more complex are the tools and techniques to have to use. This is the case of the MicroGP3 (and MicroGP4).

A parser is optimal to read and check a program's source code and it is very efficient thanks to it stack system used to analyze the read program.

Unfortunately, a parser is not sufficient to generate code, especially is not able to generate a general text without knowing anything in advantage.

To store a solution (text, program's source code) complete, or under construction, a stack is not good solution, because the manipulation of a solution involves add or remove small and big part of the solution itself in any times. Also add and remove parts are not the only functionality needed to manipulate a solution, link and unlink code fragments is essential. In addiction two parts may have more than one connection with different information and meaning.

2.1.3 Definition of the Individual

A very important step is to identify what is the individual in the MicroGP, what are its fundamental elements, and how to generate and manipulate it.

An individual must be able to allow to add new elementary part, link them together many times, to remove and link its constituent elements; it is also fundamental having the information where the links start and where they end, in other words a direction. With all this in mind a reasonable choice for represent an individual inside the evolutionary motor might be a multi-digraph, because it has exactly the behavior described.

Much more complex task is to handstand what is a possible representation for the elementary part of the individual or nodes of the multi-digraph, because this representation must be suitable for every possible language.

The real solution is not to define the way in which the nodes are represented but design a way to tell the program how a node must be: a macro, the general rules to specified what information a node has to carry on.

The link representation depends both how the multi-digraph is implemented and from the node description.

This freedom to represent the constituent elements is possible to achieve thanks to the division in three blocks of the MicroGP. In particular, it is possible to define how the Search-Space Description must look like in the evolution motor. In this way the information about nodes, links and constraints are told by the user, but how to specify them by the MicroGP.

In a general text, a single line or part of that can be adopted to represent the elementary part of the individual, in other words what it is the information a node must carry on. This is not the only visible characteristic.

A text could have macroscopic division, for example if it is a program source code could have at the beginning some precompiler directives, after that an isolation code part, then the main program and in the end other code parts and precompiler directives; another example could be a book, there is an index, follow by an introduction and several chapters.

A graph of any kind has not all the functionality to fully represent this information, it must be extended adding this macroscopic division and defining this section.

In the individual a section is defined in this way: it can contain several nodes, that can be linked together inside a section, or it can contain other sections, creating a hierarchical sections structure; a node can link to another sections or macro code fragment can exist without link; a node outside a section cannot be exist.

The multi-digraph must be extended to include all these features.

2.1.4 Technologies and Limitations

The technologies used to develop MicroGP3 were C++ and XML.

C++ was chosen because it has all the functionality of the C language, strong static typing, procedural and imperative programming language, with the versatility of the objected programming language in addition it has virtual function, operator overloading, multiple inheritance, and a standard exception handling.

This programming language is very diffuse, and it has bene standardized for many years now.

Like the C language, it is compiled into machine code. This characteristic allows the program to be efficient in the use of the machine resources and its execution is very fast. These aspects are true, if the program is well design and written, obviously if its design and/or implementation is broken the program result is unstable, slow or in the worst case unusable.

The other technology used was XML (eXtensible Markup Language). It is a metalanguage, scilicet a language capable to define its own element. In general, it is used to describe structural document and general information.

It is not only possible to generate elements and their attributes, but it is possible to define a set of roles that the particular XML file must follow to be consider valid and libraries exist to perform this check in an automatic way, for the majority of the programing language.

For example, it possible to specify that an XML file is valid for the application only if contain at least an element of type section which has the attribute “id”, or the file is valid only if the section element contains other section element which in turns contains macro element.

This feature allows to decide in an unambiguous way how the MicroGP3 user should define, macros, sections and all the input parameters needed for the programs itself to work correctly.

But unfortunately, in these two technologies we can find the causes of the lack of diffusion of the toolkit.

C++ is a very efficient and performant choice, but people find difficult to use the project since it is distributed as source code. MicroGP3 is made available in this way because it is writing in standard language independent from the hardware and from the operating system, so theoretically everyone with every machine could download, compile and use the tool.

But people complain about the difficulty of the operation, especially on Windows systems and they are not able to compile it at all and obtain the execute file for their

machines. So, a solution was to provide Win32 binaries and they were download and executed without problems.

MicroGP3 is under GNU General Public License, so everyone can download the source code, modify it and use it under the license agreement.

In this way it is possible to use the internal functions of the toolkit and expand it to meet all the needs.

C++ constructs are very powerful, but unfortunately, they can appear to a non-language expert complex and counterintuitive, so who had interest to modify barely success in the task.

While who only uses the MicroGP3 as it is complains about writing the XML, because people find writing the XML document difficult and frustrating because they cannot manage to create a valid roles and macro documents in a short period of time, but to figurate how to write a correct one requires several attempts.

2.1.5 Awareness of the Problems

Once all the MicrGP3 problems are analyzed, the weakness of the third version are the starting point for the MicroGP4.

At the end of the develop, MicroGP4 must have at least all the features of the previous version: three separate and distinct blocks (Search-Space Description, Evolutional Motor, Evaluator), the capability of creating any language, reliable and efficient code.

The main reason to create a new version is obviously to eliminate the limits of the previous one.

In the MicroGP4 compiling the toolkit itself must not be a problem anymore with any operating system. The solution must be to adopt a technology that allow people to execute programs source code while they are not used to develop programs.

Furthermore, for those who are familiar to write and to develop code, it must be easy to add functionality, modify or use only a part of it.

Another thing to change is to find a more user-friendly alternative to XML file, because if the tool is easier to use, more people will use it.

And finally, a greater idea will be to find a better way of distributing and downloading the project.

Another fundamental aspect, that must be considered, is the fact that the slowest part of the MicroGP optimization is not the MicroGP, this could be strange, but the

evolutionary operators are very fast as the generation of new individual. The slowest part is the external evaluator and it takes about the 95% of the total computational time to assign a fitness value for each individual. As mentioned, it is not a problem, but a fundamental aspect to consider during the design phase.

2.2 Python 3

One of the biggest changes from MacroGP3 was the choice of Python as developing language.

Python is an interpreted high-level programming language for general purpose programming, created by Guido van Rossum in the early 1990s.

Python is a dynamic type, multi-paradigm language. It supports object-oriented, imperative, functional and procedural. It also has a pretty solid and exhaustive standard library. The user can override the standard library base objects, inside a program. The type checking is made at run time (dynamic type) and the language has an automatic memory management.

Everything in Python is a first-class citizen, or in other words everything is an object. So for example, a function or even a class could be used as input argument of a method and they can be the returned value.

2.2.1 Interpreter

Because of it is a scripting language, Python needs an interpreter. There are several, mostly open source and they are compatible with many Operative System.

Some of the available implementation:

- CPython is the default language implementation
- Jython, written in Java, supports up to Python 2.7
- IronPython, written in C#, support up to Python 2.7
- PyPy, written in Python, uses a JIT compiler, able to generate at runtime, machine code from Python code. Now a day RPython is used to transform PyPy versions in c and then to compile it.

CPython is the reference and the default implementation of the language. It is written in C, and it is the most widely used.

CPython is available for Unix-like systems (Linux, MacOS, Solaris, etc.), Windows NT, embedded and not only environment (Nintendo DS, Nintendo Gamecube, PlayStation 2/3, Android, Apple iOS, etc.).

2.2.2 Evolution of the Language

The first published code by van Rossum was labeled version 0.9.0. In this version there were already present classes with inheritance, exception handling, core data types (str, list, dict, etc.) and functions.

In versions 1.X (1.0, 1.2, up to 1.6.1) python acquired functional programming tool (lambda, map, filter, reduce) and support for complex number.

The versions 2.0 up to 2.7 introduced list compression, a garbage collector system.

A structural important change was the unification of the types written in C and the classes written in Python, so the language's object model became a purely and consistently object-oriented language.

Python 3 broke completely the back compatibility, because of some minor change in the syntax and some types were removed or fused together. But fortunately, a conversion tool (from python 2.x to python 3) exists to solve this issue.

During the development years of Python, duplicate solutions to solve the same task were implemented in to the language, so one big change was trying to remove all the duplicate ones and also to remove old constructs and modules.

Examples of these operation are: str and unicode were merge together into str type,the behave or of division operation were changed.

The last Python stable version is 3.6, version 3.7 is curly in development.

2.2.3 The "Zen of Python"

The entire philosophy behind Python is obtainable through an Easter Egg inside the interpreter.

In fact, typing "*import this*" this text would appear:

```
The Zen of Python, by Tim Peters
```

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

All these sentences can be summarized with: keep the code as clean, linear, readable as possible and avoid not necessary complex solution.

Matter of fact code tabulation is very important, a wrong tabulation could generate syntax error or unwanted behavior, because of code blocks are not delimitated by brackets, but by the tabulation.

2.2.4 Python Package Index

Python has an official third-party software repository called Python Package Index (PyPI).

Most of the code, contained in PyPI, is Python package and it is written by the community that is very active.

It is possible to search package by keywords and to filter them by metadata information, for example like the license type or the Operative System compatibility.

In fact, PyPI can store for a single package the previous versions (for example version from Python 2 and for Python 3), compatible version for different OS, metadata, precompile source.

When a package form PyPI is downloaded, all the library or package which it depends on are also downloaded in an automatic way, if they are available.

2.2.5 Advantages and Disadvantages

Write a library, or a program such as MicroGP4, in Python, rather than in C or C++, could make the entire project slower than to write and build it in native code, because of python is a scripting language.

But this problem is not so relevant for the MicroGP4 thanks to the fact that the significant slow part is to evaluate the solution produced, in addition to that a Python program is really interpreted only the first time. It is pre-compiled in byte code the first time and the byte code will be executed each time.

On the other hand, the advantages are very relevant, and they can hopefully solve the MicroGP3 problems.

Thanks to its designed simplicity, Python is simpler to learn at a basic level and to have all the knowledge to use a complex program in easy way.

A program written in Python is very easy to download and use, if it does not need any binary addon. A single simple command “pip package_name” allows us to download a package from PyPI and it is just ready to use, without compiling or configuring it.

More and more people are starting to use Python and all of them could take advantage of the MicroGP4 just typing a single command.

If there is any need for which MicroGP4 has not the wanted features, there is a multitude of well written and design open source library to be combined to solve the problem.

Python allows to create more flexible, maintainable, easy to read and to use code. In this way future expansions and update are simple to implement and perform.

Thanks to the garbage collector, there is very few memory leach, unless broken code design.

MicroGP4 is written in this programming language because it allows to develop a new version of the evolutionary algorithm that has almost all the strengths of the previous one written in C++, with having intrinsic in the language solutions for lots of the limitations of the MicroGP3.

Because Python is interpreted, compiling MicroGP is not a huge issue anymore, the only effort is to install the interpreter and it is a straightforward task.

The problem of distribution is solved uploading the library to PyPI, so anyone can use the command pip to download and use it.

The high readability of the language allows more and more people to understand the source code and modify it as needed.

2.3 Toolkits Adopted

One big advantage to develop in Python is that the community is very active and offers a variety well written and design open source programs and libraries.

And they are compatible with all the major and standard technologies in any research areas, from mathematic to distributed, simulation, graphic, data managements and many more ambits.

2.3.1 NetworkX

A single solution is a graph, to be more precise a multi direct graph. It is a graph where each edge has a direction, two nodes can be connected by more than one edge and it can have cycles.

In principle a simple list may seem sufficient, because a program is a sequence of instruction executed by a CPU (in a single thread/process scenario). But this behavior is not enough to represent a real executable flow of data. A program can fallow different instructions path, it depends on logic conditions and language feature, and two parts of the program may have been connected multiple time with multiple meaning.

NetworkX is library to manage network and graph, written purely in Python and it is very efficient and scalable, thanks to the fact it adopts only dictionary to describes graph (and network).

It allows to manipulate big graph in order of 10^7 nodes and 10^8 edges.

The authors of NetworkX are active and when a problem is discovered, in few time an update is released to solve the issue.

They have created a well-documented Wiki, in which is explain how to use every single part of the library. Also, the source code is well documented and easy to understand.

Because of the fact it is written purely in Python any of its object are extendible.

For all this motivation NetworkX is the library chosen as the starting point of an individual inside MicroGP4.

As mentioned in the previous chapter a graph is not enough to describe an individual totally (at phenotype level). A section structure must be created and integrated with the multi-digraph.

Locally it is really easy to do, thanks to the fact that all the parts of the project are developed purely in Python.

2.3.2 JSON

A lot of time was dedicated to the analysis of the information the program needs to acquire and to design a way to represent all of this as user-friendly as possible.

JSON (JavaScript Objective Notation) has few general elements that are possible to combinate to each other, also in recursive form.

The elements are:

- Native types: string, number, Boolean value (“true”, “false”), “null” value representing an empty field;
- Object: it starts with ‘{’ and ends ‘}’, the entire document must be an object and it contains a list of keys – values attributes. The key must be a unique string inside the object, the value any elements or a string on a number;
- List: it starts with ‘[’ and ends with ‘]’, inside could be a list of anything, string, number, list and object.

Combined these two objects it is possible to describe every information.

```
{
  "key":"possible value",
  "key2":[
    "1",
    2,
    {
      "key" : 3
    },
    [

    ]
  ]
}
```

Figure 3 A valid JSON file

JSON is a text format that is completely language independent form any programming language but uses conventions that are familiar to programmers of the C-family of languages, such as C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

3. System Architecture

When a project is written from scratch, design the software base structure is the more import part. In this development phase, all the fundamental structural choices are taken; once implemented they are costly to change and, in the worst case, change them could means start again the entire project.

3.1 Project Elements

MicroGP4 needs elements that are powerful, efficient, easy to understand, to modify and compatible with all the operating systems.

As a matter of fact, the project core is composed by several data structures that define the individual characteristics.

To have the chance to describe every possible language, an elaborate data structure is need, because the individual does not represent a simple set of text chunk.

Every single element must be able to carry on information of any type, like simple primitive type (e.g. string, integer, etc.), other section and live object from standard library and from a custom one.

This information is not something inside the elements, but we have to image it as a label on them or as metadata information.

The elements that compose the individuals are: Tag, Section, Graph and Macro. At a high level they can be represented in the following scheme:

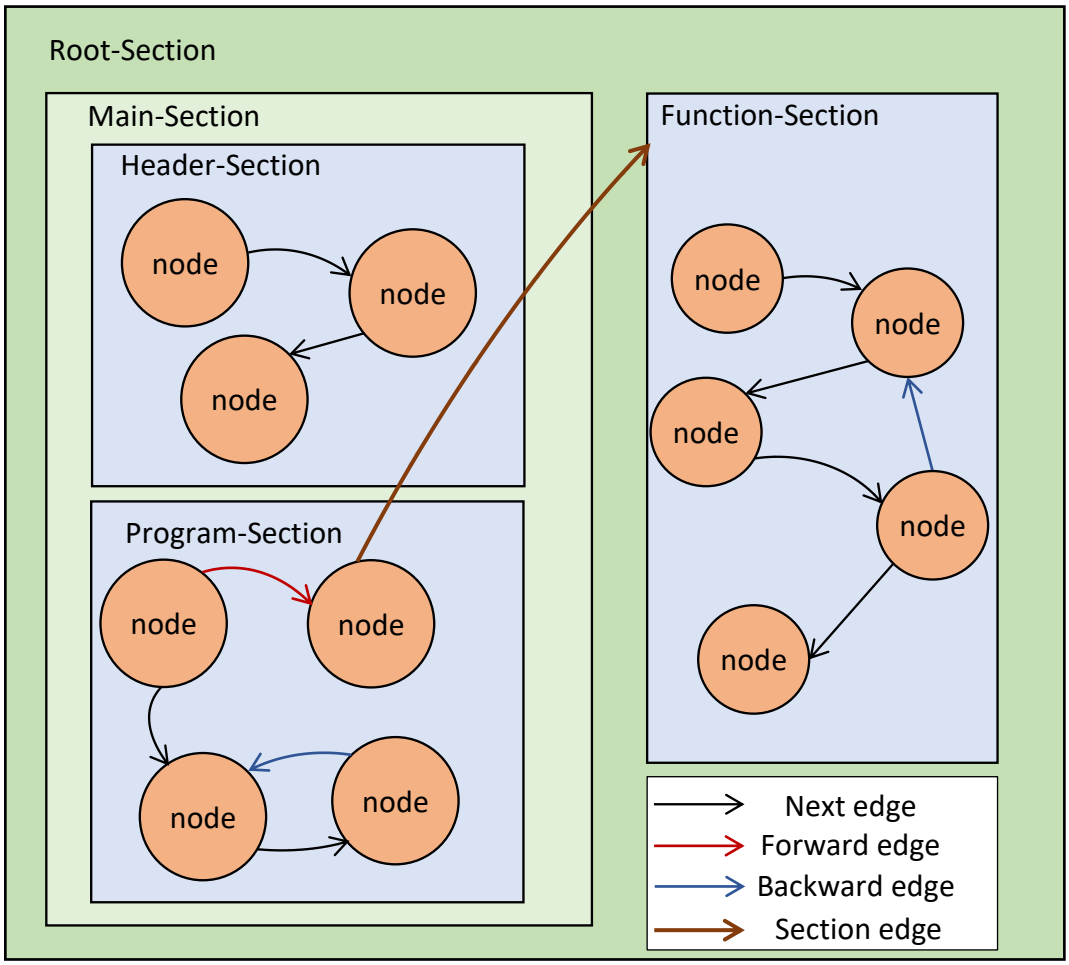

Figure 4 General individual representation

The entire individual must be inside a single Section, which can contain an arbitrary number of internal Sections and each of those inner elements can have the same content.

In the individual there is a limitation on what a Section can contain, in fact it can only contain either a list of sections or a list of nodes and a Section without content cannot exist.

Every node must contain a single rule generated from a Macro that defines can define the content of the nodes, in this case all the nodes, that refer to the Same Macro, have the same content. The true potentiality of a Macro is not to define constant, but to declare how to build the rule without specify its internal attribute, in this way, even if multiple nodes refer to the same Macro, they have different content.

For each element the information is linked to them through the Tag system, that allows to connect any data structure to any element of the individual.

3.1.1 Tag

Metadata is associated to every element, as labels on it, in the MicroGP4 these particular labels are told “tag”. They are completely independent both from the data structure and information they carry on and from the elements on which they are applied.

As a matter of fact, a tag can store any types of data, from a simple an elementary type (string, integer, float) to a complex and custom object (obviously also standard ones like list).

It can be used to store information important for the creation of the final solution, or information that are temporally, useful only in a specific life phases of the individual. To make it possible a tag as the functionality to modified, delete and recreate its internal content, and these operations are functional for any types of data that contains.

3.1.2 Section

Sections have been redesigned to be more simple, versatile, and general than in the previous version of the MicroGP.

While the basic features must remain unaltered, such as the order in which they contain element, because it is central to obtain correct solution at phenotype level, so sections must preserve it in every stage.

Sections have always been a vital element of the MicroGP, but in the fourth version they are modified at both implementation and conceptual level.

In fact, they are thought as standalone object not only as a supporting object for multi-digraph.

As standalone objects they can contain only sections and have a complete recursive structure, in other words a section can contains itself, or a descending section can contain a predecessor one.

The class is structured to be extended to add functionalities, change existing ones. As matter of fact it is a very versatile structure.

Sections are not limited to include only themselves, but the can contained any python object. In fact, the base class is also to be taken as example to how add functionality and how element should be contained by a section itself.

When sections are used in the graph as part of an individual, unlike the past, they are allowed to contain only a section or nodes, with obviously edges if there are any, but never sections and nodes together.

In this way the concept of epilogue and prologue is not possible anymore and it is violation of the design pattern just explained.

Fortunately, this is not a limitation, because it is possible to replicate the behavior of a prologue and epilogue. A way might be to create an external section which has inside three sections: the first is the prologue, the middle one contains all the real elements and data, and the last is the epilogue.

In multi-graph structure a section acquires the ability to have as sub element nodes, but it loses the possibility of having a true recursion, it can always have sections inside a section, but a section cannot have itself as sub-section, in general a descendent section cannot have a predecessor one as sub-section.

This is done to avoid endless loop operation on an induvial and keep the data structure as simple as possible.

The section data structure is also used to memorize all the information read from JSON. In this case it describes how an individual must look like, in other words it contains the syntax of the solution and how to compose together all the fundamental elements described for that particular language. In this case a section is used in more general way, because it contains a mixt of information to generate nodes, arches and sections inside the solution and it has the information to perform check at phenotype level. So, the sections continue to keep the possibility to be a completely recursive structure, to be as general as possible. This data structure is used to get the information to generate an individual and to perform all the check on it, to perform this operation the section is available from the individual; in MicroGP4 this particular section is told “meta-section”.

3.1.3 Graph

The data structure on which a solution is based is a graph, to be more precise a multidigraph. This is a graph in which the links between nodes has a direction and multiple connection between the same couple of nodes can exist. To implement the individual (and to control its correctness) is require knowing the direction of the edge; for guarantee the possibility of representing every language is also require having multiple and distinct edge between the same couple of nodes.

As mentioned earlier, a multidigraph is only the base of an individual, in fact multiple characteristic are added to it thanks to the sections and tags.

Any single node represents a single instruction, if the solution is a program of any kind, or more general an indivisible chunk of text. The information data structure is not possible to define in advantage, moreover it can contain small objects.

Also, the edges between nodes assume meaning and carry on data not completely predictable without knowing the problem and the language.

In both case the memorization is leaved to the tag system and it manages this situation in easy and clear way.

For the edges to define the information they carry on is not sufficient. They need to be also told which element they can connect, if only different nodes, a node with itself, or a node to a section.

In the general and standard graph representation there is not any particular order among the elements, but for a MicroGP solution is essential.

This need is due to the fact how a general text is defined, or a program is executed by a (micro) processor. Every instruction has an order of execution, statically or dynamically assigned due to logical expressions. In most of the case, change this order changes also the logic of the program.

So, with the addiction of this information, two graphs with the same element, but with different order are considered two different solutions, even if only two nodes are swapped.

In addiction it possible to identify and edge as backwards and forward and specified which of those an individual can have.

The sections are completed integrated inside the graph, because as mentioned before, is possible to link a node to a section. In addition to that behavior, a node cannot exist outside a section.

With sections it is possible to define macroscopic logic group of nodes and replicate this element in different points in the individual. As part of the graph, sections are used to add structural constraints to the nodes and define which type of instruction is concentered in which part of the solution.

In the MicroGP4 every element has a name with which it can be referred to. The Multidigraph is told “Sectioned Tagged Graph” when it acquires the functionality from tags and sections. In addition to those functionalities, the graph has the ability to create a unique identifier for each element. While a Sectioned Tagged Graph is told “Constrained Tagged Graph”, when it contains the meta-section and the functionality to check its correctness. In particular the Constrained Tagged Graph is the individual class.

The Constrained Tagged Graph represent an individual of an Evolutive Algorithms, so it must have the concept of genes and loci. Nodes have meaning of loci, but, before the application of an appropriate rule, they are empty waiting to be filled, and this is done by executing particular instructions called “Macros” which create a gene for each node.

3.1.4 Macro

Each single node of the Constrained Tagged Graph contains only one fundamental part of the solution, for example a single line of code or better a single instruction.

A MicroGP4 graph can have hundreds or even thousands of nodes, so it would be absurd to ask users to provide an instruction for each node. This approach would also limit the functionality of the toolkit, due to the fact that the number of nodes would be fixed as the content of the instructions.

A user defines a set of rules which have to follow by a group of nodes; for example, nodes, in a certain section, can contain an instruction with the scheme “random-word equal symbol random-number” or “random-word random-mathematical-operation random-word”, and nodes in anther sections can only follow one rule, which is different from the previous ones.

These sets of rules are called “macro”. A user must provide them to the MicroGP4 and he must tell which nodes in which sections have to use certain macros. They define only the genotype of the single instruction, while a random phenotype is assigned at the node creation phase and it is changed by evolutionary operators, in other words macros are used to create the gene (an information inside each nodes of the graph).

A macro can contain variable information (called “parameters”), fixed text or a combination of both. In particular the fixed test allows to specified constant inside a solution, for example in this way is possible to specify a fixed header or footer of the solution.

A gene, the result of executing a macro, is connected to nodes (or positioned in a certain locus) thanks to the tag system, and it can contain references of the connections between nodes and/or sections.

3.2 Recursive Structure representation

Most of the programming languages have recursive structure. A common and diffuse example of this kind of structure could be the “if else” block.

The “if” statement is followed by a logic condition, if the condition is verified the block of instruction under the “if” will be executed otherwise the code under the “else” statement will be executed, if it is present, because it is optional.

Inside the “if” code block it is possible to insert any possible code statement, so it is possible to have and other “if else” nested, the same discourse can be made for the “else” code block.

So, it is possible to have:

```
if condition
    if condition_1
        if condition_2
            ...
    else
        ...
else
    if condition_3
        ...
```

Figure 5 Example of "if else" code

An “if else” statement could be representing through sections and nodes following this scheme:

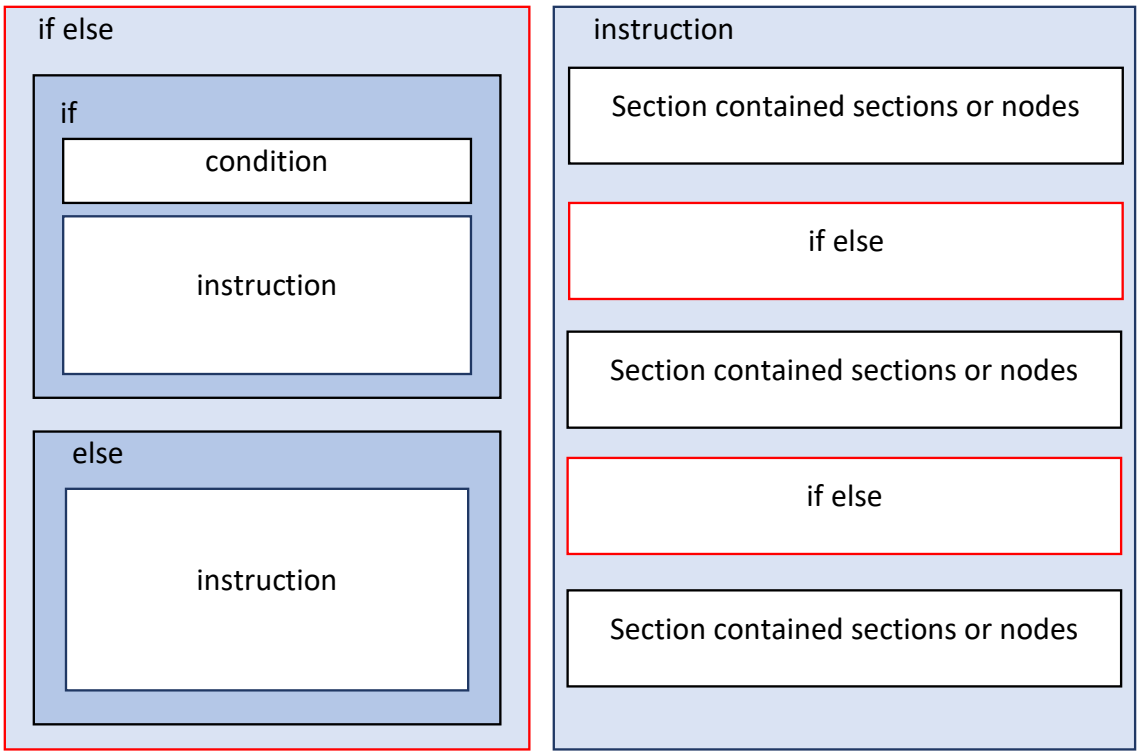

Figure 6 Possible "if else" division in to sections

In this example it is clear that the structure is recursive: the “if else” section has inside “instruction” sections that have “if else” block among its elements.

As described in the previous chapters, when a section is an element of an individual, it cannot have the ability to be a recursive data structure, while the meta-section has this feature and it may see a crucial inconsistency between the two structures.

In the meta-section, each section description has a unique name and when the section is implements a new instance of the description is created, which is refer to the section name in the meta-section.

The previous “if else” graphic example can be seen as the meta-section, while the real implementation inside the individua may look like:

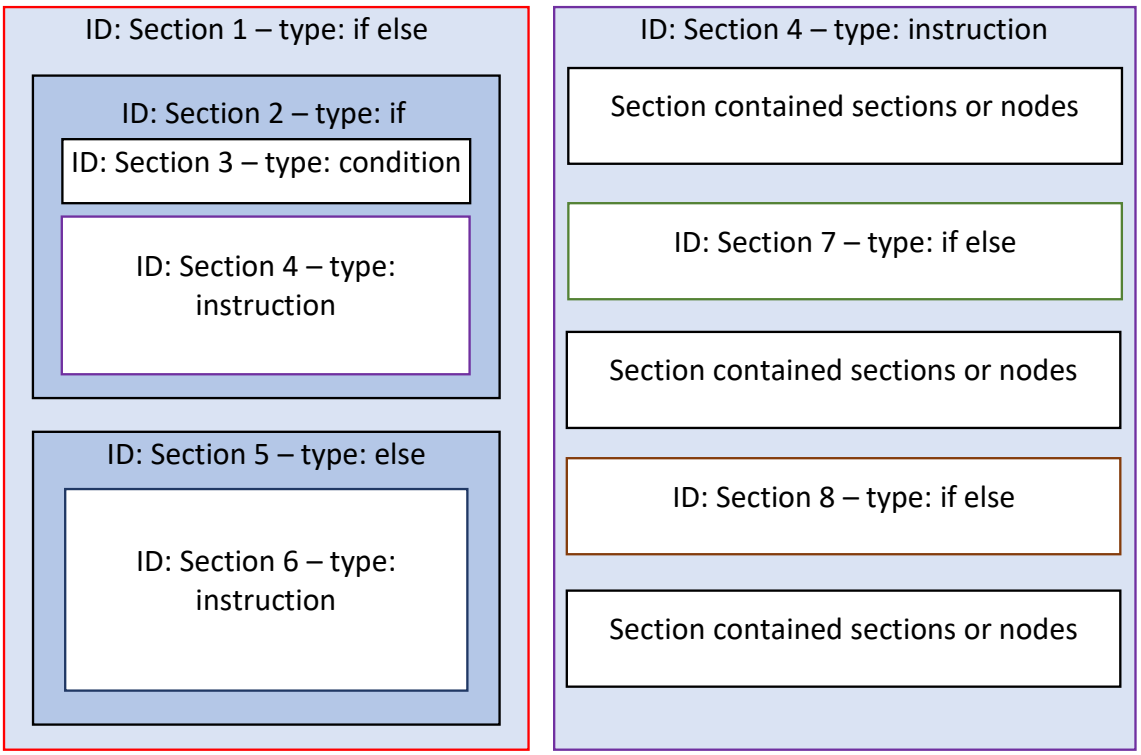

Figure 7 Possible "if else" code represented into an individual

In this last graphic example is only shone expanse what is inside the “Section 4”, the Section 6 has a similar content, with IDs of different elements.

This kind of representation may result to an endless structure, because in the example two sections of type “if else” are present within a section having type “instruction” which is a sub-section of an outer section of type “if else” and it is reasonable to suppose that the sub-sections have the same structure.

This problem is solved introducing the concept of optionality of a section, so a sub-section has only a certain probability to appear. In this example the “if else” section could be optional inside the “instruction” one, so there will eventually be a section of type “instruction” that will not have any “if else” blocks inside and the structure can finally terminate.

3.3 Separate Layer Structure

MicroGP4 must accomplish multiple complex tasks, so it is subdivided in layers and each of them achieves a single job.

In order to resolve a complex problem, an effective way to design the resolutive algorithm is to use the “Divide and Conquer” paradigm. It consists of subdividing recursively the main problem in smaller ones to the point they are simple to solve, after that all the partial results are merged together in order to obtain the final result.

A similar approach was adopted for the design of MicroGP4. The complex problem is to solve a hard problem adopting an evolutionary algorithm. At macroscopic level the toolkit is already divided in three blocks: *Search-Space Description*, Evolutionary Motor and the Evaluator. Luckily, the actual toolkit is the evolutionary motor.

The toolkit has to interface to the Search-Space Description and to the Evaluator, so it can be further subdivided in the parts that read information about constraints, about fitness of the solutions and in the part that computes the evolutionary tasks.

In particular the constrains and the syntax, with which the solution will be written with, are encoded into JSON files. There is a delineated part of the MicroGP4 that is dependent on the encoding of the information. Read it, the data is immediately decoded, after another small chunk of code analyzes the constrains and syntax and it generates intermediate objects that are independent from the previous phases and the next ones. The same approach is utilized to design and to develop every single part of the toolkit.

Taking the portion of the project that retrieves information from the Search-Space Description as an example, it is clear how the concept of dividing is adopted in the design of the MicroGP4. A big and complex task is reduced into smaller and elementary ones that can be rapidly solved. If someday the technology used to encode the information will change, the update of the entire project will be really fast and easy to do: the operation is reduced to simply having to replace a couple of functions. This is possible because of all the code that manipulates data in JSON format is concentrated in an isolable region of the program and its output is a general structure.

Each toolkit fragment performs a single task without interferer with the other parts, but code maintenance is not the only big advantage to follow a modular architecture.

Internally each layer has a hierarchy, in other words, the concept of lower and higher levels exists. A lower level provides a foundation for the uppers ones and each level is only used by the level directly above it.

The lowest part is the one that interacts with the JSON, then the next layer transforms a general data structure into meta-section, un upper layer uses it to create one or more individuals. The highest level will be the evolutionary part and the connection with the evaluator program.

Indeed, MicroGP4 can be used as black box, in other words a user can feed to it files with the wanted rules and syntax, he can connect it to an evaluator and after some cycles of iteration, he finally can collect the results. But also, thanks to its layer structure, a user can access the MicroGP4 internal structure and used it in different ways. For example, he can used it as library and utilize only the evolutionary parts, when it will be implemented, or another one of its sub-functionality. He can also create custom implementation of the objects used by the toolkit and inject them into it.

In general, an external program can be linked to any part of the MacroGP4.

3.4 The Needs for Checks

MicroGP4 is a general-purpose optimization toolkit. It is designed to solved hard problems by giving to it the description of the problem itself with constraints and other minor details to how generate the solutions and calculating for it the fitness values of the individuals.

In particular, as explained earlier, the toolkit manipulates a solution only at level of phenotype and it maps genotype only after applied the genetic operators right before to prepare the solutions for the evaluator.

While this characteristic is one of the fundamental features that allow the MicroGP4 to be as general as possible, it can also origin various problems. The toolkit is supposed to do not know anything about the genotype of the solution while it explores the search-space, so if it is altered and modified by some internal or external operation, the program will not detect this event and it might generate a complete unusable population.

The alterations can be generated by:

- Bugs inside the MicroGP4, this is the more obvious case. Even if a program is written following the best practices and it has a solid and well-structured architecture, it will likely have bugs or unintended behaviors. The solution is to solve the problems as soon as they are discovered.
- Not sufficiently stringent syntax described in the JSON. The encoded representation of the data into JSON files is designed to be very easy to utilize and to describe correctly the syntax of the solution. But the files are written by human beings, they can present imperfections in the representation of small detail, for example a wrong multiplicity or optionality of some elements, caused by distraction errors or a misunderstanding of how the information should be encoded.

- Operators that alter the genotype not voluntarily. Because they must be as general as possible, the genetic operators are built to be applicated to any problem.
- Operations that alter the genotype voluntarily. This is the least likely scenario, but it might be possible because it could be a way to navigate the solution space more. In view of the fact that the evolutionary part has not yet implemented, to design a solid architecture is a good idea to consider every situation, even if it is a very unlikely.

The alteration of the genotype is a dangerous problem, that can start an endless loop of invalid solution and in the worst case, no admissible solution will ever be found.

Solving this issue is nearly impossible. Even if MicroGP4 were a program without any bugs or defect and also the JSON files were perfect as well, alter the genotype is a peculiar aspect of its genetic operators.

So, inside the toolkit is possible that the genotype of an individual changes, but the individual as a function to check if correctness. In reality there is an entry point to start the checking, not a single function that inspections every single aspect. The more common characteristics are checked by a specific function for each aspect, for the reason that modularity continues to be a very important aspect as in the rest of the program.

The checks are based on the data contained in the meta-section, in other words in the data derived from the reading of the JSON files. As mentioned early, exist only function to check the common aspects and the JSON files could have ambiguity due to a wrong encoding of the information.

The more likely and easy example could be that inside the meta-section there is all the information for performing all the checks, but there are not all the functions to control every aspect; so MicroGP4 allows the user to add its own custom checks, this feature solves the case in which the JSON files are imperfect, thanks to the fact that is possible to create specific functions for the situation.

3.4.1 Error Checking

In a big project detect invalid data is a key aspect. Data cannot be valid in several ways, the more common are:

- Data has wrong type; for example, in a segment of the program text variable is needed but it receives a number.
- Data has wrong content; the type is correct but the variable carry on a wrong information, or variable is utilized too early or too late.

- Data has wrong both type and content; for example, a segment of the program receives an information designate for another segment.

The parts of the program, which interface with the external environment, are critical points with for the entire MicroGP4, because they need the correct information given in an expected way, and they give back data that follows a determinate structure.

These parts are seen also by the users, so if the users invoke them not in a correct way, the given errors might tell them what the causes of errors are and preferably how to solve them.

The error checking assumes a valuable rule for the using experience, because it must help users use the toolkit.

The interfaces with the external environment are not the only portions of the MicroGP4 that need to be checked. As matter of fact the entire flow of the data must be controlled and corrected the operations if it is possible, in any case any error must be reported.

Inside the toolkit the errors are divided in two general categories:

- Bugs due to the developing phase.
- Predicable errors which are the result of violation of constraints or the result of an incorrect use of the toolkit.

The first category of errors is composed of those that could be created in the developing phase, developers are human being and then they can commit mistakes. Everything that must be correct are checked by “assert”, an instruction that will stop the program if it contains a wrong logic condition and it can specify the reason why it happened.

```
assert isinstance(section_type, str), "Section_type must be a string variable."
```

Figure 8 Example of an assert instruction in Python

In this small code fragment, it possible to see how asserts work, and what types of mistakes they check: the first part is the logic condition that must be true (in this case the variable must be a string), the second one is the optional message that explains why the error occurs.

Asserts control the obvious things, like in this example the type of the variables, the length of an array, if a value in present or not and many other small mistakes which probably can happen during the development phase.

After the develops have managed write correct code, this type of checks will not be useful anymore, but an assert is an instruction as any other, that the machine executes taking time. Only one is not a problem but having hundreds or thousands of those impact the performance in a negative way. Locally they are chosen for a reason, when the program is not executed in a developing environment they vanish completely (by calling the python interpreter with “-O” or “-OO” or by distributing bytecode already

optimized). Thus, asserts slow down the program while it is written, in this phase, velocity is not so relevant.

The second category of errors is composed of those which must stay in a release environment. They are much less frequent, they do not control the obvious and, as mentioned early, they are used in the analysis of the constraints and to monitor the correct use of the toolkit.

For those the "exception" approach was taken. An exception is an object "raise" by the program, and if there is not any part of code that "catch" it, the conclusion is closure of the program due to a crash.

All exceptions of the MicroGP4 are extension of a particular base exception specially created. This architecture allows the users or a part of the toolkit to catch all the exceptions of the MicroGP4 by catching the base one.

There are many extensions of the base one, because each error class is as specify as possible regarding the type of error and also the classes are connected if the errors are connected to.

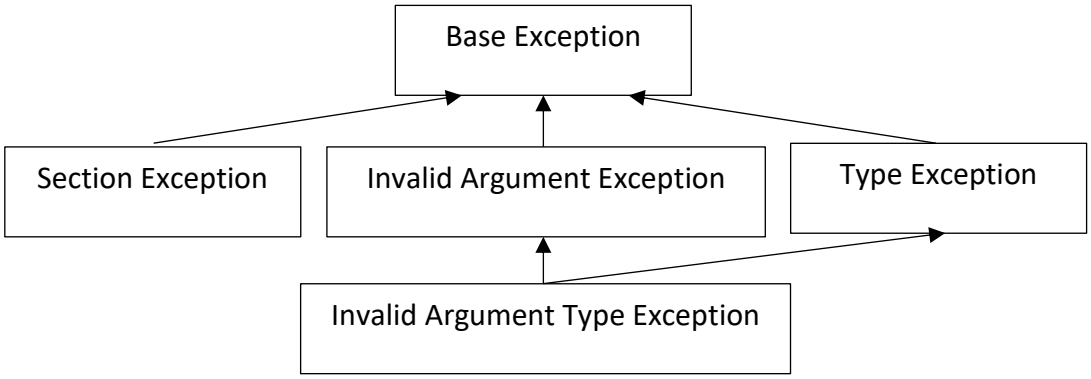

Figure 9 High level scheme of exception in MicroGP4

In this small example is clear how the concept works: every exception has the base exception as the parent exception of all of them, so catching it all of those are catch; the Invalid Argument Type Exception is a more specific case of two exception because even if it is applied for arguments the error regards the type.

An exception can carry on a message to indicate why this event occurs and not only the type of error.

```
raise Microgp4InvalidVariableType("It must be an int variable!")
```

Figure 10 Example of exception in Python

In this way a user, but also a developer, will handstand both what has just happened and why, so the cause could be easily discovered and solved.

This example is very similar to that done for the assert, but what is different is the motivation being it. This one could be checking some variable in the main which must be inserted form the user, or a parameter read from the JSON files. Both these situations are not solvable in developing phase and they depend on the usage of the toolkit and it cannot be foreseen.

Exceptions are not only used to verify the correctness of the input parameters, files, or the general usage of the MicroGP4, but they are one of the bases of the constraints checking.

3.4.2 Dynamic Constraint Injection

As described above the genotype must never change, but sometimes it happens. To steadily verify the feasibility of the solution checks are fundamental.

In some case if the genotype is altered, it is impossible to mapping the phenotype on it and dump the solution or even if MicroGP4 managed to success the operation, the generated solution is probably useless.

Constraints checks are very important, they must be able to correct any problem, with any descriptions of the solutions. In reality it is impossible to know everything of every problem in advantage, so differently from before it is extremely simple to add custom checks, but this is not the only new feature in the fourth version of this toolkit.

As the majority of the main data structure, the constraints are completely redesigned, allowing the users to have new and more powerful tools. The functionalities of the new system can be summarized as follow:

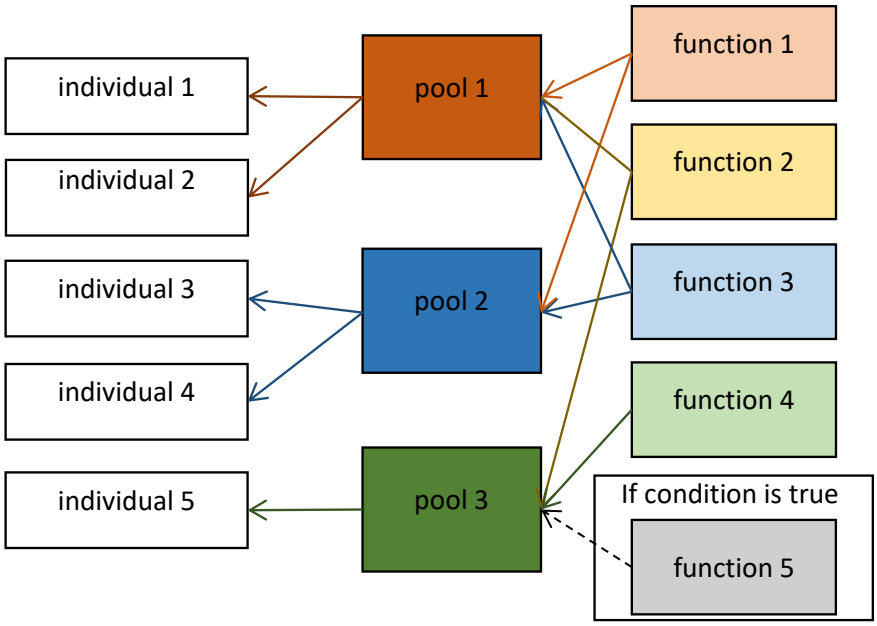

Figure 11 High level representation of the injectable constraints in the MicroGP4

- The new constants system permits to subdivide checking function in groups, called “pool”, and to have multiple of these subdivisions at the same time. Each group has a unique name to identified and to select it. These groups are objects and they are global for the entire program.
- When an individual is created, an object pool composed by constraint functions is assigned to it. When the correctness of the individual is tested, every function inside the pool is executed.
- Each individual can change between the existing constraints pool at any time.
- It is possible to insert the same check function inside different pools.
- It is possible to add new functions at any time, and any individual which uses the pool, can execute immediately the new instructions. In this way, a user can decide to check some constraints only in some situation, so the global efficiency of the toolkit can increase meanwhile the execution time can decrease. For example, some functions are needed only in some circumstance and maybe they are complex and require a long time of execution, so these particular instructions are injected inside the wanted pool only if the fitness values are in a specific range.
- Some solution may have different constraints pool. This feature allows to explore different portion of the search-space at the same time. Because when a new check is added, some solution cannot be considered valid anymore, at a set level this operation is equivalent of changing the set itself removing some portions. Practically, if the initial search-space can see as the union of different sets, for example “AUBUC”, after adding some constraints in a pool, this one

will allow solution only from “A∪B” and another pool, with the addition of other constraints functions, will instead allow solution only from “B∪C”.

The functions, that are added into a constraints pool, must follow few rules:

- The prototype of the function must follow a scheme:
  - the return value is ignored, to indicate a constraints violation it must raise an exception, in this way the search of an error inside the solution is interrupted immediately and no more code is executed in a useless way;
  - the individual is the only input parameter, because inside of it there is all the information that could need to analyze it, any other data can be directly inserted into the body of the function.
- The functions must do not deepens of the execution of the other ones, because the order in which they will run is randomize. They are designed in this way to encourage the creation of functions which accomplish only one small task. An aspect of the randomize execution of the functions is that eventually conflicts between those can be discovered and fixed.

3.6 Parameters, Classes Generator

Parameters are a key part of the toolkit, because they are the dynamic attribute of macros and with the way how nodes are interconnected, the components that allow to generate different phenotype from the same genotype.

Through an analysis of multiple programming languages, it is possible to notice that they have in common the same types, but they can have different representations. For example, if we consider an integer in C++ language it is possible to find out that it stands in a range between two constant, “INT_MIN” and “INT_MAX”. Of course, it is not the same for Python, whose integer values do not theoretically have a range.

Considering this kind of difference in MicroGP4 has been designed general types (or abstract types) and the users can provide the specific of the parameters through JSON in the same way they can specify any other solution syntax and constraints.

A user can describe only the parameters that are expected by the MicroGP4. In reality what a user describes is not the parameters itself, but its types and the toolkit is be able to randomly create a parameter that is inherent with the described type.

Different types may have different information needed to be defined. For example, the integer might need a range of validity instead a word might be not a random text, but

a random choice between a set of words or symbols. So different attributes are asked inside the JSON files for diverse type descriptions.

In more details what is inside the MicroGP4 are general types or meta-types which have to be specialized in order to be used. Inside the toolkit exists already the more common ones:

- Integer number.
- Floating point number.
- Set of specific words, symbols and certain constant number.
- Binary values and array.
- Reference, link type.
- internal and special parameters.

As mentioned early these are only the more common and represent also an example how to add general type into the MicroGP4. The procedure is really simple and few standard functionalities need to be implemented for each new meta-type.

Once the parameter types characteristics are read from the JSON a python class is generated, in this way a new variable is created whenever necessary in a macro. The parameter object contains the data to represent and it offers all the functionalities to modify and to execute the genetic operators.

The generated classes must follow a rigid scheme, they have to provide some essential characteristics:

- Random generator of the wanted value. When a parameter object is created, an option is not to set an initial value and the parameter must create a random one.
- Check of the value. In the creation the other option is to provide a candidate value and it must be controlled. There are genetic operators that can change the value of the parameter and after that it must be checked.
- General mutation function for the value. A function which mutates the parameters value following the normal distribution probability.
- Dump of the values. The mapping between the phenotype to the genotype, in other words how the value must be represented.

When a new individual is created, every single parameter is generated ignoring what its content is, the same approach is taken in the mutation step, thanks to the common functionality describe above. The random value generator and the mutation operator must be independent of the position inside the individual and the order in which it is inserted. To achieve this behavior each type should be independent from the graph and its structure. And this is a key aspect that the used has to respect when add new meta-type, he must obviously respect the architecture described early.

Unfortunately, there are few parameters that cannot be in part or totally independent from the individual structure and they are already present inside the MicroGP4.

A special type is a parameter that represents the node in which it is contained, so it is tied to the node id and cannot have random generator or mutation, because these operations are unnecessary and without of meaning, due to the fact that the node id is unmodifiable.

The other general type strongly dependent on the individual structure is the reference meta-type, because is mapped on the edge of the graph and it must contains the information of the source and the destination.

References allow to insert into the solution links between elements different form the one that indicates the order of every single nodes and they allows also to connect a node with a section.

For this meta-type the random generator and the mutation operation must be present, in fact the destination is not fixed un must be possible to change it, while the source must be always the same. The mutation is not limited by the strongly correlation of parameter and graph, because the all nodes and sections are covered every possible possibility of mutation, a similar speech is valid for the random value generator.

These two statements are true if and only if the structure of the individual is already completed and what are missed are only the edges. This is a significative problem during the individual creation phase, the solution was divided this step in two phases:

- The first one is creating all the nodes, section and parameters which are independent from the edge, and place only those that are dependent, without generating an initial value.
- Generate the initial value of the remaining parameters.

This approach can be taken by meta-types created from the users.

3.7 Extendible, Plugin System

The main task, during this thesis work, was to create the foundation for the entire MicroGP4 both creating a solid base on which the rest of the toolkit will be developed and define the data structure and pattern design that the next developers and the users must fallow in order to add functionalities to the project.

Analyzing the structure of the major components of the MicroGP4 it is possible to notice, that everything is mean to be as general and extendible as possible.

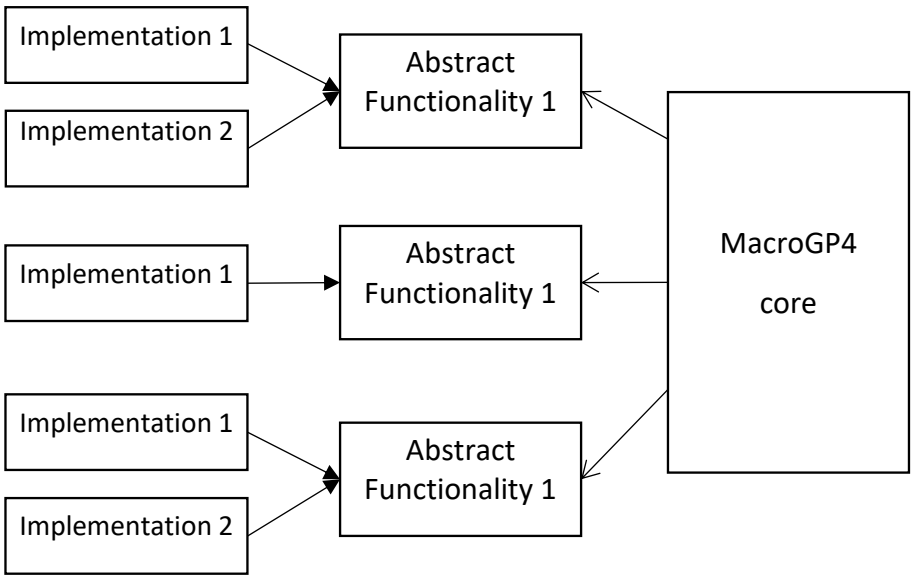

Figure 12 High level representation of the MicroGP4 architecture

The entire project may be seen as multiple plug-ins connected to the MacroGP4 core. In fact, the toolkit performs operations on the abstract representation of the functionalities, which they have several implementations.

The JSON is chosen as the markup language to define the syntax of the search-space. Using this specific data format allows to add new attributes and changing the representation of the existing ones by modified a small part of the toolkit.

Inside these files are described parameters, macros, nodes, sections characteristics and other additional information. In fact, it is possible to specify the version of the file content, in this way it possible to have certain files with a particular data description and other with a different one, The MacroGP4 is able to handle different files with different formal contents as long as they have the information of which version of the data they belong and obviously the toolkit must have the functions and methods to read the wanted version.

When new functionalities are added, like new packages, classes, methods and functions, also new error checks must be implemented as well. They must provide the information about what has caused the error and how to solve that if it is generated by an improper use of the toolkit. Inside the MicroGP4 is define where to add new exception and what meaning and relationship must have with the other exception. In other words, it is defined the inheritance pattern, which class has to extend which exceptions and on what occasion they must do it.

One of the most general and versatile parts of the toolkit is the Section. It is an empty container that can perform every operation on its content and it can carry on any types of data.

In the project is used as one of the core classes of the individual and by a stand-alone object to carry on the syntax of the search-space. This data structure can be used in future case to carry on different data which has to be constantly read, write and update.

The major functionality is the Constraints Tagged Graph and it is the class with defines the individual as it is now. This is a complex and delicate part of the toolkit and it is the result of the studying concerning the problems but also the result of the design and developing of multiplies functionalities merged together.

The base class to implement the graph is taken from the library NetworkX and on top of this one, several classes are built before to obtain the one which represents the individual inside the MicroGP4.

If in the future developing new functionalities are needed, the Constraints Tagged Graph defines the base structure and base methods to extended and how to interact with the lowest part of the project.

Another major functionality is the parameters system. Inside the MicroGP4 the more common types are already described, but they are the example of how add new ones, both in the recommended case in which the parameters are independent from the solution structures and elements and in the case in which it is impossible to achieve this characteristic.

All these data structures and systems are very different from each other, but they must work together. All these parts are connected with general structure and Python native object in a simple and elegant way. An example of that is the tag system, which allow every object to can carry on every data structure.

The parts of the MicroGP4 which interact and use the internal data structure are extensible as well. An example is that the induvial generator present inside the toolkit could be modified or flanked by several new ones.

This allows to have an initial population composed by different group of individuals at the same time. And it is possible to create solution concentrated in particular region of the search-space and to choose the best way to create the initial population for a certain problem.

4. Implementation

Another important aspect of the thesis work is to implement all the designed architectures and to test them.

A part of the new architecture is made possible thanks to the new technologies adopted. All the MicroGP4 elements are purely Python object and not native code, even if it allows to link programs part to library written in native machine code. This will remove the speed and resource limitations due to the fact python is a scripting language, but at the same time it will introduce critical aspect:

- For each operative system, a compiled version of the native code must be provided, and the python code version which interacts with the external library.
- An object imported from external library does not guarantee the interoperability with the Python standard library, and new issues could be born.
- The native code loses the clarity and the simplicity to understand and modify, typically of the Python language.

In other words, all the technological problematic of the MicroGP3 will come back with new ones.

The developing phase lasted several months, during this period of time a significant has been produced: 2943 lines of code have been written in 26 Python scripting files and more than 30 classes have been created. It is not possible to know the right number of classes, since many of those are generated at runtime.

4.1 Coding Test

The development of the MicroGP4 was not a single monolithic phase, but it was subdivided in several steps, every single developing phase was flanked by test, and every single test can be used as a particle guide how to use the toolkit and its internal functionalities.

The first part coded were those directly built on top the NetworkX class “MultiDiGraph”, which implement the structure and the functionality of a multi direct graph. So, the very first tests were a scripting file in which the new methods and the modified ones was called and analyze the response.

A similar approach of testing was adopting during the development of the new classes. But there are some parts that are not possible to checks their functionalities without interactions with other components.

For example, all the parameters extend the same abstract class and it is not possible to execute directly an abstract class without its implementations. Another example in the parameters ambit is the reference type. Because of it depends on the structure of the solution, tests on this meta-type had to be postponed until the individual was nearly a functional object.

After each single object was starting to work alone, and every functionality was tested if it is possible, the interconnected parts of the program were coded. In this final phase it was possible to see and to demonstrate that the architecture designed works well and in a clean way.

The first real test was to provide to the MicroGP4 three JSON files describing a simplified version of an assembly language and told it to generate small programs in such a way that were possible to verify by a human check if the toolkit created the wanted result. Once few bugs have been fixed, the toolkit created correctly random assembly source program with the given syntax and constraints.

MicroGP4 will be able to run for days or even weeks without stopping. So, the last and more complete test was designed to perform this check. The toolkit was feed with a small portion of the last assembly syntax for intel processors, and it as to generate a function in assembly that would be linked to a C program. The test consisted in to create functional source code for days and see if every single assembly function was executable and at the same times if the MacroGP4 was able to run for long period of time without crash or performance decrease.

The result was that the toolkit ran smoothly for the entire test periods and every single assembly function was executable.

4.1 Project Possible Application

How mentioned before, MicroGP4 is a multipurpose genetic optimizer, but thanks to its modularity the toolkit can be used entirely or only a part of it.

4.1.2 As It is Now

In the current stage of development, Microgp4 provides classes, object, methods and functions for:

- read information from the JSON and check if it has the correct syntax and information;
- generate the meta-section containing all the information of the syntax of the search-space;
- the principal parameters and how to mutate their internal value;
- generate one or more solutions from the meta-section;
- check if these solutions are valid;
- convert the individual in to source code and then save then in to files
- clone a solution.

In the previous chapters it is explain the whole architecture of the project, in particular those sections talking about the modularity of the toolkit.

All the internal functionalities present inside the MicroGP4 are usable right now, obviously is missing the evolutionary parts. For example, it is possible to get only the meta-section and analyze it to verify that all the wanted constraints are specified inside the JSON and check if they are correct; or a user might use only the methods to retrieve the information from the input files as Python objects.

As it is now, the MicroGP4 provides all the functionalities to generate random source code using the syntax given through JSON files. The resulting programs are syntactically correct and they can be adopted for testing purpose. The toolkit can produce hundreds of source codes in automatic way and, as they are totally random, even in a very short time. For example, this functionality could be very useful when a new micro-processor is created and it needs to be tested, due to the fact that even in hardware problems and bugs are possible and a way to check all the instruction register is needed. With random programs it is possible to verified if the processor reports uncorrected code behavior when it should and it completes them when programs are correct.

4.1.3 In the Future

As discuss in the previous chapters, the focus of the project was to design an extensible, modular and solid base structure, on top of it the whole MicroGP4 can be ultimate.

Several ways of created an individual are needed; the initial population must be as spread as possible to navigate the entire search-space.

Inside the MacroGP4, the more common meta-types are defined, but probably they are not enough to represent a sufficient set of parameters to generate the more common programming language, with complex syntax as the C or C++ languages.

These aspects are the minor ones to be developed for the toolkit, as matter of fact the important missing part is the evolutionary core. This part must design and code, but locally what is already done simplifies the job. In fact, a pattern to fallow to achieve modularity and to continue to have the plug-in behavior is present.

Once the evolutionary part will be designed and coded, MicroGP4 will be able to show its entire potential. MicroGP4 (including the future implemented last part) will be able to find optimal solutions for any hard problem. It will be able to do so only if the syntax of the search-space will be codable into JSON files and if it will be possible to create consistent fitness values for each solution through an evaluator.

4.2 Implementation Details

Inside MicroGP4 several classes and functions exist. Every single part works together to implement the whole architecture describe in the previous chapters. In this part the implementation of the major elements of the MicroGP4 is described in detail.

4.2.1 ConstrainedTaggedGraph

One of the more important project element is the class that represents the individual: “ConstrainedTaggedGraph”. When it is instantiated, it needs two elements the identification name of the checking functions pool which will be attached to it, and the meta section with all the information about constraints and the genes generators.

The class overrides and implements some standard graph methods, like the addiction of a node and edges, the calculation of fan-in and fan-out of a given node, to assign to them essential information for the construction the individual and for the future evolutionary part.

The ConstrainedTaggedGraph class has also no standard graph functionalities, such as methods for the handling of sections, the clone of an individual, the entry point method to start the check operation and for converting it into string printable on file.

When an individual is built, first it is necessary to create section and after that nodes. Because a node cannot exist outside a section and on creation it must be told in which section it has to be contained, while a temporary empty section object can exist. To have a complete and well-formed individual, basic characteristics are that every node is inside a section, empty sections does not exist and they contain only other sections or nodes.

The section element is an integral part of the graph, nodes can link to sections and they carry on essentials information about the individual. To ensure maximum efficiency from the library NetworkX, arches are not created to connect sections, but a special link is created between the nodes and the first nodes of the section that is linked. In this way all the low lever graph operations are unaltered, but at high level the individual has all the wanted features.

The information of which are the first and the last node in a section is very important both for the construction phase and for the dumping phase of a solution. Having this information allows at nodes level to identify sections.

4.2.2 Dynamic Constraints

A very innovative aspect of this MicroGP version is the possibility to inject constraints dynamically inside group of solutions in any times. As already mentioned, a constraints pool is assigned to each individual when it is creating, giving to it the identification name of a pool.

Each single pool of constraints is uniquely identified by a string. The entry point of this tool is the class “Constraints”, which cares about the interaction with the global and single object that stores all the pools.

When a string is given to the Constraints class, a reference to the pool having that word as identification is provided to the calling object, or a new constraints pool is created in the event that there is not a pool with the given string.

In each pool cannot exist duplicate of checking functions and they are executed in different order each time the validity check is performed on an individual. This is possible due the fact that functions are stored in “set” objects.

The Constraints class offers methods to:

- Retrieve the identification name of the current selected pool.
- Execute all the checking function of the selected pool on a given individual.
- Add new checking functions to the selected constraints pool

This particular data structure is designed to be really easy to manipulate and expand. This simplicity is possible also thanks of the Python language itself, due to the fact that every single language element is an object even a function (and a class).

4.2.3 VarType

JSON files contain several information, one is the description of the search-space types. For example, if a solution is represented throughout a C program, inside the files may be represented “int”, “float”, “char” types. For the int the description would be a number variable, with certain value as minimum and maximus representable values; a similar description would be needed for the float and the char might be represented as a set of symbols.

The MacroGP4 reads all the description and for each of them it creates a class at runtime. And inside the resulting object there is the real value. The main advantage is to have different genes generators for different types descriptions, with appropriate functionality, but they continue to be all managed at the same.

The second aspect it is possible thanks to the fact, that every generated class extend a base one the “VarType” abstract class. The class imposes which functionalities they must have:

- Generating new random value
- Checking a given value
- Mutate it
- Get and set the value
- Dump it

The only implemented aspects are the “__init__” function, or better the constructor and the getter and setter functions. The getter function simply gets the value without performing any particular operation, while the setter verifies, with the child checking

function if the received value is valid, otherwise an Invalid variable exception is raised. The constructor wants as mandatory parameter the id of the node in which the gene will be created and as optional parameter a value. If the value is given, it is checked as in the setter function, otherwise a new random one is created using the child random generating function.

5. Experiment

During the development of the MicroGP4 sever experiments have been performed. The main purpose of those is to test the correctness of the functionalities implemented.

Most of all the experiments were done with a PC with an Intel(R) core(TM) i7-6650U CPU @ 2.20GHz 2.21GHz and 16,0 GB of RAM memory, under the latest stable version of Windows 10.

5.1 Basic Solutions Generator

The very first experiment was to generate different individuals from the same meta-section, in other words each Constrained Tagged Graph was generate from the same set of constraints and search-space syntax. The syntax was really simple and contained only two set of particular word and it described a fake and simply set of assembly instructions, inspired by the syntax of the Intel 8086 processor.

The simplifications were about using a small sub set of the instructions, not considering the constrains on which register is used for which operation and not having any header or footer; all the instructions have the same syntax: [random mathematical operation] [random register] [random register].

As mathematical operations “ADD”, “SUB” and “MUL” were chosen, as registers only “AX”, “BX” and “CX” were present.

This was performed thank to a Python script file which used the functionalities of the JSON reader, constraints and Graph parts. It asked to the toolkit to generate an arbitrary number of solutions and, for each of them it printed some metrics, like number of nodes, section; using the debugging tool was possible to check some random nodes to inspect their content.

The result was that every solution was generated as expected and in a relative shot period of time. For example, to read the JSON, create the constraints and to generate one hundred graphs, whit more than one hundred nodes for individual, took about seven seconds. In the time statistics also the read the creation of the constraints were counted, because the two phases alone last less than a few tenths of a second.

5.2 Arch Solutions Generator

The experiment was the same as before, with the only addition of references between nodes and sections, multiple sub-set of words and integer value with the peculiarity to be printed in hexadecimal form to represent a memory address.

The “XOR” instruction was introduced and to map the connections unconditional jump were introduced.

In this experiment the syntaxes present were:

- [mathematical or logic operator] [register] [register]
- [mathematical or logic operator] [register] [memory address]
- JMP [unique label]
- Unique label used by an unconditional jump (always followed by another assembly instruction)

The interest part of the experiment was to see if the computational time would change and how. Generating always one hundred of Constraints Sectioned Tagged Graph, with more than one hundred of nodes, the computational time remained around seven second.

This experiment was created for two reasons, to see if the addiction of new elements can create a problem, the more significant reason was to check if the computational time is independent form the parameters types, but it depends only with the length of the solution. Repenting this experiment sever times we can conclude that the timing depends only on the length of the individual.

5.3 Dump Solutions

The next test was to see how the MicroGP4 behaves when dumping solutions with the search-space syntax and constraints of the “Arch Solutions Generator”.

To have a consistent set of solutions to analyze we chose to always dump a population of one hundred of individual with more than one hundred nodes but less the two hundred.

Tanks to the simple syntax it was possible to check the correctness of the printed solution thanks to human reading several random individuals for each experiment and always they were correct.

Only the solution dup part was measured and each run lasted about a second. So with this information we can conclude that the real heavy operation is the creation itself of an individual, but the creation of a completely new population is only a one task job and it is not a relevant issue.

5.4 Hash Function

The experiment consists into generating an intel no-prefix assembly program that performs a "perfect" hash function, in more details a hash function which does not create collision[2] in a given cardinality of value. It ran for two weeks without interruptions of any kind on a server equipped with a 4-core Intel i5 processor (i5-2500 CPU @3.30 GHz), 8GB of RAM, and running Ubuntu 16.04.3 LTS.

To perform the experiment a C file was create and used as an external evaluator. It fills an array with unsigned integer values, then it uses the function generated by the MacroGP4 to create for each of the values a hash code, finally it prints the collision number, that is the fitness value.

To provide the correct syntax to the toolkit, an existing assembly code was analyzed, in particular the parts that allow to connect it to a C program were taken into consideration.

The syntax provided to the MicroGP4 generates assembly code with constant parts in fixed spots and random parts of assembly code. In more details the structure of the generating program is: the first two lines are occupied by constant text, after random global variables of integer are generated in an arbitrary number, then other fixed lines are present, followed by the main function core, totally generated random and at the end another bunch of fixed lines.

A big issue is the possibility of creation of endless loop. They are not detected by the C program, so inside the solution only forward jumps are allowed.

A special Python script was created to use the functionality of the MicroGP4 as it is in the current developing phase. This script accepts a path for the input JSON and a path

[2] Two or more elements that have the same hash value.

where to salve the generated solution, in other words it is the interface with the external evaluator.

All these parts work together thanks to a bash script which it cares about providing the input and output path to the Python script, compiling together C and assembly source code, executing the binary file, reading the result of the programs and finally storing the best assembly code which performs the hash functions.

Regarding the chosen of the best hash function the behavior is quite simple: every time the bash script founds that the assembly program creates less conflict than the previous best, update the internal comparing value with the score of the new best one, and it saved it putting in its name the score.

The script provides also some metrics: an approximate time of conclusion, the number of iteration done and left, the current best result.

The bash script runs in circle all this operation for a given number of times, and eventually it finds the best function. During the experiment it manages to find the best function, so a hash function with zero conflict. But this is not the interesting part of the experiment; indeed, it does not stop when it found the best possible solution (an individual with zero conflicts), but when it executes for a certain time.

This test proved that the entire developed code is stable and solid, because the MacroGP4 managed to run for two weeks without the occurrence of any errors or internal problems.

6. Conclusion

This thesis work has two main purposes: the first one is to analyze the genetic optimizer MicroGP3, in order to discover and examine the problems this version has and that hinder its diffusion; the second purpose is to design and implement a new version of the toolkit (MicroGP4) with the intention to create an optimizer efficient, stable and well written as the previous one is. The new version must not suffer for the problems of the MicroGP3. In addition, the fourth version should have more functionalities.

The principal problems of the MicroGP3 are to research into the technologies adopted during its implementation. The third version of the genetic optimizer is written in C++ and distributed as source code.

The former characteristic allows MicroGP3 to be very efficient, as matter of fact its computational time is negligible if it is compared to the time in which a fitness value is assigned to an individual by the external evaluator; but unfortunately, the complexity of the language and of its data structure manipulation discourage people to develop custom aspect of the toolkit.

The latter characteristic allows to distribute the toolkit to every operative system, because it was written in standard C++ language and it is independent from both hardware and operative system; but people finds extremely difficult to build it, due to several errors in the compiling phase, especially under a Windows environment.

The other technology involved is XML; this markup langue allows to describe every data structure with a pretty robust syntax, but users find the data representation confusing and they do not achieve to create a valid XML in a short period of time.

To solve the developing language problem, Python is chosen as the new programming language for the implementation phase, because it does not need any compilation, due to the fact that is a scripting language. In addition, it has a clear and easy syntax, combined to powerful instruments.

The slowdown, due to the fact that it is interpreted, is not a problem, as its execution time is always ignorable compare to the evaluation time.

MicroGP4 is not a simply conversion of the old version from C++ to Python, but it has been completely redesigned in all its elements, adding some, removing others and modifying other ones.

The individual is built on top of a graph. The date structure is not implemented from scratch, but an open source library (NetworkX) is used as implementation of the graph structure. This choice has been done to have less code to develop and for maintenance, since this is a very standard data structure and it has been established for many years and from lots of people.

Few examples could be:

- The addition of the tag system into the graph. Tags allows to assign any information, with any structure, to any element. This aspect is fundamental because every element of the graph must have information assigned.
- The Section data structure has been completely redesigned, it has obviously conserved the purpose of regrouping nodes in different categories in the individual as in the MicroGP3, but it presents several new fundamental aspects. Sections can contain every element, also themselves, becoming a completely recursive data structure. Inside MicroGP4, they are used as one the main constituents of the individual, permitting to have infinite level of section containing other sections, while nodes must be contained only in the more deps sections which, apart from nodes, cannot contain anything else. The other fundamental purpose is to contain search-space syntax and constraints.
- Genes generators are dynamically created at run time adopting the information of the search-space syntax.

More in general, MicroGP4 is designed to be as modular as possible. Every single part has to be seen as a plug-in connected to the core of the toolkit. Actually, new implementations of the existing structures are possible to add to it and, if they follow the abstracts structures, everything is ready to use. This is possible thanks to the fact each part computes only a single simple task and, if it needs information from other components, it uses them as a black box.

In the fourth version of the optimization toolkit JSON is chosen instead XML, this is allowed to create lighter data encoding pattern and hopefully less confusing for users, thanks to the fact that can encode every information with few basic and simple elements.

6.1 Future Application

The implementation and design of the MicroGP4 is not yet completed, since the previous version was the result of about ten ears of work. But nevertheless, the basic structure, the core, how to add new elements and characteristics are defined.

The main missing part is the evolutionary core. Once this part will be designed and implemented the MicroGP4 will able to show all its potential. In fact, it will be able to find an optimal solution to any hard problems.

To achieve this goal many other steps must be done, but following a modular and clean project architecture, this will not be a long or difficult task and probability MicroGP4 will be one of the well know optimization toolkit available.

Bibliography

1. Squillero, G. Computing (2011) 93: 103. https://doi.org/10.1007/s00607-011-0157-9
2. Squillero, G. Genet Program Evolvable Mach (2005) 6: 247. https://doi.org/10.1007/s10710-005-2985-x
3. Sanchez E, Schillaci M, Squillero G (2011) Evolutionary Optimization: the µGP toolkit". Springer, New York
4. Encyclopædia Britannica. (2011) Encyclopædia Britannica Online. Online. http://www.britannica.com/EBchecked/topic/197367/evolution
5. G. E. P. Box. Evolutionary operation:
   A method for increasing industrial productivity. Applied Statistics, VI, no. 2:81–101, 1957
6. C. Darwin. On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life. Murray, London, 1859.
7. Turing AM (1950) Computing machinery and intelligence. Mind, no 9, pp 433–360
8. Koza John (1992) Genetic programming: on the programming of computers by means of natural selection. The MIT Press, Cambridge
9. Burke EK, Gustafson S, Kendall G (2004) Diversity in genetic programming: an analysis of measures and correlation with fitness. IEEE Trans Evol Comput 8(1):47–62
10. Cohoon JP, Hegde SU, Martin WN, Richards D (1988) Floorplan design using distributed genetic algorithms. In: IEEE international conference on computer-aided design, pp 452–455
11. Cohoon JP, Paris WD (1987) Genetic placement. IEEE Trans Comput Aided Des Integr Circuits Syst 6(6):956–964
12. Cataldo S, Chiusano S, Prinetto P, Wunderlich H-J (2000) Optimal hardware pattern generation for functional BIST. In: Design, automation and test in Europe, pp 292–297
13. Polian I, Becker B, Reddy SM (2003) Evolutionary optimization ofMarkov sources for pseudo random scan BIST. In: Design, automation and test in Europe, pp 1184–1185
14. Sekanina L (2004) Evolvable components: from theory to hardware implementations. Springer, Berlin
15. Higuchi T, Yao X (eds) (2006) Evolvable hardware. Springer, Berlin
16. Corno F, Cumani G, Sonza ReordaM, Squillero G (2002) Efficient machine-code test-program induction. In: Proceedings of the 2002 congress on evolutionary computation, pp 1486–1491
17. Dawis, E. P., J. F. Dawis, Wei-Pin Koo (2001). Architecture of Computer-based Systems using Dualistic Petri Nets. Systems, Man, and Cybernetics, 2001 IEEE International Conference on Volume 3, 2001 Page(s):1554 - 1558 vol.3

18. Dawis, E. P. (2001). Architecture of an SS7 Protocol Stack on a Broadband Switch Platform using Dualistic Petri Nets. Communications, Computers and signal Processing, 2001. PACRIM. 2001 IEEE Pacific Rim Conference on Volume 1, 2001 Page(s):323 - 326 vol.1
19. Laplante, Phillip (2007). What Every Engineer Should Know about Software Engineering. Boca Raton: CRC. ISBN 978-0-8493-7228-5. Retrieved 2011-01-21
20. "Software Engineering". Information Processing. North-Holland Publishing Co.year = 1972. 71: 530–538.
21. "The History of Python: A Brief Timeline of Python". Blogger. 20 January 2009. Retrieved 20 March 2016.
22. van Rossum, Guido (9 February 2006). "Language Design Is Not Just Solving Puzzles". Artima forums. Artima. Retrieved 21 March 2007
23. Python 3.6.5 documentation Online. https://docs.python.org/3/
24. NetworkX Online documentation. https://networkx.github.io/
25. NetworkX Online Wiki. https://networkx.github.io/documentation/latest/
26. JSON Online documentation. https://www.json.org/

Acknowledgements

In this last part, I want to thank all the people who have been close to me during this period and beyond. As this chapter will not cover academic arguments, but it is intended for reading friends and family, to carry out his task to the best of its ability, it will be written in Italian.

Ci tengo a ringraziare tutte le persone che mi hanno accompagnato in questo lungo viaggio, che culmina con la stesura della tesi. Poiché questo lavoro non è un compito isolato e fine a se stesso, ma è il prodotto di tutto quello che ho appreso in questi anni. Se soltanto elencassi tutti i loro nomi ne risulterebbe una lista più lunga dell'intero documento.

In primo luogo, vorrei ringraziare il Prof. Giovanni Squillero e il Dr. Alberto Tonda, per avermi seguito e consigliato durante il mio lavoro da tesista. Sono esternamente grato a loro per tutti i consigli e il tempo che mi hanno dedicato nello svolgimento di questo progetto.

Vorrei ringraziare Alessandro, il quale è dalle elementari che mi sopporta, ma soprattutto mi sprona ad impegnarmi in quello che faccio.

Ringrazio Simone che, con una risata, mi ha insegnato a non rinunciare mai ai miei obbiettivi, anche quando le situazioni sono complicate.

Ci tengo a ringraziare Chiara, che invece mi ha insegnato che quando si desidera qualcosa bisogna impegnarsi al fine di raggiungere i propri sogni.

Ringrazio Martina, che sulla quale posso sempre contare e durante tutti questi anni mi ha aiutato sempre con un sorriso.

Durante questo periodo ho incontrato molte persone splendide. Vorrei ringraziare Rocco e Marta che dal primo anno della triennale sono sempre stati ottimi amici e persone su cui posso contare.

Ringrazio sinceramente Giulia, Francesco, Flavia e Stefano per aver affrontato insieme esami e per i bei momenti di svago tra una sessione e l'atra.

Un ringraziamento speciale va a Luca, per essere stato molto di più di un semplice compagno di univeristà, ma per essere un sincero amico che ha saputo aiutarmi in molte situazioni.

Ultima per apparizione, ma non per importanza ci tengo a ringraziare dal profondo del mio cuore Eleonora, la quale ha saputo darmi la forza per continuare quando non ero

più certo di essere all'altezza di finire tutto quello che stavo facendo e con la quale sto condividendo momenti bellissimi della mia vita.

Però tutto questo non sarebbe stato possibile senza l'amore e il sostegno della mia famiglia. Un ringraziamento speciale va a tutti i miei famigliari che mi hanno sempre appoggiato e seguito in questi anni da universitario.

Infine, vorrei ringraziare tutte le persone che non ho menzionato prima, che mi hanno incoraggiato, ispirato, supportato e aiutato durate questi anni di studi, in momenti liberi e durante lo sviluppo di questo progetto.