POLITECNICO DI TORINO

SCUOLA DI DOTTORATO
Ph.D. Course in Ingegneria Informatica e dei Sistemi – XXVII cycle

Ph.D. Dissertation

Advanced Techniques for Solving Optimization Problems through Evolutionary Algorithms

Marco Gaudesi

Supervisors
Ing. Giovanni Squillero

Ph.D. Coordinator
Prof. Matteo Sonza Reorda

February 2015

Summary

Evolutionary algorithms (EAs) are machine-learning techniques that can be exploited in several applications in optimization problems in different fields. Even though the first works on EAs appeared in the scientific literature back in the 1960s, they cannot be considered a mature technology, yet. Brand new paradigms as well as improvements to existing ones are continuously proposed by scholars and practitioners. This thesis describes the activities performed on μGP , an existing EA toolkit developed in Politecnico di Torino since 2002. The works span from the design and experimentation of new technologies, to the application of the toolkit to specific industrial problems.

More in detail, some studies addressed during these three years targeted: the realization of an optimal process to select genetic operators during the optimization process; the definition of a new distance metric able to calculate differences between individuals and maintaining diversity within the population (diversity preservation); the design and implementation of a new cooperative approach to the evolution able to group individuals in order to optimize a set of sub-optimal solutions instead of optimizing only one individual.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Optimization problems are quite common in computer science, whenever whenever real-world applications are considered [90]. These problems are often impossible to resolve through an exact mathematical approach: this could be due to the inapplicability of an exact optimization method, or to a time consuming approach that do not satisfy application constraints – in such cases it could be necessary to perform a complete exploration of all the possible solutions in order to find the optimal one, unacceptable in terms of time and computational costs. Approach based on Evolutionary Algorithms might be well-suited to solve such optimization problems.

An Evolutionary Algorithm (EA) is an optimization local-search algorithm. It is based on a *population* and makes use of mechanisms inspired by the biological world, most notably: selection, reproduction with inheritance, survival of the fittest. Each *individual* represent a candidate solution. Starting from a random initial population, EAs improve it by generating new individuals through *mutation* and *crossover* operators, and removing less promising solutions. This result is an efficient exploration of the space of all the possible solutions, reducing the number of solutions needed to find a quasi-optimal one.

The aim of this Ph.D. Thesis is the study of EA techniques, and to investigate to new possible approaches for improving them. EAs were applied through the μGP [97] evolutionary tool, a generic EA optimizer-based that was designed and implemented in 2002 in Politecnico di Torino. Moreover, such approaches were applied to real problems, and to typical unreal problem discussed in literature in order to prove their efficacy.

The first improvement in my thesis is a novel approach of the evolutionary algorithms, in which the best solutions are composed by a set of individuals, instead to be composed of only one individual. This new EA paradigm has been called Cooperative co-evolutions algorithms (CCEA) in literature, but they are based on the idea of switching the evolution into co-evolutions of independent parts of the optimal solution [4][56][80]. Following the aforementioned ideas, during this thesis, it was developed a new cooperative evolution; this mechanism exploits the same basis of standard evolutionary theories: it uses a single population containing all the active individuals. The main novelty is that individuals are grouped in subsets of the populations, and cooperate together on reaching the optimal result; moreover, individuals can be shared with different groups.

To optimize this approach, two types of genetic operators work at the same time: the former are individual genetic operators, similar to the ones used in standard evolution; the latter are group genetic operators, useful to change individual configuration of a selected group.

This approach was exploited to automatically generate a set of test programs for diagnosis of microprocessor; the approach interestingly fits the problem, because would be obtained as best solution, a group covering the maximum number of faults, formed by individuals that are the more specialized as possible, covering a little amount of faults.

The second improvement addressed is the definition of a new distance working at genotype level. Evolutionary Algorithms are optimization mechanisms population based: this means that several solutions coexist together at the same step of evolution. Genetic operators are applied to selected individuals, and the generated offspring is evaluated and then inserted within the population; at this point, individuals are ordered by fitness value. Due to that the population should have always the same dimension at the end of each generational step, all the worst individuals exceeding the maximum number of individuals allowed to form the population will be destroyed. Following this approach, as the generations go on, individuals will become more similar to each other; this behavior is due to the smaller quantity of new genetic material that will be introduced within population by mutation operators, causing also the convergence of solutions towards an optima. This could be an excellent result, if the optima is the global one; otherwise, this can lead to a premature convergence towards a local optima, limiting the exploration phase.

To avoid this problematic, several approaches were studied [91], for example, the fitness sharing one, that shares fitness values between similar individuals. To apply this approach is fundamental to use a good way to calculate similarity (or distance) between individuals. This is the aim of the new distance definition, described in the following.

Another improvement, described in this thesis, regards the mechanism used to select genetic operators to be applied during the evolution. To better exploit the optimization capabilities of an evolutionary algorithm, it is fundamental to perform correct choices depending on the evolution phase: to settle to this task, automatic adaptive systems were designed, the Multi-Armed Bandit (MAB) and the Dynamic Multi-Armed Bandit (DMAB), that base the selection of genetic operators on their rewards [32]. Such methodologies update the probabilities after each operator's application, creating possible issues with positive feedback and impairing parallel evaluations. The DMAB techniques, moreover often rely upon measurements of population diversity, that might not be applied to all real-world scenarios. To fix these two techniques, in this thesis is proposed a generalization of the standard DMAB paradigm, paired with a simple mechanism for operator management that allows parallel evaluations and self-adaptive parameter tuning.

The first part of the thesis presents a summary of the current state of the art on Evolutionary Algorithm field, then μGP evolutionary tool will be described. In the following chapter new technologies, implemented during the Ph.D. course, are outlined, together with the discussions about application of evolutionary algorithms to real and typical problems.

Chapter 2 illustrates the complexity of the evolutionary algorithms fields, the state-of-the-art of this optimization technology and typical problems that are addressed through these technologies.

Chapter 3 is dedicated to a detailed description of the μGP Evolutionary Algorithm tool, that is the EA mainly used during this three years of doctorate course, focusing the description on new technologies that were implemented: the DMAB and the definition of the distance to calculate differences between individuals [47].

Chapter 4 presents improvements introduced within μGP ; the Group Evolution was applied to an optimization problem typically approached by standard implementation of evolutionary algorithms [25]. Moreover, other optimization problems are presented [46][21][23].

Chapter 5 concludes this thesis and drafts the future works.

Chapter 2

Background: Evolutionary Algorithms

Evolution is the theory postulating that all the various types of living organisms have their origin in other preexisting types, and that the differences are due to modifications inherited through successive generations. *Evolutionary computation* is a branch of computer science focusing on algorithms inspired by the theory of evolution and his internal mechanisms. The definition of this field in computer science is not well defined, but it could be considered as a branch of *computational intelligence* and may be included into the broad framework of *bio-inspired heuristics*.

This chapter sketches the basics of evolutionary computation and introduces its terminology.

2.1 Natural and artificial evolution

Fundamentally, the original theories regarding *evolution* and *natural selection* were proposed almost concurrently and independently by Charles Robert Darwin and Alfred Russel Wallace in XIX century, combined with *selectionism* of Charles Weismann and genetics of Gregor Mendel, are accepted in the scientific community, as well as widespread among general public.

This theory (called *Neo-Darwinism*) provides the basis for the biologists: through it, the whole process of evolution is described, requiring notions such as *reproduction*, *mutation*, *competition*, and *selection*. Reproduction is the process of generating an offspring where the new copies inherit traits of the old one or ones. Mutation is the unexpected alteration of a trait. Competition and selection are the inevitable strive for survival caused by an environment with limited resources.

The evolution process is a mechanism that progresses as a sequence of step, some mostly deterministic and some mostly random [71]. Such an idea of random forces shaped by deterministic pressures is inspiring and, not surprisingly, has been exploited to describe phenomena quite unrelated to biology. Notable examples include alternatives conceived during learning [22], ideas striving to survive in our culture [33], or even possible universes.

Evolution may be seen as an improving process that perfect raw features. Indeed this is a mistake that all biologists warn us not to do. Nevertheless, if evolution is seen as a force pushing toward a goal, another terrible misunderstanding, it must be granted that it worked quite well: in some million years, it turned unorganized assembles of cells into wings, eyes, and other amazingly complex structures without requiring any a-priori design. The whole neo-Darwinist paradigm may thus be regarded as a powerful optimization tool able to produce great results starting from scratch, not requiring a plan, and exploiting a mix of random and deterministic operators.

Dismissing all biologists' complains, evolutionary computation practitioners loosely mimic the natural process to solve their problems. Since they do not know how their goal could be reached, at least not in details, they exploit some neo-Darwinian principles to cultivate sets of solutions in artificial environments, iteratively modifying them in discrete steps. The problem indirectly defines the environment where solutions strive for survival. The process has a defined goal. The simulated evolution is simplistic when not even implausible. Notwithstanding, successes are routinely reported in the scientific literature. Solutions in a given step inherit qualifying traits from solutions in the previous ones, and optimal results slowly emerge from the artificial primeval soup.

In evolutionary computation, a single candidate solution is termed *individual*; the set of all candidate solutions is called *population*, and each step of the evolution process *generation*. The ability of an individual to solve the given problem is measured by the *fitness function*, that ranks how likely one solution to propagate its characteristics to the next generations is. Most of the jargon of evolutionary computation mimics the terminology of biology. The word *genome* denotes the whole genetic material of the organism, although its actual implementation differs from one approach to another. The *gene* is the functional unit of inheritance, or, operatively, the smallest fragment of the genome that may be modified during the evolution process. Genes are positioned in the genome at specific positions called *loci*, the plural of *locus*. The alternative genes that may occur at a given locus are called *allele*.

Biologists distinguish between the *genotype* and the *phenotype*: the former is all the genetic constitution of an organism; the latter is the observable properties that are produced by the interaction of the genotype and the environment. Many evolutionary computation practitioners do not stress such a precise distinction. The fitness value associated to an individual is sometimes assimilated to its phenotype.

To generate the offspring for the next generation, evolutionary algorithms implement both sexual and asexual reproduction. The former is usually named *recombination*; it necessitates two or more participants, and implies the possibility for the offspring to inherit different characteristics from different parents. The latter is named *replication*, to indicate that a copy of an individual is created, or more commonly *mutation*, to stress that the copy is not exact. In some implementations, mutation takes place after the sexual recombination. Almost no evolutionary algorithms take into account gender; hence, individuals do not have distinct reproductive roles. All operators that modify the genome of individuals can be cumulatively called *genetic operators*.

Mutation and recombination introduce variability in the population. *Parent selection* is also usually a stochastic process, while biased by the fitness. The population broadens and contracts rhythmically at each generation. First, it widens then the offspring is generated. Then, it shrinks when individuals are discarded. The deterministic pressure usually takes the form of how individuals are chosen for survival from one generation to the next. This step may be called *survivor selection*.

Evolutionary algorithms are local search algorithms since they only explore a defined region of the search space, where the offspring define the concept of neighborhood. For they are based on the trial and error paradigm, they are heuristic algorithms. They are not usually able to mathematically guarantee an optimal solution in a finite time, whereas interesting mathematical properties have been proven over the years.

If the current boundary of evolutionary computation may seem vague, its inception is even hazier. The field does not have a single recognizable origin. Some scholars identify its starting point in 1950, when Alfred Turing pointed out the similarities between learning and natural evolutions [106]. Others pinpoint the inspiring ideas appeared in the end of the decade [44] [16], despite the fact that the lack of computational power significantly impairs their diffusion in the broader scientific community. More commonly, the birth of evolutionary computation is set in the 1960s with the appearance of three independent research lines, namely: *genetic algorithms*, *evolutionary programming*, and *evolution strategies* and. Despite the minor disagreement, the pivotal importance of these researches is unquestionable.

2.2 The classical paradigms

Genetic algorithm is probably the most popular term in evolutionary computation. It is abbreviated as GA, and it is so popular that in the non-specialized literature it is sometimes used to denote any kind of evolutionary algorithm. The fortune of the paradigm is linked to the name of John Holland and his 1975 book [54], but the methodology was used and described over the course of the previous decade by several researchers, including many Holland own students [43] [20] [10]. Genetic algorithms have been proposed as a step in *classifier systems*, a technique also proposed by Holland. However, it may be maintained that they have been exploited more to study the evolution mechanisms itself, rather than solving actual problems. Very simple test benches, as trying to set a number of bits to a specific value, were used to analyze different strategies and schemes. Many variations have been proposed. Thus, is not sensible to describe a canonical genetic algorithm, even in this pioneering epoch.

In a genetic algorithm, the individual, i.e., the evolving entity, is a sequence of bit, and this is probably the only aspect common to all the early implementations. The size of the offspring is usually larger than the size of the original population. Different crossover operators have been proposed. The parents are chosen using a probability distribution based on their fitness. How much a highly fit individual is favored determines the *selective pressure* of the algorithm. After evaluating all new individuals, the population is reduced back to its original size. Several different schemes to determine which individuals survive and which are discarded have been proposed, interestingly all schemes are strictly deterministic. When all parents are discarded, regardless their fitness, the approach is called *generational.* Conversely, if parents and offspring compete for survival regardless their age, the approach is *steady-state.* Any mechanism that preserves the best individuals through generations is called *elitism.*

Evolutionary programming, abbreviated as EP, was proposed by Lawrence J. Fogel in a series of works in the beginning of 1960s [41] [42]. Fogel highlighted that an intelligent behavior requires the ability to forecast changes in the environment, and therefore focused his work on the evolution of predictive capabilities. He chose finite state machines as evolving entities, and the predictive capability measured the ability of an individual to anticipate the next symbol in the input sequence provided to it.

The proposed algorithm considers a set of P automata. Each individual in such population is tested against the current sequence of input symbols, i.e., its environment. Different payoff functions can be used to translate the predictive capability into a single numeric value called fitness. Individuals are ranked according to their fitness. Then, P new automata are added to the population. Each new automaton is created modifying one existing automaton. The type and extent of the mutation is random and follows certain probability distributions. Finally, half of the population is retained and half discarded, thus the size of the population remains constant. These steps are iterated until an automaton is able to predict the actual next symbol, then the symbol is added to the environment and process repeated.

In the basic algorithm, each automaton generates exactly one descendant through a mutation operator. Thus, there is no recombination and no competitive pressure to reproduce. The fitness value is used after the offspring is added to the population to remove half of the individuals, and, unlikely genetic algorithm, survival is not strictly deterministic. How much a highly fit individual is likely to survive in the next generation represent the selective pressure is evolutionary programming. Later, the finite-state machine representation of the genome was abandoned, and the evolutionary programming technique was applied to diverse combinatorial problems.

The third approach is *evolutionary strategies*, ES for short, and was proposed by Hans-Paul Schwefel and Ingo Rechenberg in mid 1960s. It is the more mundane paradigm, being originally developed as an optimization tool to solve practical optimization problems.

In evolutionary strategies, the individual is a set of parameters, usually encoded as numbers, either discrete or continuous. Mutation simply consists in the modification of one parameter, with small alterations being more probable than larger ones. On the other hand, recombination can implement diverse strategies, like copying different parameters from different parents, or averaging them. Remarkably, the very first evolution strategies used a population of exactly one individual, and thus did not implement any crossover operator.

Scholars developed a unique formalism to describe the characteristics of their evolution strategies. The size of the population is commonly denoted with the Greek letter *mu* (μ), and the size of the offspring with the Greek letter *lambda* (λ). When the offspring is added to the current population before choosing which individuals survive in the next generation, the algorithm is denoted as a $(\mu + \lambda)$-ES. In this case, a particularly fit solution may survive through different generations as in steady-state genetic algorithms. Conversely, when the offspring replace the current population before choosing which individuals survive in the next generation, the algorithm is denoted as a (μ, λ)-ES. This approach resembles generational genetic algorithm, and the optimum solution may be discarded during the run. For short, the two approaches are called *plus* and *comma* selection, respectively. And in 2000s literature these two terms can be found in the descriptions completely of different evolutionary algorithms. When comma selection is used, $\mu < \lambda$ must hold. No matter the selection scheme, the size of the offspring is much larger than the size of the population in almost all implementations of evolution strategies.

When recombination is implemented, the number of parents required by the crossover operator is denoted with the Greek letter rho (ρ) and the algorithm written as $(\mu/\rho \overset{+}{,} \lambda)$-ES Indeed, the number of parents is smaller than the number of individuals in the population, i.e., $\rho < \mu$. $(\mu \overset{+}{,} 1)$-ES are sometimes called *steady-state evolutionary strategies.*

Evolution strategies may be nested. That is, instead of generating the offspring using conventional operators, a new evolution strategy may be started. The result of the sub-strategy is used as the offspring of the parent strategy. This scheme can be found referred as *nested evolution strategies*, or *hierarchical evolution strategies*, or *meta evolution strategies.* The inner strategy acts as a tool for local optimizations and commonly has different parameters from the outer one. An algorithm that runs for γ generations a sub-strategy is denoted with $(\mu/\rho \overset{+}{,} (\mu/\rho \overset{+}{,} \lambda)^{\gamma})$-ES. Where γ is also called isolation time. Usually, there is only one level of recursion, although a deeper nesting may be theoretically possible. Such a recursion is rarely used in evolutionary programming or genetic algorithms, although it has been successfully exploited in peculiar approaches, such as [33].

Since evolution strategies are based on mutations, the search for the optimal amplitude of the perturbations kept busy researchers throughout the years. In real-valued search spaces, the mutation is usually implemented as a random perturbation that follows a normal probability distribution centered on the zero. Small mutations are more probable than larger ones, as desired, and the variance may be used as a knob to tweak the average magnitude. Since different problems may have different requirements, researchers proposed to evolve the variance and the parameters simultaneously. Later, because even the same problem may call for different amplitudes in different loci, a dedicated variance has been associated to each parameter. This *variance vector* is modified using a fixed scheme. While the *object parameter vector*, i.e., the values that should be optimized, are modified using the variance vector. Both vectors are then evolved concurrently as parts of a single individual. Extending the idea, the optimal magnitudes of mutation may be correlated. To take into account this phenomenon, modern evolution strategies implement a *covariance matrix*. The idea was presented in mid 1990s and has represented the state of the art in the field for the next decade.

Since all evolutionary algorithms show the capacity to adapt to different problems, they can sensibly be labeled as *adaptive*. An evolutionary algorithm that also adapts the mechanism of its adaptation, i.e., its internal parameters, is called *self adaptive*. Parameters that are self adapted are named *endogenous*, like hormones synthesized within an organism. Self adaptation mechanisms have been routinely exploited both in the evolution strategies and evolutionary programming paradigms, and sometimes used in genetic algorithms.

In the 2000s, evolution strategies are mainly used as a numerical optimization tool for continuous problems. Several implementations, written either in general-purpose programming languages or commercial mathematical toolboxes, like MatLab, are freely available. And they are sometimes exploited by practitioners overlooking their bio-inspired origin. Also evolutionary programming is mostly used for numerical optimization problems. The practical implementations of the two approaches mostly converge, although the scientific communities remain deeply distinct.

Over the years, researchers also broaden the scope of genetic algorithms. They have been used for solving problems whose results are highly structured, like the traveler salesman problem where the solution is a permutation of the nodes in a graph. However, the term genetic algorithm remained strongly linked to the idea of fixed-length bit strings.

If not directly applicable within a different one, the ideas developed by researchers for one paradigm are at least inspiring for the whole community. The various approaches may be too different to directly interbreed, but many key ideas are now shared. Moreover, over the year a great number of minor and hybrid algorithms, not simply classifiable, have been described.

2.3 Genetic programming

The forth and last evolutionary algorithm sketched in this is introduction is *genetic programming*, abbreviated as GP. Whereas μGP shares with it more in its name than in its essence, the approach presented in this book owes a deep debit to its underlying ideas.

Genetic programming was developed by John Koza, who described it after having applied for a patent in 1989. The ambitious goal of the methodology is to create computer programs in a fully automated way, exploiting neo-Darwinism as an optimization tool. The original version was developed in *Lisp*, an interpreted computer language that dates back to the end of the 1950s. The Lisp language has the quite unique ability to handle fragments of code as data, allowing a program to build up its subroutines before evaluating them. Everything in Lisp is a prefix expression, except variables and constants. Genetic programming individuals were lisp programs, thus, they were prefix expressions too. Since the Lisp language is as flexible as inefficient, in the following years, researchers moved to alternative implementations, mostly using compiled language. Indeed, the need for computational power and the endeavor for efficiency have been constant pushes in the genetic programming research since its origin. While in Lisp the difference between an expression and a program was subtle, it became sharper in later implementations. Many algorithms proposed in the literature clearly tackle the former, while are hardly applicable to the latter.

Regardless the language used, in genetic programming individuals are almost always represented internally as trees. In the simplest form, leaves, or terminals, are numbers. Internal nodes encode operations. More complex variations may take into account variables, complex functions and programming structures. The offspring may be generated applying either mutation or recombination. The former is the random modification of the tree. The latter is the exchange of sub-trees between the two parents. Original genetic programming used huge populations, and emphasized recombination, with no, or very little, mutations. In fact, the substitution of a sub-tree is highly disruptive operation and may introduce a significant amount of novelty. Moreover, a large population ensures that all possible symbols are already available in the gene pool. Several mutations have been proposed, like promoting a sub-tree to a new individual, or collapsing a sub-tree to a single terminal node.

The evolutionary programming paradigm attracted many researchers. They were used as test benches for new practical techniques, as well as in theoretical studies. Its challenges stimulated new lines of research. The various topics tackled included: representation of individuals; behavior of selection in huge populations; techniques to avoid the growth of trees; type of initializations. Some of these researches have been inspiring for the development μGP .

Chapter 3

μGP

In this chapter are described the new technologies implemented within μGP , an evolutionary algorithm useful to optimize solution of complex problems, and its actual configuration. This EA tool is completely defined in three blocks: an evolutionary core, a constraints library and an external evaluator. The evolutionary core contains the implementation of evolution-based optimizer; the constraints library serve to the user for a generic structure definition of individual. The external evaluator is a user-written program called by μGP for checking each proposed solution, returning a feedback representing the goodness of the solution in approaching the accounted problem.

3.1 Design Principles

μGP is an evolutionary toolkit designed to be flexible and simply adaptable to very different environment, and applicable to many optimization problems.

The tool implementation follows the idea of maximize the modularity of the code; therefore, it is possible to extend the program reusing some parts, expressly designed to be generic. Operators, for example, are implemented starting from a generic class that define virtual methods that must be implemented in each working genetic operator to be used during optimization process.

The evolutionary core consists on a program to be compiled only once, with the aim of obtaining an executable file runnable in the target machine, on which user will run experiments.

Through this approach, the μGP executable can be reused without modification for different optimization problems: external XML configuration files are requested by the tool for setting evolution parameters and constraints for individuals generation. This is due to the modularity reached by the particular implementation approach: the evolutionary core is the more static part, that not need any modification. The external evaluator is provided by the user, and it is bound by the particular problem to be optimized. The constraints library, defined through the aforementioned XML file, provides informations to the evolutionary core in order to describe the structure of each individual within the population of solution. A solution can be formed by a non-fixed number of fields, each of which can be repeated an arbitrary number of times; values related to these variables argument will be chosen automatically, during evolution. Within constraints configuration file the user can indicate types and ranges of values that can be assigned to each field part of the individual; the user could also define new enumerated types, specifying all the possible values that a variable can assume.

Original application for which μGP was made is the creation of assembly-language programs for testing microprocessor. This precise scope is highlighted by the particular internal representation of candidate solutions, that is engineered with the aim of manage assembly programs, including functions, interrupt handlers and data, and provides also a complete support to conditional branches and labels within instructions.

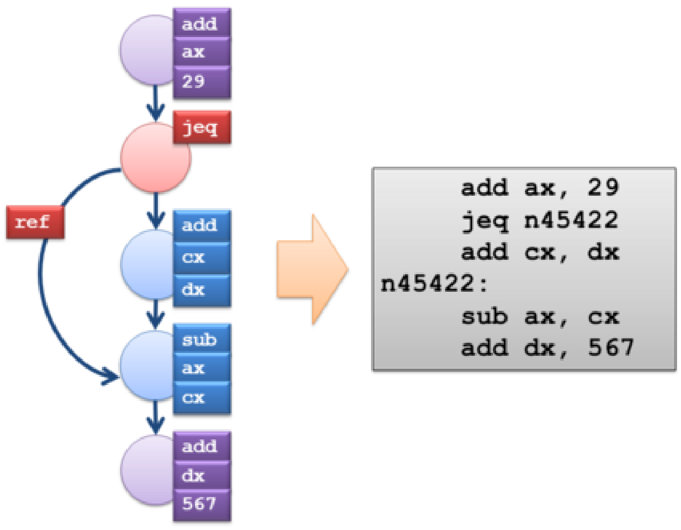
Candidate solutions are forged starting from the description provided by the user through the constraints configuration file; this file, through a xml-oriented structure, define the internal structure that each individual must have to be considered as a valid solution to the approached problem.

Constraints file basically define some macros, that are the building block of individuals. Each macro is formed by two parts: a fixed one that is the expression, and another one that is variable and define the parameters that will be used within the expression. The internal encoding structure of candidate solutions is based on a directed multigraphs, in which each node encode a specific macro defined within the constraints file.

Each node of the graph represents a macro, but the variable parts (parameters) are stored as tags or additional edges. The user defined rules are used also to convert the graph to text file, in order to have a simple representation to be passed to the user-provided evaluator.

The afore described representation, that is by construction the most generic as possible, shows two important ideas on which μGP is based: the former is that no knowledge about the problem being solved is included in μGP itself; the latter is that the user is only required to describe the appearance of the candidate solutions to the faced problem and to provide a program able to evaluate the goodness of each solution proposed by the evolutionary algorithm.

The Figure 3.1 shows the internal encoding of an assembler program, and its respective text representation ready to be sent to the user-provided evaluator program.



```
        add ax, 29
        jeq n45422
        add cx, dx
n45422:
        sub ax, cx
        add dx, 567
```

Figure 3.1. Internal representation of an assembler program

3.2 μGP Evolution Types

Several kinds of evolution can be simulated through the μGP evolutionary tool; the optimization method must be specified by the user at the beginning of the configuration file. This is necessary because different evolution approaches can require different parameters to be provided to correctly set the optimization process.

Evolution types actually performed by μGP are: *standard*, that indicates the default setting for a classical evolution; *multi-objective*, that allows the simultaneous optimization of two or more conflicting objectives; and finally *group evolution*, a new approach on the evolutionary algorithms panorama that permit the cooperative optimizations of sub-populations of solutions.

To correctly setting the evolution process, there are several parameters that can be tuned by the user. Values to be assigned to these parameters can be optimized trough a trial-and-error approach; normally by using correct parameters the optimized solution can be reached rapidly, but final best solution should be comparable.

Parameters characterizing an optimization process are:

- μ: is the size of the population; at the beginning and at the end of each generation, there will be exactly μ individuals;
- λ: is the number of genetic operators that will be applied at each step. Genetic operators can creates more than one individual, so the offspring size is usually bigger than λ.

In μGP , however, are present several other parameters that should be set for adjusting the evolution related to the faced optimization problem:

- ν: the size of the initial randomly-created population. After the first generation, as number of individuals, will be complied the value specified as the μ parameter. This option permits to start the evolution creating a larger number of random solutions to raise the search space explored and to begin the evolution from a better starting point.
- σ: through this parameter is possible to regulate the strength of genetic operators. It is a self-adapting value that regulate the difference from exploration and exploitation phases. At the beginning of the evolutionary process his value is higher and genetic operators will create individuals that differ much from parents, in order to make macroscopic changes and explore quickly the majority of the search space. The σ value will be lower at the end of the evolution, when exploitation is better, regulating genetic operators for creating individuals more similar from their parents, thus refining their genetic heritage.
- *inertia*: this parameter represents the resistance of the system to the self-adapting push towards new values. It is used to tune the self-adapting mechanism of μGP , in order to set the velocity on which the algorithm itself modifies its behavior as the evolution goes on, trying to always have the best performance given the current situation.

  In μGP , therefore, two different types of selection of parent individuals are available:

- *tournamentWithFitnessHole*: a classical tournament selection, with the further possibility of comparing individuals on their delta entropy instead of the fitness value(s). The dimension of the tournament is managed by the auto-adaptive τ value, that set the number of individuals involved. In addition to this mechanism, as described in [79], with a probability equal to the *fitness hole*, the tool does not select individuals based on their fitness but on a different criterion. Currently, the alternative criterion is the contribution of the individual to the total entropy.

- *ranking*: with this other type of parent selection, each individual has a probability to be chosen for reproduction based on its position in the population, ordered by fitness value(s). *Inertia* parameter is used to auto-adapt this value depending on the phase of the evolution.

3.2.1 Standard Evolution

The *standard evolution* is the most widespread use of genetic algorithm.

This approach places the basis of optimization through an evolutionary algorithm. It is a population based approach, in which the EA is required to optimize solutions with the aim to enhanced them basing their goodness on numerical value(s), called *fitness*. The fitness is indicating the goodness of each solution to solve the problem addressed by the optimization, and it is a value assigned by the external evaluator. In the case in which several numerical values are needed to evaluate each individual, the μGP will consider them according to an importance order, reflecting the same order in which values are returned by the evaluator.

At the beginning of the optimization process, the EA first creates a population containing random solutions. Then, at each generation, genetic operators are applied to selected individuals in order to create new ones: offspring is then evaluate in order to verify their goodness to solve the addressed problem. The process goes on with the aim of exploring the whole search space and to find the global optimal solution.

3.2.2 Multi-Objective Evolution

The *Multi-Objective evaluation* is an optimization process slightly different from the previous one. It evolves a population of individuals basing their goodness to solve the problem on two or more values that, differently from the standard evolution, are all taken into account with the same importance.

Due to conflicts that should be present among fitness values, at the end of the evolution will not be possible to select a single best solution; this is an expected behavior of this kind of optimization: a single individual cannot represent the best solution for all the conflicting fitnesses values involved withint the optimization process.

At the end of the evolution, μGP will indicate a set of solutions forming a Pareto front. Individuals present on the Pareto front will be the best ones balancing the optimization taking into account all the defined fitnesses.

3.2.3 Group Evolution

The approach faced through this particular kind of evolution, is based on recent new ideas developed within the evolutionary computation field. This evolutionary algorithm technology is based on the evolution of groups of solutions, in order to obtain a set of individuals able to solve together the faced problem in an optimal way. Each individual of the population could be considered as a complete solution to the problem but, considering the typical evolution process implemented, the algorithm was forged to be able to obtain from each individual a partial solution. The final best set should group the partial solutions fitting together as a team in the best way to solve the optimization problem.

The approach being discussed was already applied to two optimization problems: the former is more trivial and can be mainly considered as an academic experimental activity, and regards the placement of a set of lamps to illuminate a certain area [105]; the latter proposes a method for the automatic generation of SBST on-line test programs for embedded RISC processor [25].

In [25] authors shows, through preliminary experiments based on the aforementioned new technology, that it performs better than other techniques commonly applied in the CAD field; in particular, making a comparison with normal evolutionary technique, this new approach shows the capability to reach an optimal solution composed by a set of homogeneous elements. On the contrary, this is not possible to be obtained by several run of normal approaches, without an a-priori knowledge both of the problem and of the role each individual should play in the global solution. These preliminary experiments were addressed to show that this algorithm, with respect to the objective of creating a cooperative solution formed by several individuals sub-optimized to reach a goal, performs better than other techniques typically used within CAD environment, such as Multi-Run testing.

In [25], as aforementioned, is described an approach for a real CAD problem about automatic test programs generation for a microprocessor. As expected, the optimization process was able to obtain a group containing a test set of sub-optimal test programs that achieves about 91% fault coverage against stuck-at faults on the forwarding unit of the considered RISC microprocessor.

3.3 Evaluator

The μGP evolutionary algorithm tool is designed to be the as flexible as possible; following this purpose, it does not provide any internal mechanism to evaluate generated individuals.

Through this approach, the advantages are twofold:

- The μGP executable need to be compiled only once: no modifies are required to apply the EA to different problems. The only adjustment required is to set constraints file in order to allow the tool to generate solutions coherent with the addressed problem and, if necessary, change parameter settings of the EA.
- The evaluator, being an external part, can be implemented with whatever technology or programming language. It is enough to provide, indicating it on the main configuration file, the name of the executable or script file that μGP should invocate whenever it is necessary to evaluate a solution.

μGP uses the constraints user-defined file as guidelines to create new individuals. Roughly speaking, in the constraints file is described the precise structure that each solution should have, and the definitions of values that each section can assume. Despite this strictly constraints definition, occasionally it could happen that a generated individual does not comply with the requirements needed by the evaluator. In this cases, the evaluator should be designed in such a way to return to μGP the zero value, as to refer that the current solution to be checked is unfeasible. This step is very important, because this value will be used by the mechanism that rewards operators in order to regulate their usage during evolution; this mechanism will be described in detail in Section 3.4.

Due to the particular design of evolutionary algorithm, time is flowing step by step, defined by generations. This means that, during the period in which solutions are evaluated, the time is standstill. This important characteristics can be used to evaluate several individuals of the offspring at the same time, running more than one instances of the evaluator program in parallel. The number of simultaneous evaluations can be defined through the parameter *concurrentEvaluations* within the configuration file, in the same section in which is defined the evaluator program that should be used by the μGP .

3.3.1 Cache

During the evolution, and depending on the dimension of the search space expressed through the definition of constraints, it is quite frequent that application of genetic operators bring to the creation of a new solution identical to one generated and already evaluated. Since the evaluation of solutions is the time consuming part of the optimization, it is clear that is preferable to avoid unnecessary evaluation processes. To avoid the re-evaluation of solutions of which are already know the goodnesses to solve the problem, an internal cache mechanism was implemented.

The caching method is based on an internal memory map, containing three values for each entry: the *hash signature* of the individual, the complete description of the solution converted as a string, and his fitness value obtained by calling the external evaluator: each recent solution optimized by the evolutionary tool is cached using this mechanism.

As default setting, the cache is enabled with a maximum dimension fixed to 10,000 entries; when the maximum capacity is reached, a Least Recently Used (LRU) algorithm is used, that discards the least recently used items first. The *cacheSize* parameter within the setting configuration file allows the user to change the maximum dimension of the cache. If the optimized solutions are evaluated by an environment that change in time, so an individual could have different fitness values, the cache system must be deactivated. The deactivation means that cache will be flushed at the end of each evolutionary step (i.e. generation), but keeping this mechanism within the same step in which the evaluating system should be unchanged.

Through the cache it is possible to reduce the whole evaluating time, shrinking the duration of a complete optimization process. This caching mechanism is useful in particular when the Group Evolution (described in Section 3.2.3) is used: group population is formed by individuals grouped in subsets of the whole population; due to this, groups can share one or several individuals among themselves. μGP , to perform correctly the optimization, will ask to the external evaluator both fitness of the whole group, both fitnesses of each individual part of it;

3.4 Operators' Activation Probability

In this section is described the Dynamic Multi-Armed bandit (DMAB) mechanism, used within the μGP with the aim to select different operators to be applied during evolution, balancing choices on rewards obtained by their previous applications.

3.4.1 The Multi-Armed Bandit Framework

The Exploration VS Exploitation dilemma has been intensively studied in game theory, especially in the context of the MAB framework [64][8]. Let's consider an hypothetical slot machine with N arms (the bandit); at time t_k, the i-th arm, when selected, gets a reward 1 with probability p_i, and 0 otherwise. A solution to the MAB problem is a decision making algorithm that selects an arm at every time step, with the goal of maximizing the cumulative reward gathered during the process. The widely studied Upper Confidence Bound (UCB) algorithm [8], proves that to maximize the cumulative reward with optimal convergence rate, the player must select at each time step t the arm i that maximizes the following quantity:

$$\hat{p}_{i,t} + C \cdot \sqrt{\frac{\log \sum_k n_{k,t}}{n_{i,t}}} \tag{3.1}$$

where $n_{i,t}$ is the number of times the i-th arm has been activated from the beginning of the process to time t; while $\hat{p}_{i,t}$ denotes the average empirical reward received from arm i. C is a scaling factor that controls the trade-off between exploration, favored by the right term of the equation; and exploitation, favored by the left part of the equation, that pushes for the option with the best average empirical reward.

3.4.2 DMAB and Operators Selection in EA

The MAB problem can be intuitively applied to operator selection in EAs: every arm of the bandit can be mapped to one operator. Using the UCB metric, the algorithm keeps exploring all arms, while favoring good operators. However, contrary to the theoretical bandit, an evolutionary run is a dynamic environment, in which the standard MAB algorithm would require a considerable amount of time to detect that the best operator has changed. To solve this issue, [32] proposed to use a statistical change detection test, creating the Dynamic MAB (DMAB) algorithm. Specifically, the Page-Hinkley test [53] is used to detect whether the empirical rewards collected for the best current operator undergo an abrupt change. In the DMAB, if the PH test is triggered, suggesting that the current best operator is no longer the best one, the MAB algorithm is restarted.

An overview of the most successful DMAB based mechanisms can be found in [39]. Further works build on the DMAB algorithm most notably by comparing various credit assignment mechanisms and measuring how well they complement the DMAB selection scheme. In [68], for example, the authors propose to combine the DMAB with a credit assignment scheme called *Compass*, that evaluates the performance of operators by considering not only the fitness improvements from parent to offspring, but also the way they modify the diversity of the population, and their execution time.

3.4.3 μGP Approach

We propose a DMAB selection strategy that not only allows operators to sporadically fail without being completely removed from the process, but is also able to consecutively apply several operators without needing a performance feedback after each application. For our approach, we consider the following EA structure:

```
operators ← {available operators and their MAB state};
policy.init(operators);
parents ← {some random individuals};
until reached a stop condition do
    offspring ← [];
    applications ← [];
    policy.before_selections(operators);
    until $\lambda$ successful operator applications do
        op ← policy.select(operators);
        children ← op.apply(parents);
        if children = $\emptyset$ then
            policy.failure(op);
        else
            policy.success(op);
            applications.append((op, children));
            offspring.append(children);
    evaluate(offspring);
    policy.reward(parents, offspring, applications);
    parents ← selection(parents, offspring);
```

Algorithm 1: Outline of our target EA

This general structure is shared by different EAs, and now it is present also in the μGP . In this type of architecture the evaluation phase can be easily parallelized, and operators can occasionally fail without being removed from the selection process.

During the current generation, the only information that the policy can gather is whether the selected operator actually produced children, through the functions `policy.success()` and `policy.failure()`. After the evaluation phase, the policy can access more information: the fitness of the newly produced offspring makes tournaments possible.

3.4.4 Notations

Each operator is considered as an arm of a MAB and is associated to several statistics. First, we count the number of successful applications awaiting a reward, in *op.pending*. This statistic is reset after each each generation, when the operator actually receives the rewards corresponding to its applications. Then, the fields *op.enabled* and *op.tokens* account for this operator's applicability. An operator is enabled when it can produce new offspring, and will be selected only if it has a positive number of tokens. Finally, we maintain a short history of the last obtained rewards in *op.window* and the classical DMAB statistics, in *op.n*, *op.$\hat{p}$*, *op.m* and *op.n* [68]. Algorithm 2 covers the initialization of these variables.

```
function policy.init(operators) is
    foreach op in operators do
        // Intra generational call count
        op.pending ← 0;
        // Failure handling statistics
        op.enabled ← False;
        op.tokens ← 3;
        // Compass-like window[68]
        op.window ← queue of size $\tau$;
        // DMAB statistics[68]
        op.$n$ ← 0;
        op.$\hat{p}$ ← 0;
        op.$m$ ← 0;
        op.$M$ ← 0;
```

Algorithm 2: Initialization of the operator statistics

3.4.5 Operator Failures

We define a “failure” as the application of an operator that does not result in any new usable solution, or valid individual. This can happen for two reasons: either the operator is not applicable to the genome of a candidate solution in the current problem, and will thus always fail; or its execution can sporadically fail, for example based on the structure of the selected parents. We exclude inapplicable operators by considering that all operators are disabled until they prove their usefulness by building a new valid solution. Until that happens, failing operators are called periodically to check whether some emergent characteristic of the population enables them to work in a later stage of evolution.

Once an operator is enabled for good, however, its failed executions are disregarded: they do not count against the λ required executions per generation, and the same operator (or a different one, when operator selection is stochastic) is just re-applied. As a consequence, when an unforeseen edge case is encountered during its execution, an operator can just fail without any penalty: this also makes it easier to add new operators to a framework, as foreseeing their effect on all possible genomes is not necessary.

Performance problems can arise if an operator is computationally intensive, builds very good solutions, but fails most of the time. Such an operator might get called repeatedly and use a considerable amount of CPU time for the production of few viable children. To avoid this situation, we use failure tokens, that is, a maximum number of failed calls allowed per generation. Our failure handling mechanism is implemented as a "filter" before the real selection strategy, as shown in Algorihtm 3.

3.4.6 Credit Assignment

The state of the art for credit assignment is probably the Compass [69][68] method. In short, Compass associates the application of an operator with the variation of two characteristics of the population on which it operates: ΔD (mean diversity) and ΔQ (mean fitness). Its execution time T is also stored. These three values, averaged over a window of the last τ applications of the operator, are used to compute a reward. The meta-parameter Θ defines a compromise between the two first criteria ΔD and ΔQ: according to the authors, it affects the *Exploration-vs-Exploitation* (EvE) orientation of the algorithm, and this compromise value is divided by the execution time of the operator to produce its reward.

While extremely ingenious, this technique cannot be translated directly into all EAs, and μGP in particular, for several reasons. First of all, μGP does not use the Θ angle: the tool already features a self-adapted σ parameter that controls the strength of mutation operators and effectively regulates the amount of exploration. As for diversity preservation and promotion, μGP provides different mechanisms such as fitness sharing, fitness scaling, delta entropy, and fitness holes, which encapsulate the diversity-versus-quality problem into the comparison of individuals, either during selection of parents (fitness hole) or selection of survivors (scaled fitness) [79, 34, 88].

Moreover, μGP does not make any assumption about the regularity of the fitness function, which deprives a difference between two fitness values of any meaning beyond its sign. The ΔQ criteria is thus not available in μGP .

We therefore replace the $(\Delta Q, \Delta D, \Theta)$ triple with a single measure defined as such: we organize a tournament between all the parents and all the freshly evaluated offspring, and reward the offspring proportionally to their rank. This procedure provides a comparison of the new offspring's fitness with respect to the fitness of their parents, and a juxtaposition between all operators, finer for higher values of μ,λ.

As many other EAs, μGP is designed to target problems where the evaluation cost is predominant over the time spent in the evolutionary loop: in this context, operator execution times are often irrelevant. The only metric that could be of interest is the individual evaluation time. However, the architecture has been designed to impose the lowest possible coupling between the evaluator and the evolutionary algorithm. For this reason, individual evaluation times are not reported to μGP , and anyway are not required to be meaningful or correlated with the generating operator. Thus, we do not consider the execution time T.

```
function policy.before_selections(operators) is
    foreach $op \in operators$ do
        op.pending $\leftarrow 0$;
        if $op.enabled$ then
            op.tokens $\leftarrow \lambda$;
        else
            if $current\ generation \mod 10 = 0$ then
                op.tokens $\leftarrow 1$;

function policy.select(operators) is
    foreach op in operators where ¬ op.enabled do
        if op.tokens ¿ 0 then
            return op;
    enabled $\leftarrow \{op \in operators \mid op.enabled\}$;
    if $enabled = \emptyset$ then
        redistribute 3 tokens to all operators;
        return any operator;
    selectable $\leftarrow \{op \in enabled \mid op.tokens > 0\}$;
    if $selectable = \emptyset$ then
        redistribute $\lambda$ tokens to all enabled operators;
        selectable $\leftarrow$ enabled;
    if $\exists o \in selectable \mid o.n = 0$ then
        return $\operatorname{argmin}_{o \in selectable} o.n + o.pending$;
    return policy.real_select(selectable);

function policy.failure(operator) is
    operator.tokens $\leftarrow$ operator.tokens $- 1$;

function policy.success(operator) is
    operator.pending $\leftarrow$ operator.pending $+ 1$;
    if $\neg operator.enabled$ then
        operator.enabled $\leftarrow True$;
        operator.tokens $\leftarrow \lambda$;
```

Algorithm 3: Execute at least once all operators before using DMAB and limit failure rate

One feature of interest remains from Compass: the time window of the last τ

values of $(\Delta Q, \Delta D, \Theta)$. Our selection scheme will indeed use a window of past fitness improvements to distribute rewards. However, the original version of Compass[69] and the more recent work to pair it with DMAB [68] differ: while the former computes a mean, the latter argues that using extreme values (i.e. the max) yields better results, borrowing the idea from Whitacre et al.[109]. We choose to compromise between the two options by computing a weighted sum of the values, assigning higher weights to the highest values. The compromise is tuned by the discount $\in [0, 1]$ parameter. A value of 0 gives the maximum, a value of 1 the mean.

This leads us to Algorithm 4 for reward distribution.

```
function policy.reward(parents, offspring, applications) is
    tournament ← parents ∪ offspring;
    sort tournament by increasing fitness;
    foreach (op, children) in applications do
        improvement ← $\max_{\text{children}} \frac{\text{tournament.rank(child)}}{\text{tournament.size()}}$;
        op.window.append(improvement);
        W ← sorted op.window in decreasing order;
        discount ← 0.5;
        $r \leftarrow \frac{\sum_{i=0}^{\text{W.size()}} \text{W}_i \text{discount}^i}{\sum_{i=0}^{\text{W.size()}} \text{discount}^i}$;
        // DMAB algorithm from DaCosta[32]
        $\text{op}.\hat{p} \leftarrow \frac{1}{\text{op}.n+1}(\text{op}.n\ \text{op}.\hat{p} + r)$;
        $\text{op}.n \leftarrow \text{op}.n + 1$;
        $\text{op}.m \leftarrow \text{op}.m + (\text{op}.\hat{p} - r + \delta)$;
        $\text{op}.M \leftarrow \max(\text{op}.M, \text{op}.m)$;
        if $op.M - op.m > \gamma$ then
            reset all MAB statistics of all operators;
```

Algorithm 4: Credit assignment

3.4.7 Operator Selection

The limitation we found to the DMAB selection scheme lies in the exclusive dependence of an operator's selection on its obtained rewards: in our case it means that, during one generation, DMAB will select the same operator λ times. Experimental results will show that this strategy is suboptimal with higher values of λ, because when the DMAB takes the decision to exploit an operator it does so λ times in a row without any exploration, and conversely, when exploration is needed for an operator, it is called λ times even if it's the worst available.

We propose to mitigate this “all or nothing” intra-generational effect in the following way: we consider that operator applications during the generation receive immediately a fake reward equal to their current estimated reward. Put another way, we simply increment the number of successful executions n for each successful application while maintaining the three other MAB statistics ($\hat{r}$, m, M) to their original values. This makes the DMAB scores vary enough during the generation to allow exploration to happen. After the evaluation of all candidates, the other DMAB statistics are updated as usual using the actual rewards. We call this strategy PDMAB (Parallelized DMAB).

```
function policy.real_select(operators) is
    total_n ← $\sum_{o \in \text{operators}} o.n + o.pending$;
    return $\underset{o \in operators}{\text{argmax}}\ o.\hat{p} + C\sqrt{\frac{\log total\_n}{o.n + o.pending}}$
```

Algorithm 5: PDMAB strategy

3.5 A Novel Distance Metric

Defining a distance measure over the individuals in the population of an Evolutionary Algorithm can be exploited for several applications, ranging from diversity preservation to balancing exploration and exploitation. When individuals are encoded as strings of bits or sets of real values, computing the distance between any two can be a straightforward process; when individuals are represented as trees or linear graphs, however, quite often the user must resort to phenotype-level problem-specific distance metrics. This work presents a generic genotype-level distance metric for Linear Genetic Programming: the information contained by an individual is represented as a set of symbols, using n-grams to capture significant recurring structures inside the genome. The difference in information between two individuals is evaluated resorting to a symmetric difference. Experimental evaluations show that the proposed metric has a strong correlation with phenotype-level problem-specific distance measures in two problems where individuals represent string of bits and Assembly-language programs, respectively.

3.5.1 Introduction

Defining a *distance metric* in an Evolutionary Algorithm (EA) is both theoretically sound and practically challenging – and ultimately useful. Being able to quantify the similarity of two individuals can be used to promote diversity inside the population’s gene pool, to avoid the over-exploitation of niches in the fitness landscape, to balance exploration and exploitation, and ultimately to ease the premature convergence problem. Not surprisingly, the topic has been actively investigated by the evolutionary community for many years.

From the theoretical point of view, two different aspects must be examined when a distance is defined: the level at which it is calculated; and the purpose for calculating it. On the other hand, for the practitioner the complexity involved in the calculation is the key point.

The level at which a distance is defined may be: genotype, phenotype, or fitness. The first and the last are probably the most easily definable: the genotype corresponds to the internal representation of the candidate solution; the fitness is ultimately the number, or numbers, returned by its evaluation. In biology, the phenotype is the sum of all the observable characteristics of an organism that result from the interaction of its genotype with the environment. It is hard to translate the concept in Evolutionary Computation since the environment is missing, being indirectly defined by its effects through the fitness function. Yet, in several classical problems – where an individual is a fixed-length bit string, for instance – the need to distinguish between genotype and phenotype is reduced. As a consequence, several works assimilate the fitness to the phenotype.

In many other cases identifying phenotype and fitness is not an option. The fitness is a synthetic information, and may not be able to convey the necessary data to separate individuals. Even in the simplistic one-max problem two solutions may have the same fitness without sharing a single gene (e.g., "0011" and "1100"). Moreover, the very same solution can be encoded in different ways. If the individual is the movement of a robot, for instance, a single 90° turn could also be represented as two consecutive 45° ones. More generally, whenever the genotype cannot be evaluated directly by the fitness function, but needs to be transformed into something else, fitness and genotype should be distinguished. In such scenarios, the phenotype could be easily defined as the "something else" in which the genotype needs to be transformed into.

The final goal for measuring the distance between individuals plays an important role. If the distance metric is used to avoid that a region of the search space becomes overly populous, then it should be defined at the level of phenotype. However, phenotype-level distances are often difficult to define or practically impossible to calculate. Remarkably, NSGA-II, the widely used multi-objective evolutionary optimizer, adopts a sharp and computationally efficient mechanism called *crowding distance* to scatter individuals [35]. Here, the crowding distance may rely exclusively on information from the fitness because the genotypes are fixed-length arrays of real numbers, requiring no transformation; and the fitness is composed of several different values, reducing the loss of information.

Conversely, if the distance metric is used to promote diversity in the gene pool, balancing exploration and exploitation, it could be based on the genotype. For example, in [70] solutions are encoded as fixed-length bit strings and a metric based on the hamming distance is used to assess the global diversity inside the population. When the phenotypes are sets of real values of fixed size, computing the distance between them is relatively straightforward, albeit not trivial [30]. However, phenotypes in Genetic Programming (GP) [63], Linear Genetic Programming (LGP) [19] and other complex EAs pose a harder challenge: calculating the similarity between two binary trees, linear graphs, or generic compound structures is an open problem.

This work proposes a new distance metric easily usable in different types of LGPs. The distance is calculated quite efficiently at the level of genotype, yet it is able to convey a considerable amount of information about the individual. Thus, it may be used to reduce crowding in place of a phenotype-level distance. The proposed approach computes the *symmetric difference* [2] between the global information contained in two individuals; while the global information itself is evaluated resorting to the concept of n-grams [100].

Experimental results demonstrate that the proposed distance is highly correlated with other phenotype-level problem-specific distance measures, both where individuals are string of bits and Assembly language programs. Further experiments show that exploiting the proposed metric to perform fitness sharing in a sample problem produces results comparable to using a phenotype-level metric.

Linear Genetic Programming

LGP is a variant of GP that evolves computer programs as sequences of instructions. It was introduced by Markus Brameier and Wolfgang Banzhaf between the late 90s and the early 2000s [11] [19], after the seminal work of Friedberg [44]. A traditional, *Koza-style* GP encodes individuals as trees. Such tree GPs – or TGPs, as they are sometimes called – are commonly used to evolve mathematical expressions: leaves correspond to *terminals*, such as input values, constants or variables; inner nodes represent *functions*. Quite differently, LGP evolves a simplified list structure that represents a sequence of instructions in an imperative programming language. The resulting individuals represent real programs, although in a simplified language, that can grow to a significant complexity. Since their appearance, LGPs have been widely used to solve both practical problems and perform theoretical studies.

In LGP the difference between genotype and phenotype becomes fully apparent. The *genotype* is the internal, list-based representation; the *phenotype* is the actual program resulting from the interpretation of the genotype; the *fitness* is the final result of the evaluation of the program (Figure 3.2).

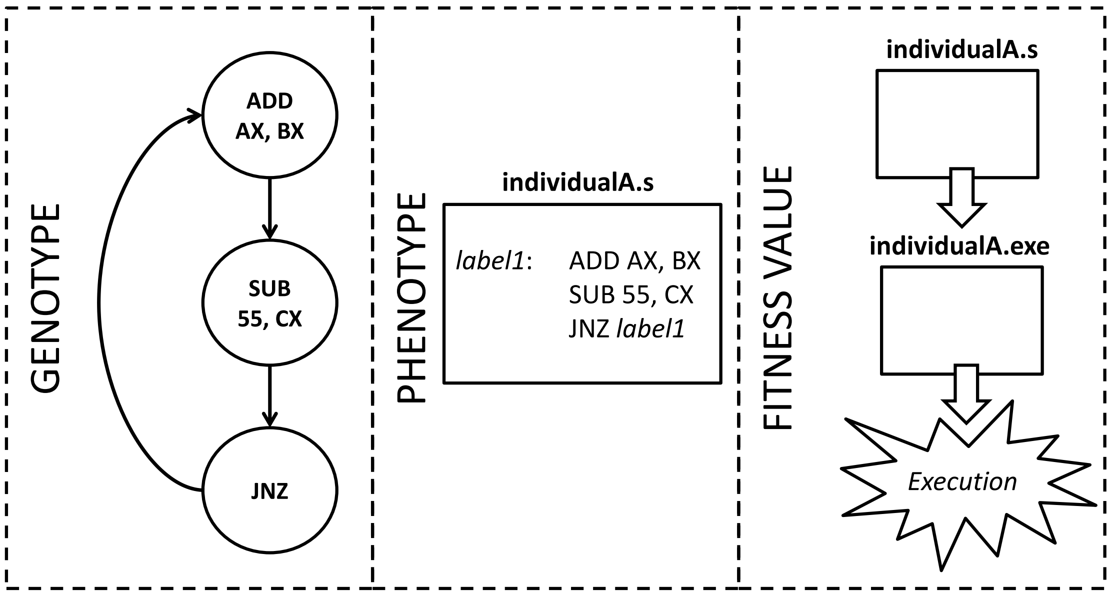


Figure 3.2. Distinction between genotype, phenotype and fitness value in an example with LGP used for Assembly language generation.

Symmetric Difference

In set theory, the symmetric difference [2] of two sets A and B is defined as

$$A \triangle B = A \cup B - A \cap B \tag{3.2}$$

In practice, the symmetric difference contains all elements which are in either of the sets and not in their intersection. The Venn diagram of the symmetric difference is reported in Figure 3.3.

Considering the set as the *information* carried by an individual, the symmetric difference appears a plausible formalization of the intuitive idea of distance: when two sets are almost completely overlapping, their symmetric difference is very small; when they are completely separated, it is big.

Moreover, the symmetric difference exhibits useful properties for a distance: it is commutative ($A \triangle B = B \triangle A$); and the empty set is neutral ($A \triangle \emptyset = A$ and $A \triangle A = \emptyset$). The symmetric distance is also associative, but this fact is negligible in this application.

Fitness Sharing

When a reliable distance metric is defined, one useful application is to exploit it for *fitness sharing*, one of many methods to enforce diversity inside the population of an EA [91] [85].

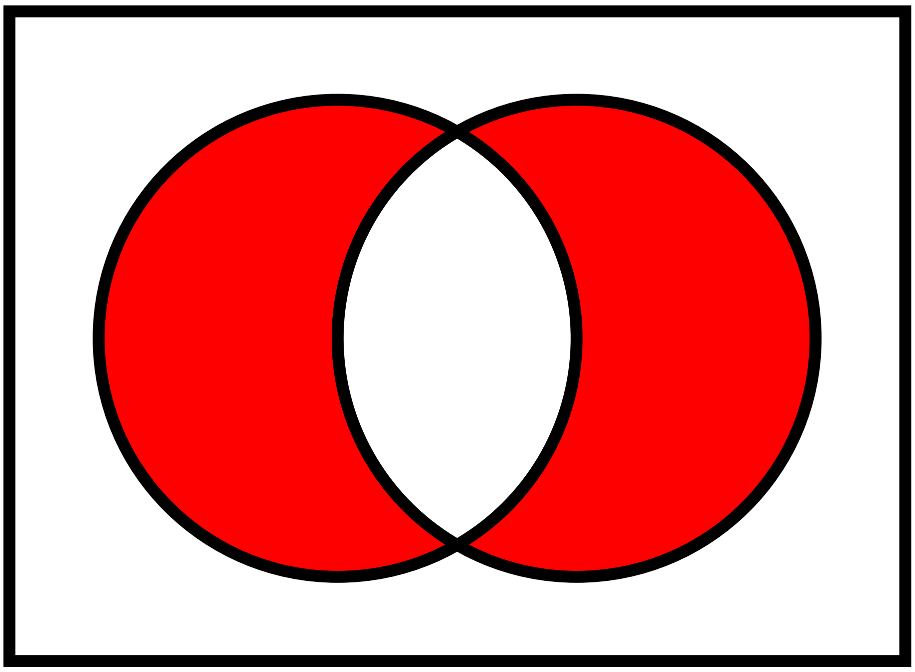


Figure 3.3. Venn diagram of the symmetric difference. The area corresponding to $A \triangle B$ is depicted in grey.

The general idea of fitness sharing is to artificially decrease the fitness of individuals in crowded areas of the search space. The fitness f_i of an individual I_i is modified in a fitness $f'_i = f/m_i$, where m_i is dependent upon the number of individuals in a given radius σ_s from individual I_i, and their distance from the individual itself. In particular,

$$m_i = \sum_{j=0}^{N} sh(I_i, I_j) \tag{3.3}$$

where N is the number of individuals in the population, and $sh(I_i, I_j)$ is defined as

$$sh(I_i, I_j) = \begin{cases} 1 - (\frac{d(I_i,I_j)}{\sigma_s})^{\alpha} & d(I_i, I_j) < \sigma_s \\ 0 & d(I_i, I_j) \geq \sigma_s \end{cases} \tag{3.4}$$

where σ_s is once again a user-defined radius, and $d(I_i, I_j)$ is a distance measure applicable to the individuals' representation. α is a constant parameter that regulates the shape of the sharing function. In many practical cases $\alpha = 1$, with the resulting sharing function referred to as the triangular sharing function [49].

3.5.2 μGP Approach

In LGP, Shannon entropy can be effectively used as a metric to assess the diversity in a population at a given generation [28]. The entire population is considered a message, and each allele appearing in an individual is a symbol: entropy is then computed on the basis of the number of different symbols and their occurrences.

In a preliminary work [99], a variant of this approach is sketched. Instead of considering just the alleles of each gene, their disposition inside the individual is also taken into account. A symbol is no longer considered equivalent to a single allele, but to the allele and its position inside the individual instead. Following the idea that recurring structures might possess meaning, *n-grams* of nodes are also regarded as symbols. An n-gram is a group of n items from a longer sequence. For example, `a b`, `b c` and `c d` are all 2-grams from the sequence `a b c d`, while `a b c` and `b c d` are 3-grams. Very often n-grams are used for the purpose of modelling the statistical properties of sequences, particularly natural language [100].

Building on the same principles, a generic genotypic *Universal Information Distance* (UID) for individuals in LGP is proposed. Considering two individuals I_i and I_j, the UID is defined as

$$UID(I_i, I_j) = |S(I_i) \triangle S(I_j)| \tag{3.5}$$

where $S(I)$ represents the set of symbols in individual I, $\triangle$ is the symmetric difference as defined in Equation 3.2, and the operator $|S|$ denotes the cardinality of set S.

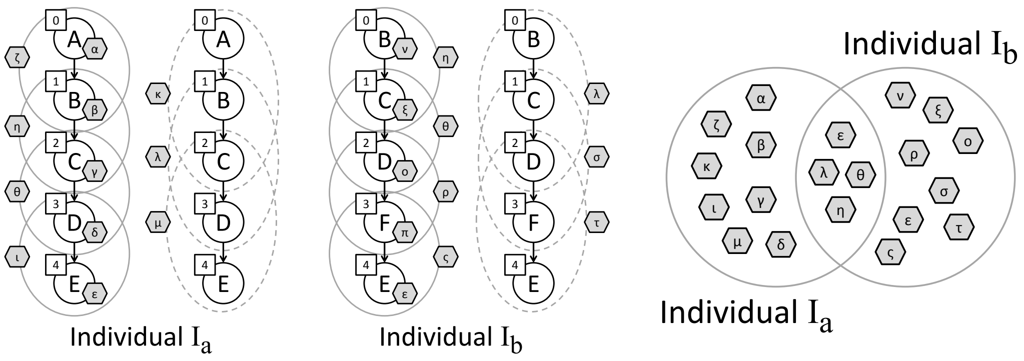


Figure 3.4. Example of symbols computed for alleles and (2,3)-grams for two individuals. Symbols are represented as Greek letters inside hexagons, alleles as Roman letters inside circles, while their position in the individual is reported in a square. The symbols common to the two individuals are ϵ (corresponding to allele E in position 4), η (2-gram $B-C$), θ (2-gram $C-D$) and λ (3-gram $B-C-D$). The UID between the two individuals is thus $|S(A) \triangle S(B)| = |S(A) \cup S(B) - S(A) \cap S(B)| = 16$

In other words, the UID between two individuals is the number of distinct symbols they do not have in common. Intuitively, when two individuals share many common symbols, the UID will be small; on the contrary, if they have no symbols in common, their UID will be high. An example is reported in Figure 3.4.

When used in practice, symbols for each individual are computed resorting to a hash function of the n-grams and alleles. It is interesting to notice how the proposed UID, that acts at genotype level and is quite straightforward to compute, could provide the same information of more computationally intensive distance metrics that are evaluated at phenotype level: thus, UID could be used for fitness sharing, delivering the same results as problem-specific metrics.

3.5.3 Experimental Evaluation

The correlation between the proposed UID and two phenotypic distance metrics is examined, in two problems where individuals are encoded as strings of bits, and as Assembly language programs, respectively. Experiments with an evolutionary toolkit that supports LGP [88] are then performed for the two problems, and the effectiveness of UID for fitness sharing is compared to the previously considered phenotypic distance metrics.

In the all the experiments, the computation of UID is limited to n-grams of order 2 and 3, as a trade-off between computational efficiency and thoroughness of the approach. Symbols are computed resorting to the DJB[1] hash function.

Considered Problems

The proposed approach is tested on two benchmarks: NK-landscapes, a NP-complete problem where individuals are represented a strings of bits, and a simple Assembly-language generation task.

NK-Landscapes

In the NK-landscapes problem [62], the individual is a string of bits of fixed length: both the overall size of the fitness landscape and the number of its local optima can be adjusted by tuning the two parameters, N and K. Generally speaking, values of K close to N create more irregular landscapes. Albeit simple, this benchmark is widely studied in the optimization community, because it is proven to be NP-complete [108]. In the following experiments, values of N and K are very close, in order to obtain a fairly rugged fitness landscape.

[1]http://cr.yp.to/djb.html

Assembly Language Generation

This second set of experiments targets a simple Assembly language generation problem: the fitness of a program is the number of bits set to 1 in registry *%eax* at the end of its execution.

During the Assembly-generation problem, the minimum length of the variable part of a program is set to 1 instruction, the while the maximum length is set to 1500 instructions. For the initial population, individuals are generated in order to possess an average of 70 instructions, with a large standard deviation of 60 in order to efficiently sample the search space. Table 3.1 recapitulates the possible genes (instructions) appearing in the individuals.

Gene	**Parameters**	**Prob.**
<ins><sreg>, <dreg>	**ins**={addl, subl, movl, andl, orl, xorl, compl}, **sreg**={%eax, %ebx, %ecx, %edx}, **dreg**={%eax, %ebx, %ecx, %edx}	0.33
<ins><scon>,<dreg>	**ins**={addl, subl, movl, andl, orl, xorl, compl}, **scon**={integer in (0,255)}, **dreg**={%eax, %ebx, %ecx, %edx}	0.33
<ins><reg>	**ins**={incl, decl, notl}, **reg**={%eax, %ebx, %ecx, %edx}	0.33

Table 3.1: Possible genes appearing inside the individuals during the Assembly generation problem. For each gene, all variables and corresponding values are listed, as well as the probability of occurrence.

The fitness function used for this experiment is based on both the result of a candidate program's execution and the length of its code, and it is defined as

$$f(I) = 10^4 \cdot (\sum_{i=0}^{N=31} \%eax[i]) + max(0,10^4 - length(I)) \tag{3.6}$$

where $\%eax[i]$ is the value of the ith bit of register $\%eax$, while $length(I)$ represents the number of instructions of candidate program I. Thus, the most important objective is to set to 1 bits in register $\%eax$, while a small bonus is assigned to individuals that perform the task with a moderate number of instructions.

Correlation

An important result that it is possible to immediately esteem looking at the figures is how much the proposed UID distance is well correlated with problem-specific phenotype-level distances. Figure 3.5 plots the UID against the standard Hamming distance for 500 random 50-bit individuals[2]. The cloud of points does not stretch down to the origin, nor up to the maximum because it is quite unlikely to find two identical strings, or two completely different ones, in a random pool.

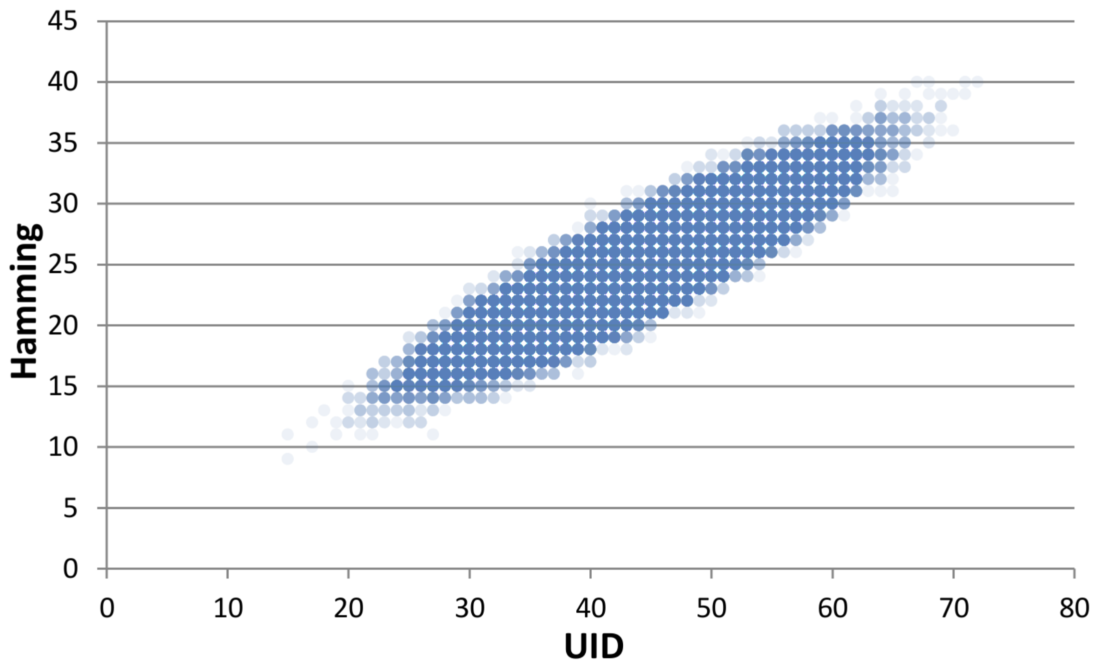


Figure 3.5. Correlation between the proposed UID distance and hamming distance in the standard OneMax problem (50 bits)– Sample of 500 random individuals.

Figure 3.6, on the other hand, plots the same data for all individuals generated during a run, until the optimal solution is reached. Since there is a strong similarity between all individuals in the same parental group, the cloud stretches down to distance zero. The correlation is even more evident than in the preceding example.

Figure 3.7 plots the proposed distance against the Levenshtein, or *edit*, distance for 500 random programs on the Assembly OneMax problem. Here, differences are more subtle and the number of overlapping values is reduced compared to the previous case. The triangular shape of the cloud is indicative: the two distances are better correlated for low values – that is, exactly when they are more useful: distinguishing

[2]Several values are overlapping.

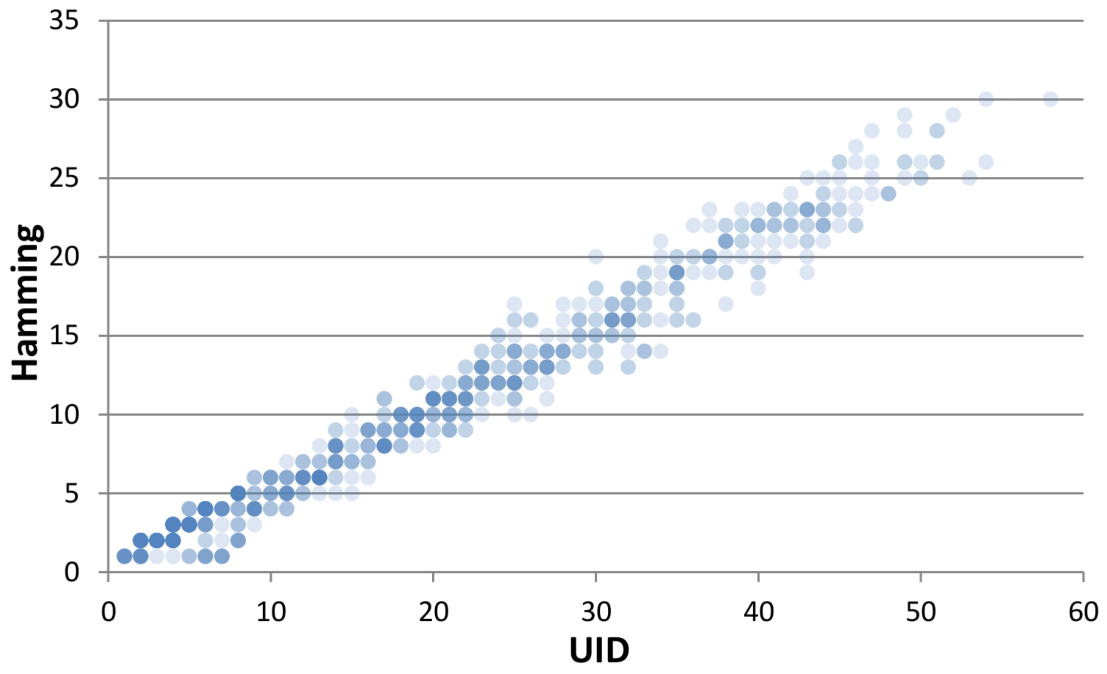


Figure 3.6. Correlation between the proposed UID distance and hamming distance in the standard OneMax problem (50 bits) – Individuals generated during a run.

between closely related individuals is in fact quite harder than discriminating between very different ones. The Levenshtein distance is a computationally-expensive metric that can be computed only at the level of phenotype. The proposed UID, on the contrary, can be efficiently calculated at the level of genotype and it is effective in estimating the distance between *similar* individuals.

Fitness Sharing

Since the proposed metric shows a heavy correlation with phenotypic distance metrics, its effectiveness can now be tested in multi-modal problems where a fitness sharing might help to spread the individuals over the search space. The aim of the following experiments is to show how using the proposed UID for fitness sharing delivers the same results as employing more rigorous problem-specific distance metrics, that are also more computationally expensive.

The first experiments are performed on the NK-landscapes (or NK-model) benchmark, tuned to obtain a rugged fitness landscape: the UID is compared to a classical Hamming distance [51]. A second set of experiments is then executed on a simple Assembly-language generation problem, where the objective is to obtain a program able to set all bits of register *%eax* to 1. In this latter tests, the UID is weighted against the Levenshtein distance [65].

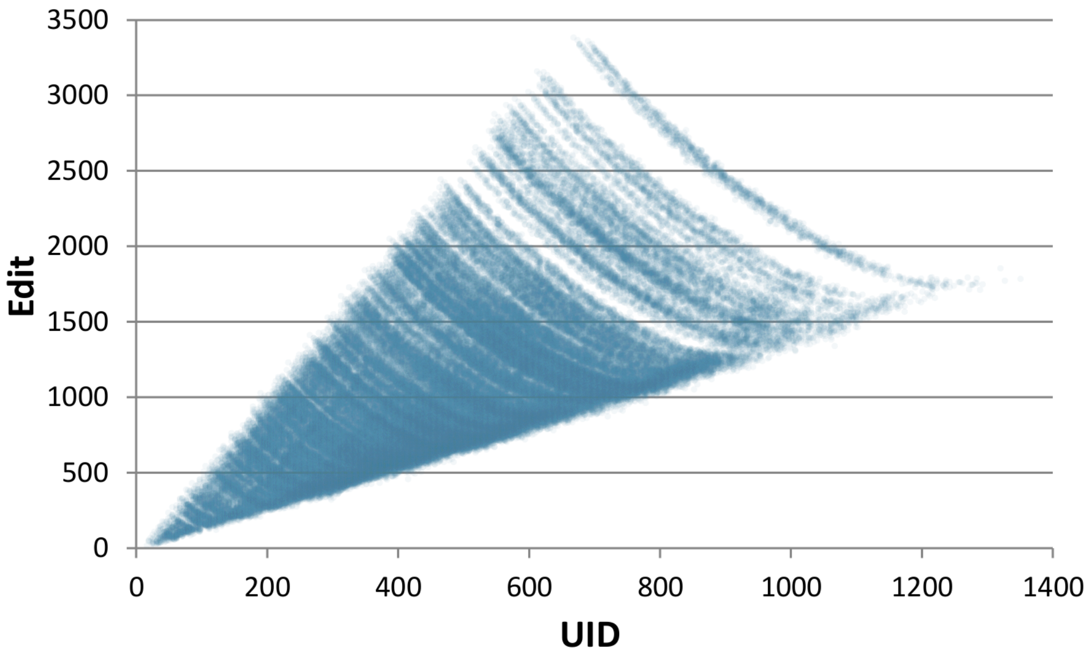


Figure 3.7. Correlation between the proposed UID distance and the Levenshtein distance in the Assembly OneMax problem (32 bits) – Sample of 500 random individuals.

In the LGP tool used for the experiments [88], μ is the population size; λ represents the number of genetic operators applied at each generation, rather than the offspring size; and σ is the *strength* of the mutation operators. Each time a mutation operator is applied to an individual, a random number r in the interval (0,1) is generated: if $r < \sigma$, the operator creates another mutation in the same individual, and a new r is generated.

It is important to notice that the parameters of the tool used in the trials have not been tuned to optimality, since the main objective of this experimental evaluation is to assess whether the fitness sharing mechanism behaves comparably when different distance metrics are applied with an equivalent radius, even with sub-optimal settings.

NK-landscapes

Parameters of the LGP tool used in the experiments are reported in Table 3.2. The mutation operator in this case simply changes the value of a random gene, while the one-point crossover selects a random cut point.

For each landscape, 10 experiments are run with the Hamming distance, and 10 with the UID, respectively, using equivalent radius measures derived from the correlation described in Subsection 3.5.3. At the end of the evolution, the fitness of

Parameter	**Value**	**Parameter**	**Value**
μ	32	P(one-point crossover)	0.5
λ	32	P(mutation)	0.5
σ	0.5	Max generations	50

Table 3.2: Parameters used during the experiments with fitness sharing in the NK-landscapes benchmark.

the best individual (*online fitness*) and the average fitness of the final population (*offline fitness*) are compared.

Fitness sharing with Hamming distance					Fitness sharing with UID				
Radius (Hamming)	Online fitness (avg)	StDev	Offline fitness (avg)	StDev	Radius (UID)	Online fitness (avg)	StDev	Offline fitness (avg)	StDev
N=16, K=14, seed=1234567890									
5	0.6710	0.0167	0.3655	0.0256	10	0.6739	0.0326	0.3798	0.0391
N=16, K=14, seed=4242424242									
5	0.6774	0.0318	0.4094	0.0142	10	0.6948	0.0304	0.4099	0.0235
N=16, K=15, seed=1234567890									
5	0.6543	0.0228	0.3819	0.0124	10	0.6468	0.0109	0.3901	0.0137
N=16, K=15, seed=4242424242									
5	0.6770	0.0209	0.3912	0.0352	10	0.6671	0.0256	0.4067	0.0316

Table 3.3: Results for the set of experiments on the NK-landscapes benchmark. Experiments with fitness sharing with the Hamming distance (**left**) and the UID (**right**); experiments with a corresponding radius are reported on the same line.

From results in Table 3.3 it is noticeable how, for the same NK-landscape, the final online and offline fitness values are very close, as well as the average distance value between individuals in the population. In fact, running a two-sample Kolmogorov-Smirnov test on corresponding distributions for an equivalent radius reveals that the distributions are undistinguishable with $p < 0.01$.

Assembly OneMax

This second set of experiments targets a simple Assembly language generation problem: the fitness of a program is the number of bits set to 1 in registry *%eax* at the end of its execution. Table 3.4 summarizes the parameters used for the LGP tool in this experiment.

The mutation operator, in this case, can add a random instruction; remove a random instruction; or change one or more parameters inside a random instruction, with equal probability. The crossover can operate on one or two cut points, with equal probability.

Table 3.5 shows the results over 10 experiments with each parameter configuration.

Parameter	Value	Parameter	Value
μ	10	P(crossover)	0.25
λ	7	P(mutation)	0.75
σ	0.7	Max generations	10

Table 3.4: Parameters used during the experiments with fitness sharing in the Assembly language generation problem.

At the end of the evolution, the fitness of the best individual (*online fitness*) and the average fitness of the final population (*offline fitness*) are compared.

Fitness sharing with Levenshtein distance					Fitness sharing with UID				
Radius (Leven-shtein)	Online fitness (avg)	StDev	Offline fitness (avg)	StDev	Radius (UID)	Online fitness (avg)	StDev	Offline fitness (avg)	StDev
3	325,927	7,205.14	312,903	19,800.8	2	320,919	12,608.4	297,899	32,800.8
5	324,939	7,992.28	309,902	27,989.6	3	324,949	8,008.23	315,909	17,616.6
10	318,909	11,422.8	292,901	20,998.7	5	314,930	15,015.5	285,923	32014.3

Table 3.5: Results for the set of experiments on the Assembly-language generation benchmark. Experiments using fitness sharing with the Levenshtein distance (**left**) and the UID (**right**); experiments with a corresponding radius are reported on the same line.

Again, the results for an equivalent radius are indistinguishable through a two-sample Kolmogorov-Smirnov test with $p < 0.01$.

3.6 μGP Operators

Genetic operators, used within an evolutionary algorithm, are the tools allowing modification and combination of individuals within a population of solutions, performing in this way the optimization process. Operators are typically divided in two main categories:

- *Mutation operators*, implemented with the aim to maintain genetic diversity within the population modifying some characters within the individual genome;
- *Crossover operators*, able to combine genome informations extracted from existing individuals in order to create new ones.

Normally, mutation operators require a single parent and generate a single new individual, whereas crossover operators are working by selecting two parents and generating one new individual. The parents selection is a critical step, since offspring depends on genetic operators characteristics and on the quality of the genome on which they are applied. In μGP parents selection is performed using a mechanism based on the *tournament selection with fitness hole*, as described in Section 3.2.

The number of times that genetic operators must be invoked to be applied to the population is a value characteristic of evolutionary algorithm settings. This value is set by the user and, as widely used within genetic computation community, it is indicated through the Greek letter λ. Due to the fact that each genetic operator may create more than one individual, the offspring size can vary at each generation and it is usually bigger than λ. This means that, if a genetic operator is unable to create a valid individual, its invocation is not considered as valid: the μGP , at each evolutionary step, will make λ valid application of genetic operators.

In μGP the operators are all implemented starting from the class *Operator*, and consequentially they are treated equally by the algorithm. Following the basic ideas of *C++* project implementation, from the aforementioned abstract class μGP implements two others generic classes, one for each macro-class of genetic operators: standard genetic operators and group genetic operators. This choice was made also for allowing the user to tweak the tool by adding their own custom operators in a quite simple way: it is enough to implement some necessary method inherited by the class corresponding to the category of operator that the user is implementing

Each operator should be implemented in order to have an internal check for the static analysis of its own applicability on a particular context, defined through constraints definition. Using this mechanism at the very beginning of the evolution, the μGP tool is able to check constraints file and verify which operators can be successfully applied to the particular structure of individual implemented. This means that the tool can automatically verify the usefulness of a genetic operator, so if it is capable to produce valid modification to a particular configuration of constraints or not. If the operator is enabled by the user but μGP discovers that it is needless, the tool disable it as default behavior. Otherwise, if the operator is disabled by the user but the initial analysis finds that the operator is applicable (and then it is potentially useful during evolution), μGP warn the user through a log message at the verbose logging level. To help the user, as default behavior, μGP performs checking all the operators and enabling only those applicable to the addressed optimization problem.

Whenever a genetic operator is applied a random number [0,1] is generated: if the random value is less than the current σ, the process is iterated. When σ is high, this operator favors exploration, generating offspring that is quite different from the parent; when σ is low, the children individual is closer to the parent, thus favoring exploitation.

3.6.1 Mutation Operators

After the definition of common parameters involving all the operators defined within the μGP tool, now we describe in detail each genetic operator. The first sections are dedicated to mutation operators, that work by selecting a single parent for each of their application:

- *Alteration Mutation*: this operator is working selecting one parent from the population, and generates a single child individual. It copied the parent and the selects a random node from its genome; the mutation is performed by setting all parameters of the chosen node to a random value taken from the possible configuration depicted within the constraints file.

- *Insertion Mutation*: the *insertion mutation* operator creates a new individual by cloning the parent and adding a randomly generated node in a random selected point of the genome. If the resulting individual is too big (with respect to the constraints) it will be killed. Due to its particular behavior, that consists in adding some genome parts, this operator will not work when individuals constraints provides for a fixed length.

- *Removal Mutation*: this operator is the dual of the previous one. It creates a child individual by cloning the parent, then randomly selects a node inside the child, and removes it. As aforementioned, there is an important check on constraint length violation: if the obtained individual is smaller than the minimum allowed size, the operator fails. It is important to notice that neither this operator will work with fixed-size individuals.

- *Replacement Mutation*: as the previous mutation operators, it creates a child individual by cloning the parent and replacing a random node of the individual genome with a new, randomly generated one.

- *Single Parameter Alteration Mutation*: this operator performs by creating a child individual by cloning the parent, then randomly selects a node (macro instance) inside the child, and a parameter inside that node. The current value of that parameter is then randomized. It is important to notice that this operator will not work if all macros in the constraints file have no parameters.

- *Subgraph Insertion Mutation*: this operator creates a child individual by cloning the parent, then inserts a whole random subgraph (instance of a subsection) at a random valid position. If the resulting individual show a non-valid number of subgraphs, the operator fails. This operator will not work on individuals with a single subsection, or with a fixed number of subsections.

- *Subgraph Removal Mutation*: this operator works creating a child individual by cloning the parent, then selecting a whole random subgraph (instance of a subsection) in the child, and removing it. If the resulting individual show a non-valid number of subgraphs, the operator fails. It is important to notice that this operator will not work on individuals with a single subsection, or with a fixed number of subsections.

- *Subgraph Replacement Mutation*: the last mutation operator works similarly to the previous ones: it creates a child individual by cloning the parent, then selects a random subgraph (instance of a subsection) inside the child and replaces it with a new random subgraph. When individuals possess only one subgraph, this operator would basically create completely random individuals, by replacing the only subgraph with a random one, so instead it fails.

3.6.2 Crossover Operators

Those described so far are mutation genetic operators, which work by selection only one individual as parent. Now we start describing crossover operators, which select two parents for creating offspring.

- *One Point Crossover*: the one point crossover operator takes two parent individuals, and creates two children by recombining the parents using a single cut point. This operator restores external references (*outerLabel* parameters) correctly, differently from the *imprecise* crossover operators. It must be noted that, in absence of *outerLabel* parameters in the constraints, the behaviors of this operator and of its *imprecise* version are the same.

- *One Point Imprecise Crossover*: this version of the one point crossover takes two parent individuals, and creates two children by recombining the parents using a single cut point. This operators, differently from the previous one, does not restore external references (*outerLabel* parameters) correctly. It must be noted that, in absence of *outerLabel* parameters in the constraints, the behaviors of this operator and of its *precise* version would be the same, so instead this one fails.

- *Two Points Crossover*: this crossover operator produces two children individuals from two parent individuals, by randomly selecting two points of cut inside corresponding subgraphs (instances of subsections) of the two individuals, and swapping the parts between the two points of cut. This operator restores external references (*outerLabel* parameters) correctly. In absence of *outerLabel* parameters in the constraints, the behaviors of this operator and of its *imprecise* version are the same.

- *Two Points Imprecise Crossover*: this operator produces two children individuals from two parent individuals, by randomly selecting two points of cut inside corresponding subgraphs (instances of subsections) of the two individuals, and swapping the parts between the two points of cut. Differently from the previous version, his operator does not restore external references (*outerLabel* parameters) correctly. In absence of *outerLabel* parameters in the constraints, the behaviors of this operator and of its not *imprecise* version would be the same, so instead this one fails.

3.6.3 Scan Operators

In μGP there are present also categories of special purpose operators, deeply different from the ones described in literature and eventually useful for particular application. The choice of using genetic operators belonging to these categories must be thoroughly analyzed: these operators could cause an explosion in offspring dimension and greatly slow down the optimization process.

- *Scan Mutation BITARRAY*: this scan operator performs a local quasi-exhaustive search in the proximity of the parent individual. One parameter of type *bitArray* is chosen at random in the individual, and several children are produced with values of the parameter close to the parent's. The number of individuals generated depends on sigma: every time a new individual is produced, a random value [0,1] is generated, and if the value is less than the current sigma, the process is repeated. At each iteration, an individual at a certain Hamming distance from the parent is created. Once all individuals at a certain Hamming distance have been created, the Hamming distance is increased. All individuals produced will share the same allopatric tag. This means that at the most one of the offspring will survive in the slaughtering step.

- *Scan Mutation CONSTANT*: it performs a local quasi-exhaustive search in the proximity of the parent individual. One parameter of type *constant* is chosen at random in the individual, and a child is produced for each possible value of the constant parameter. All individuals produced will share the same allopatric tag. This means that at the most one of the offspring will survive in the slaughtering step.

- *Scan Mutation FLOAT*: the scan mutation float operator performs a local quasi-exhaustive search in the proximity of the parent individual. It is similar to the previous one but a parameter of type *float* is chosen at random in the individual, and several children are produced with values of the parameter close to the parent's. All individuals produced will share the same allopatric tag.

This means that at the most one of the offspring will survive in the slaughtering step.

- *Scan Mutation INNERLABEL*: it performs a local quasi-exhaustive search in the proximity of the parent individual. One parameter of type *innerLabel* is chosen at random in the individual, and a child is produced for each possible value of the constant parameter (each other node the label can legally point to). All individuals produced will share the same allopatric tag. This means that at the most one of the offspring will survive in the slaughtering step.

- *Scan Mutation INTEGER*: the last scan operator performs a local quasi-exhaustive search in the proximity of the parent individual. One parameter of type *integer* is chosen at random in the individual, and several children are produced with values of the parameter close to the parent's. The number of individuals generated depends on sigma: from two to several tens of individuals are produced, with a sampling of the range specified in the parameters. All individuals produced will share the same allopatric tag. This means that at the most one of the offspring will survive in the slaughtering step.

3.6.4 Group Operators

Genetic operators discussed hitherto were designed to be used in normal evolutionary optimization. In this section are discussed the operators that operate only when the group evolution is enabled; these operators are capable to exchange individuals among groups, so they does not work directly on internal representation of individual (i.e. genome level).

- *Group Balanced Crossover*: this group operator selects two groups from the population and performs a random number of individual exchanges between them; as result of the application of this operator two new groups will be generated, characterized by the same size of parent groups.

- *Group External Insertion Mutation*: this is a group operator that randomly select a group from the population, and generated a child group by adding an individual that is chosen by an external evaluator provided by the user; in this way could be possible to interfere the evolution, driving the optimization with a non casual creation of a stronger group. This group will not work if applied to a group containing the maximum number of individuals allowed by user settings.

- *Group K Fold Scan Crossover*: this group operator randomly selects two groups from the population, slices the biggest one in K parts, and moves each part to the a new clone of the other group; as a result, this operator generates K offspring groups formed by an existing group in which a part of the bigger group is added. This operator works correctly when the group on which will be added individuals is not formed by the maximum numers of individuals allowed within a valid group.

- *Group K Fold Scan Removal*: it is a group operator that randomly selects one group, slices it in K parts, and creates K subgroups each missing one of the parts; the aim of this operator is to identify useless individuals from the given group and eliminate them. It is working if the size of the selected group is not equal to the minimum number of individuals allowed.
- *Group Random Insertion Mutation*: as the *Group External Insertion Mutation* one, this group operator selects a group from the population, and creates a child group by adding a random individual selected between the ones present within the whole population; then, a random value [0,1] is generated. If the value is lower than sigma, the process is repeated. As constraints, the selected group should be smaller in size than the maximum dimension allowed.
- *Group Random Removal Mutation*: As the previous one, this group operator selects a group, but creates a child group by removing a random individual. Then, a random value in [0,1] is generated. If the value is lower than *sigma*, the process is repeated. In this case, the application of this operator works fine if the selected individuals has not already reach the minimum size.
- *Group Unbalanced Crossover*: like the balanced one, this group crossover operator selects two groups and moves randomly some individuals between them changing their compositions and sizes, but without regarding to the size of resulting groups.
- *Group Union Intersection*: this is a group operator that randomly selects two groups from the population and tries to build new groups based on their intersection and their union.

3.6.5 Random Operator

Within μGP there is also a particular random operator, useful just in some optimization process; due to their characteristics, this operator is disabled by default.

- *Randomizer*: This genetic operator generates a new random individual starting from the constraints' description, in the same random way on which individuals in the initial population are created. As the evolution goes on, individuals created by this operator are probably going to be killed very soon, as their fitness will not be on par with individuals in the current population. Nevertheless, it could be used to maintain diversity inside the population.

Chapter 4

Evolutionary Algorithms Applications

This chapter describes a new technology designed and implemented with the aim of improving the actual implementations of evolutionary algorithms, and application of evolutionary algorithms to real optimization problems.

This technology is related to a new approach to evolution: the normal evolutionary paradigm is joined with a cooperative scheme, that consists on group individuals within the population in order to create group. The idea is to evolve each single individual as a partial sub-optimal solution to the addressed problem, and considers a group as a complete solution, formed by several partial ones.

This approach, called *Group Evolution*, foresees the implementation of new genetic operators, new fitness values and new methods to apply evaluation of individuals and groups.

Optimization processes discussed were ran all with the μGP EA tool described in chapter 3; three problems will be faced:

- The first problem is the optimization of wetland design. A wetland is an artificial basin in which running waters will be filtered and purified from pollutants by contact with particular species of plants. The optimal position of plants within the basin is an hard optimization problem that is normally faced by experts and a trial and error approach, because there is no a precise approach to reach the optimal configuration. Through the μGP and a dynamic simulator of water flows, was possible to obtain result comparable with those reached by human expert using an automatic optimization process.

- The second one is a feasibility study in which the aim is to generate malicious code and inject it into existing executable file, without compromising its correct behavior. Through this work, it was possible to identify some parts within the executable that are accessible and modifiable pursuing our purpose.

- The last work described in this chapter regards the compaction of test programs for microprocessors. The EA used in this work tries to remove some instructions from the test programs to be compacted; if the faults coverage does not decrease, it means that the removed instructions are usefulness. The goal of this work is to create an automatic system able to reduce the number of instructions forming a program for testing a microprocessor; in this way it is possible to reduce the amount of time and memory requested to execute it.

4.1 Automatic Generation of On-Line Test Programs through a Cooperation Scheme

Software-based Self-Test (SBST) can be used during the mission phase of microprocessor-based systems to periodically assess the hardware integrity. However, several constraints are imposed to this approach, due to the coexistence of test programs with the mission application. This work proposes a method for the generation of Software-Based Self-Test (SBST) programs to test on-line embedded RISC processors, which is one of the most heavily impacted by the on-line constraints. The proposed strategy exploits an evolutionary optimizer that is able to create a complete test set of programs for a considered processor module relying to a new evolutionary paradigm that exploits a cooperative scheme. Experimental results obtained high fault coverage values on two different modules of a MIPS-like processor core. These two case studies show the effectiveness of the technique and the low human effort required for its implementation.

4.1.1 Introduction

The increasing use of electronic components based on microprocessors for safety and mission-critical applications drives manufacturers to integrate more and more dependability features in their products.

Aiming at increasing product dependability, as well as coping with market and standard requirements, producers must include a series of auditing processes during the lifecycle of the product and in-mission periodic testing campaigns for error detection. For example, car manufacturers are adopting the ISO/DIS 26262 [59] standard that demands the inclusion of the on-line self-test techniques as essential test processes in critical electronic vehicle parts to insure high quality and mission safety throughout the product useful life.

One of the most interesting techniques to achieve these results is based on the application of on-line test programs. Usually, a set of test programs is first devised to test the whole processor or some modules inside of it, and then periodically run during the device lifecycle. It is possible to define a test program as a carefully devised piece of code aiming at testing some part of the processor core resorting only to the processor instruction set and exploiting the available resources of the processor. Interestingly, the test program is not only in charge of exciting processor faults, but also contributes to bringing results to an observable point. Even though this technique was firstly introduced more than 20 years ago [103], it easily matches with on-line testing requirements and it is therefore gaining again the attention of the research community.

In Software-Based Self-Test (SBST) the application of test programs is carried out by the very same processor core on the electronic device. The technique can be easily applied to on-line testing performing periodically few simple steps:

- The mission application is interrupted;
- The test program is upload in the code memory;
- The test program is executed by the processor core;
- Results are gathered and save;
- The mission application restarts.

Test programs must follow all the requirements and constraints established for the device under testing. Indeed, transforming an SBST test set to an on-line SBST test set may not be possible, or it may require excessive effort. For example, as described in [9] [15], on-line tests must preserve processor status, have a reduced execution time, and preserve memory integrity. Several works presented in the literature show why software-based techniques are often preferable to hardware approaches. However, while SBST generation approaches are described in [81] [77] [73] [94], there is not a mature technique able to cope with all the constraint introduced by the current on-line testing requirements.

In this work we concentrate on the generation of SBST programs oriented to microprocessor on-line testing. The proposed strategy exploits a new evolutionary concept that automatically generates a whole set of test programs, taking advantage of their complementarities. The proposed approach achieves very good coverage figures, and requires lower generation time and memory footprint when compared with analogous strategies. The final results gathered on a freely available version of a pipelined microprocessor clearly cope with on-line constraints requiring low human intervention.

4.1.2 Background

SBST is an emerging alternative for identifying faults during normal operation of the product, by performing on-line testing [6]. Several reasons push this choice: SBST does not require external test equipment, it has very little intrusiveness into system design and it minimizes power consumption and hardware cost with respect to other on-line techniques based on circuit redundancy. It also allows at-speed testing, a feature required to deal with some defects prompted by deep submicron technology advent.

Evolutionary algorithms (EAs) have been little, but steadily, used in the CAD community during the past 20 years [98]. EAs provide an effective methodology for trying random modifications, where no preconceived idea about the optimal solution is required. Being based on a population of candidate solutions, they are more robust than pure hill climbing. Both small and large modifications of existing solutions are possible, but with different probabilities. Recombination makes it possible to merge useful characteristics from different solutions, exploring efficiently the search space. Furthermore, EAs are quite simple to set up, require no human intervention when running and are inherently parallel. Finally, it's easy to trade-off between computational resources and quality of the results.

A versatile toolkit, named μGP (MicroGP), has been developed at Politecnico di Torino in the early 2000s and it is now available under the GNU Public License [88]. It must be noted that μGP does not aim at creating a program to solve generic problems, but rather to optimize realistic assembly-language programs for very specific purposes.

The last release of μGP is composed of three clearly separated blocks: the evolutionary core that is able to generate syntactically correct assembly programs using the information about the structure of the language from the constraint library; every new generated program is then evaluated resorting to an external evaluator that provides the evolutionary core with a measure of its goodness or usefulness (usually obtained by simulation or fault simulation). Borrowing the term from biology, this value is called fitness, and is used during the rest of the generation, together with some structural information, for enhance and improve the test programs. At the end, this process provides the user with the best test program.

The simplest, and probably most common, approach used for finding solutions of complex compound problems with an EA is based on iterative runs. That is, since a single solution is not able to solve the whole complex problem, the EA is set to solve as much as possible of the problem. Then, the solved sub-problems are removed from the original problem, and a new run of the EA is used to solve the remaining part. Iteration after iteration, the EA provides a set of solutions that is cumulatively able to solve the whole problem. However, the process is plainly inefficient, and the final set is likely to be sub-optimal and oversized. Moreover, a deep knowledge of the problem is required to identify the solved portions.

For instance, if the problem is attaining 100% fault coverage on a circuit, then at each step detected faults are removed from the fault list (the compound problem), and in each iteration the evolutionary test program generator is run against the undetected faults. This approach can lead to good solutions, as [86], despite the fact that test programs in the final test set are likely to cover multiple times the same faults.

The approach proposed in this work is based on a recent development in the evolutionary computation field, sometimes called group evolution or group selection. The EA is asked to determine a set of solutions, cumulatively able to solve the whole problem in an optimal way. Thus, only a single iteration is required, leading to a better solution and a significant reduction of computational time [104].

4.1.3 Concurrent SBST generation of test programs for on-line testing

The final goal of this work is to introduce an effective strategy to generate SBST programs, suitable to be periodically run during the device mission. It is important to notice that the proposed approach uses the fault coverage obtained by the test programs against the stuck-at faults of the module under consideration as fitness function, that is, as feedback value.

In a preliminary step, the user is required to select one by one every processor module requiring a test program. Then, targeting the selected module, the user is required to follow the proposed approach based on three steps, namely: *ISA distillation*; *concurrent test set generation*; and *final on-line set up.* At the end of the whole process, the proposed approach produces a complete test set to be applied during mission time.

The proposed method requires low human intervention, as described in the following steps:

ISA distillation

This step requires the intervention of the user, by wisely selecting among the processor instructions belonging to the processor ISA, a reduced set of instructions able to firstly excite and then propagate the results of the elaboration to a well-defined memory location.

The instructions gathered during this step are placed into μGP 's constraints library. In addition, the user must also set μGP to limit test programs' size, in order to cope with on-line testing constraints.

This step can be performed using two different strategies:

Single instructions constraint The user collects and elaborates the constraints library for μGP by gathering most of the instructions able to interact with the module under consideration. The user must be able to gather different types of instructions that prepare and apply the input values to the module under consideration, and also observe the obtained results. However, the user is not required to organize the gathered instructions in a particular way; instead, the whole organizing process is left to μGP .

Atomic block-based constraint The user, supported on his/her own experience, devises a block of instructions, called atomic block, that must be as general as possible, so as to cope with different module requirements.

Roughly speaking, the atomic block targets the module under consideration by applying suitable values to the module inputs using controlling instructions; then, the test program executes a target instruction that excites the module; and finally, results are propagated to the processor outputs through instructions that, for example, save results in appropriate memory locations.

Figure 4.1 shows the structure of a sample atomic block oriented to test the arithmetic module in a pipelined processor.

```
1. rA RNDM (constrained) value
2. rB RNDM (unconstrained) value
3. rE f(rA, rB)
4. sd_instr rE, M[rA]
```

Figure 4.1. Atomic block pseudo-code.

The first two instructions in Figure 4.1 (lines 1 and 2) load random values in the registers rA and rB. The value in rA is a constrained random address value (constrained) selected within a writable Test Memory addressing range. On the other hand, the value placed in rB is purely random without any constraint. The target instruction at line 3 manipulates rE by applying the function f(rA, rB). The selected function must be able to excite the module under test. The value in rE is later stored (line 4) in memory completing the observability task of the atomic block for testing the considered module.

It is important to notice that even though the atomic block structure is fixed, in the presented case, μGP is allowed to modify not only the random values but also the involved registers.

Depending on the module under consideration, the user may devise not only one atomic block but a set of them; and then, μGP can be set to freely combine them during the generation phase.

Concurrent test set generation

The innovative idea is to generate the complete test set of programs for the module under consideration at the same time, taking into account the constraints for on-line testing. This goal is achieved exploiting *Group Evolution* [89], a new technique able to optimize solutions composed of homogeneous elements. The approach cultivates a population of partial solutions, and exploits non-fixed sets called *groups*. Group Evolution operates on individuals and groups in parallel.

During the evolution, individuals are optimized as in a common EA, and concurrently groups are turned into *teams*. A group in itself does not necessarily constitute a team: teams have members with complementary skills and generate synergy through a coordinated effort which allows each member to maximize its strengths and minimize its weaknesses. A team comprises a group of entities, partial solutions in our case, linked in a common purpose. Teams are especially appropriate for conducting tasks that are high in complexity, whereas leveraging individual cooperation. As a result, the initial sets of random test programs slowly turns into coordinated teams, where each member tackles its specific portion of the problem.

Algorithm details The algorithm proposed is population-based: we generate and keep track of a set of distinct individuals or test programs which share the same structure. In parallel, we manage a set of groups, each one composed by a certain number of individuals. An individual, at any step, is part of at least one group.

The evolutionary algorithm exploits an external evaluator able to assign a fitness value to every individual or test program, as well as to every group considering all individuals composing it.

At the beginning of the evolutionary process, the initial population of test programs is randomly created on the basis of the instructions selected in the previous step (*ISA distillation*). Groups at this stage are randomly determined, and each individual can be part of any number of different groups. Figure 4.2. shows a sample population with 8 individuals and 4 groups. Notice for example that individual A is part of only one group, while individual B is part of 3 different groups.

The whole evolutionary process follows a generational approach: at each evolutionary step, called *generation*, a certain number of genetic operators is applied to the population. Genetic operators produce new individuals and groups starting from existing individuals or groups.

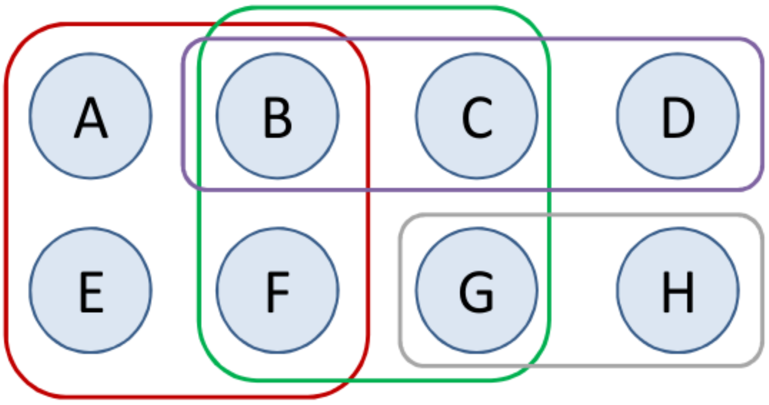


Figure 4.2. Individuals and Groups in an 8-individual population

The creation phase is made of two different actions at each generation step: Application of *group genetic operators* Application of *individual genetic operators*

Each time a genetic operator is applied to the population, parents are chosen and offspring is generated. The children are added to the population, while the original parents remain unmodified. Offspring is then evaluated, while it is not compulsory to reconsider the fitness value of the parents again.

Group genetic operators work on the set of groups available in the population. Each operator needs a certain number of groups as parents and produces some offspring groups to be added to the population. The operators implemented are: crossover, union, separation, adding-mutation, and removal-mutation.

Individual genetic operators work as in the classic EA on the population individuals. However, individuals are selected for reproduction considering not only their individual fitness, but also the fitness obtained by the group the individual belongs to. The main novelty, regarding classic evolution, relies in the fact that every time a new individual is generated, for each group the individual is part of, the algorithm generates a new copy of the group including the new individual created by the genetic operator. Since this process could lead to an explosion in the number of groups into the population, the algorithm keeps a new group only if its fitness value is greater than or equal to the one of the parent group.

At the end of every generation step, the group population is ordered and resized according to the number of groups defined in the experiment configuration. If this process leaves some individuals without any reference to a group, those individuals will be removed from the population.

Adding-mutation with external support A new genetic operator is included in the current EA for supporting the inclusion of an individual belonging to the population to a group. This operation may represent a critical task, since the evolutionary algorithm has no other information than the fitness values assigned to the individuals, which may not provide any information regarding the performance the group may achieve with the new individual. Thus, we implement a new genetic operator that receives a group and a series of individuals and then externally evaluates them in order to determine which individual is the most suitable to be added to the selected group.

Final on-line set up

At this point, the process provides the user with a series of test programs for the targeted processor modules. However, the final test set may require to be slightly tweaked to actually comply with on-line constraints. Therefore, this step requires once again human intervention, since the testing engineer must verify that the final test set complies with on-line constraints. The advantages presented by this approach are multiple, with respect to manual approaches, automatic approaches based on classic EA, and random approaches. In particular, as corroborated by the experimental results, the proposed approach:

- Reduces the generation time, since produces at every run a complete group of solutions, instead of only a single one;
- Intrinsically thrusts a cooperative effort on the members of a group, reducing the final number of test programs and coverage overlapping;
- Reduces test application time.

4.1.4 Case studies and Experimental results

The feasibility and effectiveness of the proposed approach have been evaluated on a freely available processor core called miniMIPS [1]. The processor is synthesized with an in-house developed library, resulting in a circuit containing 33,117 logic gates. The miniMIPS architecture is based on 32-bit buses and includes a 5-stage pipeline.

In the presented work, we concentrate our efforts on two different modules available in the processor: the address calculation module, involved mainly in load and store instructions; and the forwarding unit, used to avoid processor interlocks due to data hazards.

In order to run our experiments, we implement a framework that closes the loop required by the evolutionary optimizer to automatically generate test programs. The framework follows the description given in [15]. Roughly speaking, the whole process is performed as follows: μGP creates a test program that is firstly simulated using the RTL description of the processor core. At the end of this process, the logic simulator gathers the information on the I/O signals of the processor core saving it in a VCD file. Then, the VCD file is passed to a fault simulator that targets only the stuck-at faults of the module under consideration. Then, a coverage value (FC%) is calculated and, finally, it is fed back to the evolutionary optimizer closing the generation loop.

The logic simulation was performed using ModelSim SE 6.4b by Mentor Graphics, while the circuit fault coverage was evaluated using TetraMax v. B-2008.09-SP3 by Synopsys.

Address calculation module

The address calculation module is in charge of calculating the addresses for memory access operations performed when *load* and *store* instructions are executed. In pipelined processor cores, this computation is usually performed by dedicated adder, whose purpose is to sum an offset to a base value to address the RAM memory for reading or writingvalue.

Usually, this adder is not part of the processor ALU, and thus it does not perform any arithmetic computations required by instructions like *add* and *sub*. Testing an adder is often deemed as a trivial task, but in the case of the address generation module, controllability and observability are limited, and on-line requirements pose additional constraints.
The criticalities in testing this module with a software-based approach are mainly due to the type of instructions (*load* and *store* and all their variants) that activate the address calculation. In fact, a test program including many of such instructions may potentially induce some undesirable effects:

- Store instructions may corrupt the mission data and compromise the correct resuming of the system;
- Load instructions may retrieve data from memory zones (i.e., the parts containing the mission application variables) whose content can hardly be forecasted a priori, therefore compromising the signature generation, no matter how it is calculated.

The adder performing the address calculation that belongs to the synthesized miniMIPS processor is a separated unit within the execution stage, counting 342 logic gates and 1,988 stuck-at faults.
In the following we describe the outcome of the described phases while targeting the address calculation module:

ISA distillation Targeting the address calculation module, we devise a *constraint library* for μGP that exploits a similar atomic block as the one described in [14]; the adopted structure is devised constraining the memory writing and reading spaces in a strictly bounded area that comply with the on-line constraints defined in [14]. Intentionally, we decide not to include additional instructions in the constraint library. The constraint library for the address calculation module is described in a xml file containing 160 lines.

Concurrent test set generation The evolutionary optimizer was set to create an initial population of 30 test programs containing a set of blocks that varies from 1 to 10. The number of groups allowed in the population is 20 and every group may include from 2 to 15 individuals. In order to support the new genetic operator described in section 4.1.3 a *Perl* script counting 150 lines was implemented. It is important to mention that the external evaluator keeps the fault simulation results of every one of the test programs created by μGP , such that in the case μGP requires to evaluate an individual already evaluated, the fault simulation process can be skipped.

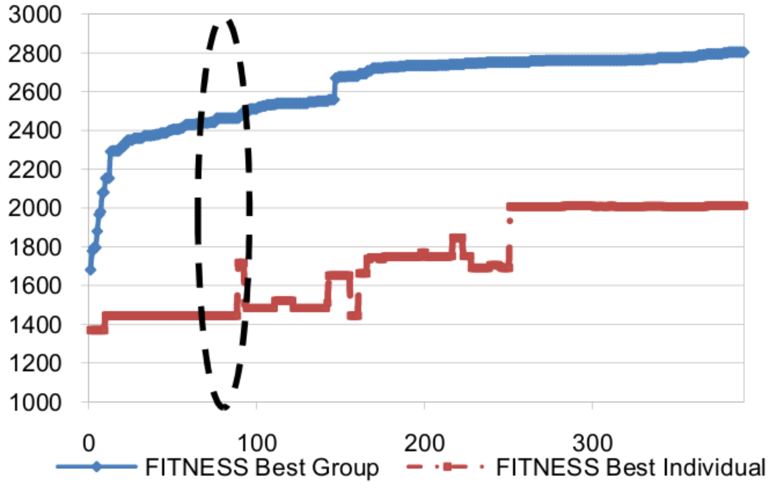


Figure 4.3. Evolutionary run for the address calculation module

Final on-line setup In a preliminary run, the test program generation took 432 generations and about 141 hours of computation in a server running Linux OpenSUSE 11.1 OS with 2 CPUs Intel Xeon E5450 @3.00 GHz and 8 GB of RAM, the evolutionary optimizer produces a group composed of 15 individuals that are able to cumulative reach 89.23% fault coverage. In synthesis, every test program belonging to the final best group contains about 6 atomic blocks, and on average the programs contain 35 instructions each. Executing the programs contained in the best group requires about 700 clock cycles. However, each individual test program is run in less than 100 clock cycles.

Let us assume that in order to correctly schedule and run the test programs to execute during the mission cycle, it is required to interrupt the mission application by no more than 75 clock cycles (7.5 μs considering a clock frequency of 10 MHz): then, the only modification required on the final test set of programs is to split one of the programs that includes 5 more atomic blocks than the rest of the programs. Thus, the considered program may be divided in two programs in order to do not exceed the clock cycle budget established for every test program.

An additional experiment was performed in order to compare the obtained results. In this case, we exploit an evolutionary approach based on iterative runs [86] creating a final test set that obtained about 87% fault coverage generating 20 test programs whose generation process and general characteristics are similar than the ones used in the previous experiment.

It should be noticed that in the presented experiment using the Group Evolution approach, even though the obtained coverage is slightly lower (less than 2%) than the one obtained in [14], the final test set is most suitable for on-line testing considering the final programs size and execution time. In addition, comparing the proposed approach with [86], the new approach obtained better results also with regards to generation time.

Forwarding unit

Register Forwarding (or data bypass) and Pipeline Interlock are functions managed by combinational logic units included in a pipelined microprocessor, to avoid data hazards.

The methods employed to avoid data hazards consist mainly in checking if there is a data dependency between two instructions that are simultaneously in different pipeline stages, and take suitable counteractions accordingly. Typically, when an instruction is passing through the decode stage in the pipeline, the system checks if its input operands correspond to the same register which any other instruction already present in the pipeline is using as output. If this is true, there is a data dependency between the two instructions and some actions have to be taken. In this case, Register Forwarding must be activated: the input data for the second instruction, instead of coming from the register file, are directly taken from the stage where the first instruction produces them. In case instruction 1 is not yet in that stage, Pipeline Interlock is used. Pipeline Interlock implies that the pipeline is stalled until the first instruction reaches the stage in which the data is produced. At that moment, Register Forwarding is used to send the result of the first instruction to the stage where the second instruction needs its operands.

According to this working mechanism, there are different possible forwarding paths and their activation depends not only on the existence of some data dependency between two instructions but also on the stage in which the instructions produce/require the data.

The forwarding unit, implemented in the miniMIPS processor core occupies around 3.4% of the total number of gates of the processor, accounting for a total of 3,738 stuck-at faults.

ISA distillation In order to target the forwarding unit, we decide to use most of the available instructions included in the miniMIPS ISA, since the module under consideration needs to exploit the data dependencies between program instructions. In this case, we do not rely to an atomic block specially devised for the considered unit.

The constraint library devised for the forwarding unit counts 53 different instructions, and it is described in an *xml* file containing 624 lines.

Concurrent test set generation Once again, the evolutionary optimizer was set to create an initial population of 30 test programs containing on average 60 different instructions. The number of groups in the population is 20 and every group may include from 5 to 15 individuals.

The same Perl script used in the previous experience for the new genetic operator was also exploited in this experiment.

Figure 4.3 shows a part of the concurrent evolution of individuals and groups during the evolutionary run performed tackling the forwarding unit. The graph shows on the X axis the first 350 generations or steps, while the Y axis indicates the number of covered faults by both the best group (the highest line) and the best individual (the lowest line) at every generation.

Interestingly, the reader can notice that the group trend is always incremental, whereas it is different in the case of the individuals. In fact, it is possible to observe that around the 90th generation, the best individual manages to cover 1,717 faults; however, it seems that this outstanding individual is not able to efficiently support a team, probably because its faults are already covered by less-performing individuals, and then some generations later it is discarded from the population. In any case, the coverage figures obtained by the best group outperforms along the evolution the best values reached by the individuals, showing that the cooperation scheme pursued by exploiting the Group Evolution is actually obtained at the end of the evolutionary process.

Final on-line setup After 616 generations, taking about 94 hours of computational time in the same server described above, the evolutionary optimizer creates a group composed of 17 individuals able to cumulative reach 85.93% fault coverage on the miniMIPS forwarding unit. Summarizing, every test program belonging to the final best group contains about 59 instructions. Executing all programs in the best group takes 1,103 clock cycles, while each test program requires less than 100 clock cycles to be run.

```
...
ori $18, $14, 35284
sw $18, 1024($0)
sw $5, 1024($0)
lw $30, 0($0)
bne $7, $30, lK32K
beq $23, $29, LK325
...
```

Figure 4.4. Sample from on of the program in the best group at the end of the evolutionary run for the forwarding unit.

Figure 4 shows a small part from one of the programs composing the best group at the end of the evolution.

Remarkably, it is possible to see different data dependencies between *ALU* instructions and *LD/SD* instructions (ori → sw), and also between *LD/SD* instructions and *BRN* instructions (lw → bne).

In order to experimentally validate our approach, we compared the obtained results in the forwarding unit with the ones obtained by a test set of programs tackling the whole miniMIPS processor core that achieves about 91% fault coverage against stuck-at faults. The test programs contained into the test set were developed following state of the art strategies such as [81]. However, the stuck-at Fault Coverage achieved on the considered forwarding unit reached only about 66%, thus proving that specific test programs are required for it.

4.2 An Evolutionary Approach to Wetland Design

Wetlands are artificial basins that exploit the capabilities of some species of plants to purify water from pollutants. The design process is currently long and laborious: such vegetated areas are inserted within the basin by trial and error, since there is no automatic system able to maximize the efficiency in terms of filtering. Only at the end of several attempts, experts are able to determine which is the most convenient configuration and choose up a layout. This work proposes the use of an evolutionary algorithm to automate both the placement and the sizing of vegetated areas within a basin. The process begins from a random population of solutions and, evaluating their efficiency with an state-of-the-art fluid-dynamics simulation framework, the evolutionary algorithm is able to automatically find optimized solution whose performance are comparable with those achieved by human experts.

4.2.1 Introduction

Nowadays, more and more specialists are becoming involved in pollution control, one of the biggest problem of our time. Ecosystems are stressed by pollution. And organic chemicals, while contributing to their destruction, can also make the water not usable by animals and humans. To bring down the quantity of chemical dissolved in water in the latter case, researchers proposed a new approach, based on bio-geochemical processes naturally present in the environment, adopting *free surface constructed wetlands*. A wetland consist of a small artificial basin, partially flooded with water and containing many vegetated areas, in which the water flows and undergoes a natural filtering process from pollutants due to particular plant species, which are able to use these waste products to support its vital functions (e.g., *Phragmites Australis, Typha Latifolia*); vegetated areas have to be distributed over the wetland in order to increase the filtering performance.In the last half century a great effort in wastewater treatment has been performed with special plants able to process polluted water. It as been demonstrated that this approach is more useful with *point sources*, characterized by little quantities of fluid polluted by high concentrations of chemicals, rather than *point sources*, characterized by big quantities of fluid polluted by low concentrations of chemicals.

To design a wetland, experts creates several configurations which are then processed by a tool to simulate the flow of water and to calculate the efficiency in terms of filtering of the configuration sets. The classic *trial and error* approach is the only viable one, since it is not possible to implement an inverse function able to identify with precision positions and characteristics of each vegetated area to be inserted in the basin, in order to obtain an optimum filtering capability.

The proposed idea is to evolve a population of individuals, each one representing a complete configuration of vegetated area. The evolutionary approach is autonomously able to optimize the performance of the wetland, while an appropriate set of constraints enforces realistic configurations. The preliminary study of a system able to automatically calculate solutions for the problem was verified in [36]. Here, the goal it to tackle a realistic problem by include different constraints.

4.2.2 Background

Wetlands

Cowardin [31] defines a wetland as an ecosystem transitional between aquatic and terrestrial ecosystems, in which the water table is usually at or near the surface or the land is covered by shallow water [13]. Before the extensive land reclamation through the last century, wetlands were common along the coasts, where they functioned as a natural buffer between inner agricultural zones and coastal areas. Today there is a pressing necessity to restore these areas and their role, defining optimal design criteria to obtain, at reasonable costs, the best removal efficiency.

The removal efficiency of natural and constructed free-surface wetlands is controlled by the time spent by contaminants into vegetated zones [78]. The role of vegetation in wetlands is important for two main reasons: water passing through vegetated zones decreases its local velocity, favoring the sedimentation of suspended solids; and biochemical processes determine a transformation of the dissolved substances. In combination with bathymetry, distribution of vegetation can produce preferential pathways of water (hydraulic shortcuts) that can substantially decrease the overall efficiency of a wetland. Removal efficiency is also affected by other hydrodynamic characteristics, as water depth and discharge, both dependent on vegetation distribution and density [5] [61]. Wetlands constructed for waste water treatment are often designed considering an average water residence time [61], even though these methods cannot adequately describe spatial configurations of vegetation in real wetlands [60]. These models, usually called *zero-dimensional*, are often used because they require few data and are easy to manage. Nevertheless, zero-dimensional models produce significant inaccuracies in the prediction of the efficiency of contaminant removal. Other *one-dimensional* models with transient storage were recently used [67] to assess the contaminant removal in a constructed wetland, giving in most cases a good approximation of breakthrough curves.

These models, however, fail to describe different flow paths across the vegetation and through main channels. The evidence of different flow pathways results in a clear bimodality of the solute breakthrough curves, that account for the different characteristic time scales of water residence time. Since spatial heterogeneity of the variables assumes a prominent role in determining the removal efficiency, the use of a more detailed *two-dimensional* approach becomes necessary to obtain reliable predictions.

Evolutionary Algorithms

Natural evolution is not a random process: while it is based upon random variations, their preservation or dismissal is determined by objective evaluations. Darwinian *natural selection* is the process where only changes that are beneficial to the individuals are likely to spread into subsequent generations, and sometimes it strikingly resembles an optimization process. Unlike most optimization processes, however, it does not require the ability to design intelligent modifications, but only the assessment of the effect of random modifications.

Several researchers, independently, tried to replicate such a characteristic to solve difficult problems more efficiently. Evolutionary computation does not have a single recognizable origin, but most scholars agree on identifying four macro areas: genetic algorithms [55], evolution strategies [92], evolutionary programming [41], and genetic programming [63].

The different paradigms share some key concepts, and can be cumulatively called evolutionary algorithms. An EA starts by generating an initial set of usually random candidate solutions for the given problem. These solutions, called *individuals*, are evaluated using problem-dependent metrics. The result of the evaluation, that is, the *goodness* of the solution, is termed *fitness*. The set of candidate solutions, also known as *population*, is then sorted on its fitness values. Subsequently, offspring is produced by altering the existing solutions: often the best solutions have a higher probability of being selected for reproduction. Offspring might be added to the existing population, or replace it entirely; in any case, some of the worst solutions are deleted before iterating the process, starting from reproduction. When a given stop condition is met, the iterations end and the best solutions are returned to the user.

Being based on a population, EAs are more robust than pure hill climbing. Both small and large modifications are possible, but with different probabilities. Sexual recombination makes it possible to merge useful characteristics from different solutions, exploring efficiently the search space. Furthermore, EAs are quite simple to set up, and require no human intervention when running. They are inherently parallel, and a nearly-linear speed-up may be easily achieved on multiple instruction/multiple data (MIMD) architectures. Finally, it's easy to trade-off between computational resources and quality of the results.

4.2.3 Proposed Approach

In the proposed approach the design of a wetland is fully automated exploiting an evolutionary algorithm. Each individual of the population represents a complete configuration of the wetland, expressed as a set of patches of vegetation arranged within the area of the basin; each vegetated area is defined by its position and diameter. The evolutionary algorithm handles the creation and evolution of individuals, while the actual evaluation is performed by a tool able to simulate the flow of water within the wetland and calculate the filtering capacity. Differently from the feasibility study, candidate solutions has been provided more stringent constraints in order to evolve towards optimized solutions close to a real ones. This constraint has been applied to the maximum area that can be covered by vegetation patches; the limit was set at 60%, in order to push the evolution towards the realization of optimized individuals describing more closely a configuration similar to those that are actually made.

Mathematical Models

A wetland is modeled using a two-dimensional depth averaged model that solves hydrodynamics, coupled with a two-dimensional solute transport equation with a first order decay term. Under the assumption of hydrostatic pressure, stationary flow, and negligible wind and Coriolis forces, the depth-averaged velocity field and water depth can be described by the following equations [112]:

$$\frac{\partial(hU)}{\partial x} + \frac{\partial(hV)}{\partial y} = 0 \tag{4.1}$$

$$\frac{\partial(hU^2)}{\partial x} + \frac{\partial(hUV)}{\partial y} = = -gh\frac{\partial z_s}{\partial x} + \frac{1}{\rho}\frac{\partial(hT_{xx})}{\partial x} + \frac{1}{\rho}\frac{\partial(hT_{xy})}{\partial y} - \frac{\tau_{bx}}{\rho} \tag{4.2}$$

$$\frac{\partial(hUV)}{\partial x} + \frac{\partial(hV^2)}{\partial y} = = -gh\frac{\partial z_s}{\partial y} + \frac{1}{\rho}\frac{\partial(hT_{yx})}{\partial x} + \frac{1}{\rho}\frac{\partial(hT_{yy})}{\partial y} - \frac{\tau_{by}}{\rho} \tag{4.3}$$

The quantities U and V represent the depth-averaged velocities $[ms^{-1}]$ along the x and y direction, respectively, h is the water depth $[m]$, z_s is the water surface elevation $[m]$, and ρ the water density $[kgm^{-3}]$. The bed shear stresses τ_{bx} and τ_{by} $[Nm^{-2}]$ in the x and y direction respectively are calculated using the following relationships:

$$\tau_{bx} = \rho c_f m_b U \sqrt{U^2 + V^2} \tag{4.4}$$

$$\tau_{by} = \rho c_f m_b V \sqrt{U^2 + V^2} \tag{4.5}$$

In the case modeled here, the bed slope is set to zero and the investigated velocity range makes it possible to consider the friction coefficient as a constant. This assumption generally holds where the velocity is sufficiently fast to assume turbulent flow. For a flat bathymetry, the bed slope coefficient m_b is unitary and the coefficient of friction c_f can be rewritten using Manning equation as $c_f = gn^2h^{-1/3}$. The effect of different vegetation densities is modeled here using different values of Manning roughness coefficient. This choice is confirmed by many studies that relate vegetation density, stem diameter and flow conditions to an equivalent roughness coefficient [9] [50] [110]. Fluid shear stresses $T_{ij}(i, j = x, y)$ associated to viscous and turbulent effects, are determined using the Boussinesq assumption:

$$T_{xx} = 2\rho(\nu + \nu_t)\frac{\partial U}{\partial x} \tag{4.6}$$

$$T_{xy} = T_{yx} = \rho(\nu + \nu_t)(\frac{\partial U}{\partial y} + \frac{\partial V}{\partial x}) \tag{4.7}$$

$$T_{yy} = 2\rho(\nu + \nu_t)\frac{\partial V}{\partial x} \tag{4.8}$$

where ν, ν_t, are the kinematic and eddy viscosities $[m^2s^{-1}]$. Since the kinematic viscosity has a lower value than the eddy viscosity, it can be neglected in most cases. For a turbulent flow regime, as it was assumed in this preliminary study, ν_t can be expressed using Elder depth-averaged parabolic model [38] as $\nu_t = \alpha U_* h$, where the term α is an empirical coefficient $[-]$ and U_* is the shear velocity $[ms^{-1}]$. For longitudinal dispersion Elder proposed a value of the coefficient α of about 5.9 [38], for transverse dispersion, Fischer found that α varies between 0.3-1.0 in irregular waterways with weak meanders [40]. In accordance with [7] [112] a value of 6.0 and 0.6 was chosen for the longitudinal and transversal dispersion coefficients respectively.

Solute transport of a reactive tracer through the wetland is simulated with a depth-averaged solute transport model accounting for the effect of advection, turbulent diffusion, dispersion and decay. In the simulations, the tracer is assumed to interact with vegetation and the chemical breakdown due to the permanence in the vegetated zones is modeled with a first order decay relationship. The equation governing the transport of a reactive tracer in the wetland can be modeled as:

$$\frac{\partial(hUC)}{\partial x} + \frac{\partial(hVC)}{\partial y} = = \frac{\partial}{\partial x}(hE_x\frac{\partial C}{\partial x}) + \frac{\partial}{\partial y}(hE_y\frac{\partial C}{\partial y}) - h\lambda C \tag{4.9}$$

where C is the depth-averaged solute concentration $[kg m^{-3}]$, U, V are the vertically integrated velocity components under steady flow conditions $[m s^{-1}]$ in the x, y directions respectively. Coefficient E_x, E_x $[m^2 s^{-1}]$, account for both turbulent diffusion and dispersion. A constant homogeneous value of E_x, E_y is chosen $(10^{-5} m^2 s^{-1})$ throughout the entire domain.

Evolutionary Core

The EA used is μGP[88], is a versatile toolkit developed at Politecnico di Torino in the early 2000s and available under the GNU Public License from Sourceforge[1]. μGP original use was to assist microprocessors' designers in the generation of programs for test and verification, hence, the Greek letter mu in its name. But over the years has been used as optimizer in a much wider spectrum of problems, including numerical optimizations.

The algorithm initially creates a set of random candidate solutions to the given problem, that are then evaluated, and sorted by their fitness value (see Subsection 4.2.3). Offspring is then created favoring the fittest individuals and also trying to favor diversity among the population. New candidate solutions are then evaluated and added to the initial population. Solutions are again sorted, and the worst ones are removed until the population returns to its original size. The process is then iterated, starting from offspring generation, until a stop condition is reached.

Two categories of genetic operators are used to generate the offspring: *mutations*, or single-parent operators, and *crossovers*, or recombination operators. Mutation operators create new candidate solutions by altering one single parent solution; crossover operators mix the information contained in two or more parents solutions to create offspring. The most common operators are available inside μGP, but the toolkit also implements *differential evolution*, and other operators specially calibrated for real parameters.

Individuals are internally represented as a multigraph, μGP relies on a external configuration file constraints the multigraphs to sensible structure, and maps the internal individuals to valid solutions of the problem. In the specific context, each individual encodes a candidate configurations of the wetland, that is, it details the features of the several patches of vegetation, with variable number of occurrences from 20 to 35, that are going to be placed in the water; the order in which the patches are described within the individual is irrelevant. All islands are assumed to be of circular shape. Since they can overlap, however, they can create more complex shapes. An island is characterized by its position (x, y coordinates expressed in real values) in the wetland and its radius; in this simplified approach friction value is

[1]http://ugp3.sourceforge.net/

always the same. An island's position is constrained by the size of the wetland; its radius is constrained following the minimum and maximum size of actual islands of vegetation used in real wetlands.

Fitness Function

The definition of an appropriate fitness function is a key aspect in the use of an EA. The process of evolution is based on *differential survival*, that is, different individuals must have a different chance to spread their offspring in future generations. In the artificial environment modeled by an EA, it is essential that different individual get different fitness values. It is a common practice to include in the fitness some heuristic knowledge, in order to help the EA explore the most promising regions of the search space.

In μGP, the fitness is not a single value but a vector of positive coefficients. The individual A is considered to be fitter than the individual B if the first j elements of the two fitness vectors are equals, and the $(j+1)-th$ element of the A's fitness is greater than the $(j+1)-th$ element of the B's fitness. In the context of wetland optimization, three values have been used.

In order to evaluate the goodness of a candidate wetland layout, a simulation of the hydrodynamic field is performed extracting computed values of discharge $Q\,[m^3\,s^{-1}]$ and water depth h at the inlet and at the outlet sections of the wetland. During the simulation, a *reactive tracer* with a known concentration is injected at the inlet. Thanks to the presence of vegetation the tracer is gradually degraded and reaches the outlet section. Mass flux $\hat{M}\,[kg\,s^{-1}]$ passing through these sections is measured, and the difference between the two values represent the first parameter of the fitness function. In order to obtain the optimal vegetation distribution, this difference must be maximized.

On the other hand, a candidate layout must still let the water flow, avoiding configurations where the vegetation is so dense to make the flow impossible. The energy requested by the water to flow can be represented by the difference between the water depth at the inlet and outlet section. This difference represents the second parameter of the fitness function. This parameter is minimized by the algorithm: solutions that completely block the water flow are then heavily penalized.

The third and last fitness parameter measures the difference of discharge between the inlet and the outlet sections of the wetland. This value assures that the stationary flow conditions are reached and that the mass fluxes are finely computed. This discharge difference is strongly minimized.

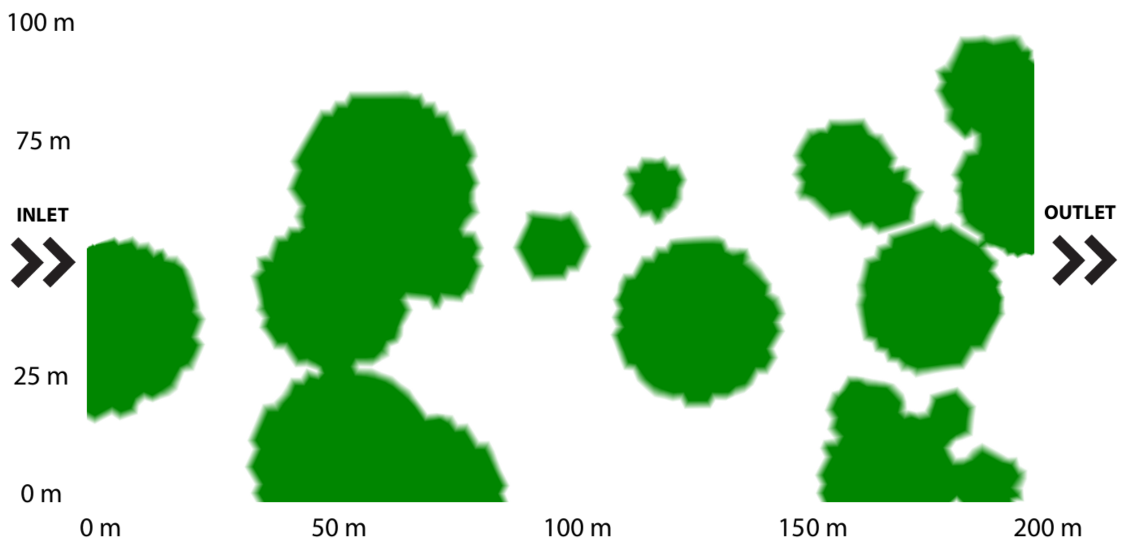


Figure 4.5. Individual B: Representation of the phenotype of an individual extracted from the first generation of evolution; dark areas show the distribution of vegetation over the wetland surface.

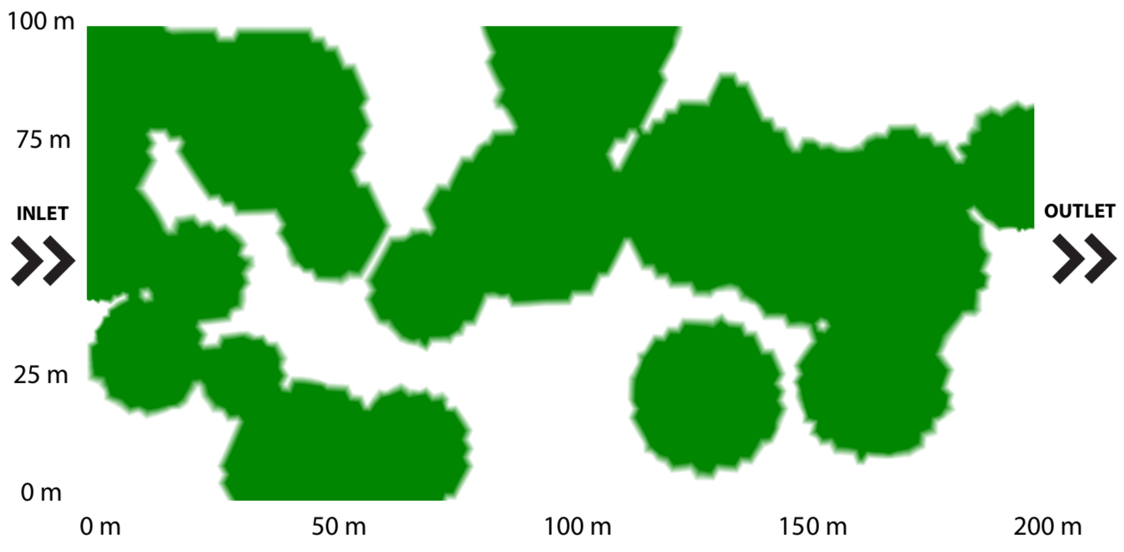


Figure 4.6. Individual 7: Individual with percentage of vegetation next to the maximum limit but without good filtering performance, due to the distribution not optimized within the basin.

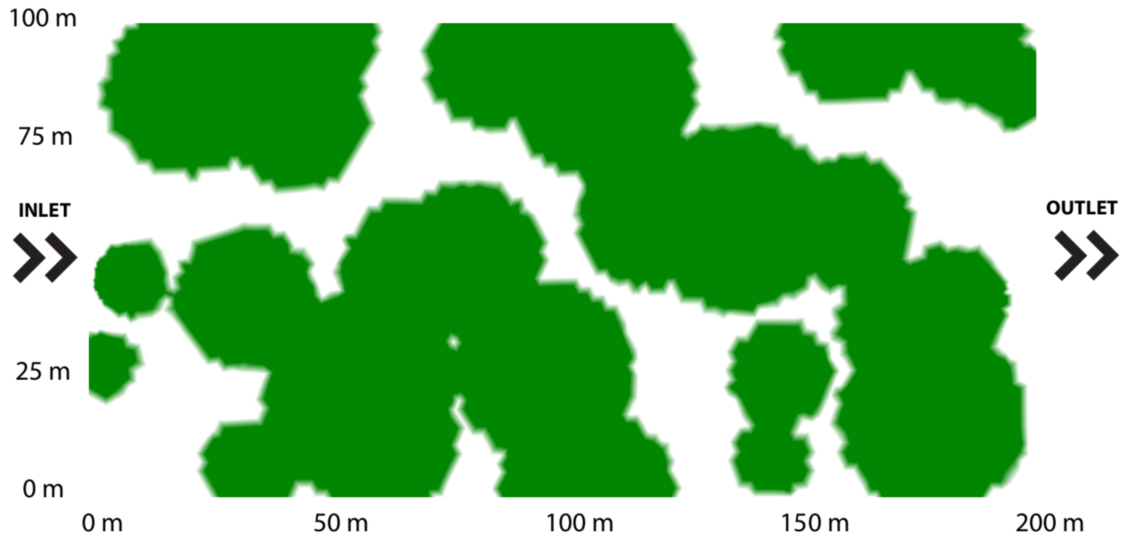


Figure 4.7. Individual AAU: Representation of the individual that reached the best optimization level. The percentage of vegetation is close to the imposed limit to 60% but, thanhs to the best arrangement of vegetation patches, its filtering performance is optimal.

4.2.4 Experimental Evaluation

Setup

The artificial basin take into consideration in this work has a rectangular shape with dimensions $200m$-long-by-$100m$-wide, with a water depth considered constant over the entire surface and equal to $0.5m$. The inlet and outlet sections are located at the centre of the shorter sides of the wetland and have 10 m of size amplitude. In this way can be reached two important objectives: the first, related to the proportions of the area, concerns the total spread of the incoming water flow over the entire section of the basin; the second, due to the constant depth, makes this basin more similar to the natural ones and also makes it possible to simplify the system, which will not consider any slopes of the basin's bed [111] [112]. In addition, a constant discharge of $0.2m^3s^{-1}$ is imposed at the inlet section. The rest of the wetland was considered impermeable and laws of friction have not been applied at the side walls. In order to monitor the filtering process of the wetland, within the inlet section is injected a reactive solute with a constant concentration of $1kgm^{-3}$; in this way it is possible to extract the fitness value (which indicates the filtering capability of the basin) by calculating the average value of the concentration of this reagent in outlet area.

In order to simulate the hydrodynamic flow within the basin and the correct values of decay related to pollutants, it has been necessary to set some parameters into the simulation tool. The basin was defined through an adaptive triangular mesh, so as to ensure a sufficient numerical stability and the required resolution in case of steep gradients of the hydrodynamic and solute transport solutions. In addition, was applied to each node a value of the Manning roughness coefficient and a decay value, depending on the structure of each individual. In the particular configuration of this experiment, in which we impose the constraints that cannot exist individuals with vegetated area greater than 60 % related to the total area of wetland, it has been chosen to simplify the decay coefficients, and the the structure of the vegetated patches. In conclusion, it was chosen to apply a single law of decay to a node of the mesh in which there is an island, or a zero coefficient otherwise; it was chosen a decay coefficient equal to $5^{-6}s^{-1}$. In the same way, Manning roughness coefficients are set to $0.20sm^{-\frac{1}{3}}$ to nodes with vegetation, and $0.02sm^{-\frac{1}{3}}$ otherwise.

As previously introduced, to achieve this automatic optimization system were used two different tools, both open-source and freely available on internet. The tool used for evolutionary algorithm is μGP version 3.2.0 (revision 198). To simulate and evaluate each individual instead was used a tool called *TELEMAC2D*, part of the wider set of programs *openTELEMAC* [45] [52]. The code of the latter has been specifically modified in order to extract information relating to fitness in the format required by the μGP tool.

Each individual evolved by the evolutionary tool is converted to the *TELEMAC2D*format, that consists of a map of basin's nodes, and each of these nodes can be covered or not by a vegetation patch. For this reason, each individual undergoes a sort of pre-processing that inserts in the nodes of the map values associated with vegetated areas. The process has been elaborated on a single machine, equipped equipped with an *Intel Core i7-950* CPU running at 3.06 GHz, and the whole system was setting in order to process up to 4 individuals simultaneously, with an average computation time of 90 minutes for each individual.

EA Configuration and Result Discussion

In order to obtain the results described in this work, the EA has been configured in such a way to create a random initial population of 20 individuals ($\mu = 20$), on which they are applied, at each generation of the evolution, 12 genetic operators ($\lambda = 12$) chosen among the 20 available in μGP tool. The entire process evolved for 90 generations, for a total of 1070 individuals generated. During the conversion of individuals to the format compatible with *TELEMAC2D*, a certain percentage of them was discarded because it was violating the introduced constraint about maximum area that vegetation patches can cover.

Starting from a random population, the evolution has shown several interesting features, which show the actual goodness of this approach. Among individuals of first generations, its possible to find some as the individual *B* which are formed by a low number of vegetated areas clearly separated between them, configuration that shown a low filtration capacity; in particular, the configuration shown in Figure 4.5 ensures a performance of pollutant reduction of 21% respect to the inlet concentration. As evolution proceeds, grows the trend of evolutionary algorithm to generate individuals which respect the constraint of the maximum coverage and, using the maximum available number of islands, the EA is able to combine them to create complex shapes able to modify the water flow and to optimize the filtering performance.

The figure Figure 4.6 and Figure 4.7 compares the two individuals 7 and *AUU*, both characterized by a vegetated coverage very close to the imposed limit of 60%, but with different fitness. Individual 7 belongs to the third generation, in which evolution is still very close to the starting stage and, despite the use of maximum coverage allowed, performances in terms of filtering amount to 27%. Individual *AUU* instead represents the best configuration achieved in this experiment, comparable to previous in terms of vegetated area; in this case the filtering capacity has been optimized to achieve performances of 33.2%.

4.3 Towards Automated Malware Creation: Code Generation and Code Integration

The analogies between computer malware and biological viruses are more than obvious. The very idea of an artificial ecosystem where malicious software can evolve and autonomously find new, more effective ways of attacking legitimate programs and damaging sensitive information is both terrifying and fascinating. The work proposes two different ways for exploiting an evolutionary algorithm to devise malware: the former targeting heuristic-based anti-virus scanner; the latter optimizing a Trojan attack. Testing the stability of a system against attacks, or checking the reliability of the heuristic scan of anti-virus software could be interesting for the research community and advantageous to the IT industry. Experimental results shows the feasibility of the proposed approaches on simple real-world test cases. A short paper on the same subject appeared at the *29th Symposium On Applied Computing* (SAC'14).

4.3.1 Introduction

Malware is a collective noun denoting programs that have a malicious intent – the neologism standing for *mal*-icious soft-*ware* [74]. Specifically, malware usually denotes hostile, intrusive, or simply annoying software programmed to gather sensitive information, gain access to private systems, or disrupt the legitimate computer operations in any other way. Since computer technology has nowadays emerged as a necessity in various aspects of our day to day life, including education, banking, communication, and entertainment, the threat posed by malware can't be overlooked.

The most popular form of malware is represented by *computer viruses*, a term coined by Fred Cohen in 1983 [26]. Viruses are programs able to replicate themselves and infect various system's files. As many other in computer science, the idea of self-replicating software can be traced back to John von Neumann in the late 50s [107], yet the first working computer viruses are much more recent. *Creeper*, developed in 1971 by Bob Thomas, is generally accepted as the first working self-replicating computer program, but it was not designed with the intent to create damage. On the other hand, the virus *Brain*, written by two Pakistani brothers and released in January 1986, is widely considered the first real malware [24].

Since the late 90s, computer malware creation has emerged as a commercial industry with revenues skyrocketing to several million dollars a year [12]. Programs that fight malware are generally called *anti-virus*. However, nowadays, the majority of threats are not posed by viruses themselves, but by *worms* and *trojans*. The former are self-replicating software able to send themselves to other computers on the Internet or a network; the latter are software that emulate the behavior of an authentic, legitimate program but also perform some fraudulent, hidden action.

Several terms may be used to describe specific malware, denoting their purpose, replication strategy or specific behaviors. These terms are clearly non-orthogonal, and the same program may be described by several of them. A *spyware* is a software which is installed on a computer system to hijacks his personal and confidential information. A *keylogger* is a particular type of spyware that records user interactions, trying to steal passwords or credit-card data. A *rootkit* is designed to take control of infected machine by gaining administrator access of the system – the name comes from the term *root* under UNIX. A *dialer* is a program that connect the telephone line to a fraudulent provider. A *botnet* is a remotely controlled software, and the machine under control is sometimes called a *zombie*. *Adware* is generic advertising-supported software whose functionality is to displays or downloads the advertisements to a computer.

Another common classification distinguishes five different generations of computer programs [96]: first generation malwares cause infection by simply replicating their code into other software; second generation have additional functionalities, such as the ability to identify files already infected (*self-recognition*); the third generation marks the appearance of *stealth* techniques to avoid detection by anti-virus software; the fourth-generation malware possess *armoring* procedures specifically designed against removal and analysis (anti-anti-virus techniques); finally, malware of fifth generation applies algorithms to obfuscate its code with every replication, altering the program structure itself.

While the anti-virus industry is able to counter most of the menaces few days after they appear, creating tools for protecting users against *0-day malware*, that is, unknown threats as soon as they appear, is a taxing problem [17].

This work proposes to exploit an Evolutionary Algorithm (EA) to create, or rather, optimize, malware. The EA is used with two different goals: to make malware undetectable by existing anti-virus program; to optimize the injection of the code inside a given host, creating a Trojan horse. Producing new malware with negligible human intervention could be extremely advantageous to anti-virus producers to test and enhance their products. Moreover, the creation of trojans could also be used, for example, to test the security of computer infrastructures.

4.3.2 Background: Stealth and Armoring Techniques

Anti-virus programs are designed to detect and remove all malicious applications, assuring the continued integrity of a system. Nowadays, these applications are perceived as needed in Windows-based operating systems (OSs): new versions of this OS include anti-virus software directly in their basic distributions. Despite Mac owners sense of security, recently, all main anti-virus companies also launched products targeting specifically Mac OS. Still to a lesser extent, the same phenomenon is visible for Linux.

The most simple, and widely used, technique for detecting malware is to recognize specific fragments of code. This approach is called *signature-based detection*, because the comparison is performed calculating and checking specific signatures. While moderately efficient, this approach suffers from important drawbacks. The more obvious is that to be effective, the specific malware must be already known, its signature analyzed, recorded, and provided to end-users. However, this can be hardly achieved because tens of thousands of new malware appear every single day[2].

The traditional viruses are able to *infect* executable programs by appending their code to the existing one [66]. This kind of attacks may be easily detected because the size of the compromised program is modified. In order not to increase the host size, the *cavity viruses* such as *CIH* or *Chernobyl*, both appeared in 1998, infect a program by overwriting its code. This was possible due to the many empty gaps in the file format for executables and object code called *Portable Executable* (PE). All these malwares are easily detected resorting to signature scanning, and was able to spread only thanks to the relative slowness in the process of getting virus samples and delivering the new signatures to end users.

From the 3$^{\text{rd}}$ generation, malicious programs start hiding their own code from scanners through encryption [113]. The virus called *Cascade*, appeared in 1986,

[2]*Panda Security* claims to have detected 27 million new forms of malware in 2012, an average of 73,000 per day. According to *Kaspersky Lab*, nearly 200,000 new malware samples appear around the world each day, while, probably due to a different classification, *McAfee* reports the slightly smaller figure of 46,000.

adopted a symmetrical encryption/decryption routine based on a XOR cipher. Despite its simplicity, the ploy was demonstrated quite effective: the decryption routine was so small that caused several false-positive, and even when it was correctly detected, the anti-virus programs were unable to discriminate between the different strains of the virus.

Concurrently to improvement in signature-based detection, new stealth strategies were devised by malware authors, namely: *oligomorphism*, *polymorphism* and *metamorphism*. Oligomorphic malwares mutate their own decrypt routines. An example of such technique is a virus called *Whale*, first detected in late 1990. Whale, as the early oligomorphic malwares, could generate at most a few hundreds of different encryption schemes, and thus could still be detectable using signatures. Newer ones created decrypt routines dynamically, making harder for the anti-virus vendors to write comprehensive signatures able to catch all variations. Indeed, history showed that it was practically infeasible to catch every new strain of oligomorphic malwares using simple signature.

Polymorphic malwares are even more able to escape signature-based detection. A *polymorphic engine* is able to create many distinct encryption schemes using obfuscation methods, such as dead-code insertion, unused register manipulation, or register reassignment. The *Tequila* and *Maltese Amoeba* viruses caused the first widespread polymorphic infections in 1991 [101]. In both oligomorphic and polymorphic malwares, the code may change itself each time, but the function, that is, the semantic, is never modified. Thus, anti-virus programs may simulate the execution of a potential malicious application to recognize the pattern of operation.

To thwart such analyses, metamorphic malware denotes polymorphic programs able to rewrite their own polymorphic engine. This is usually performed translating their binary code into a temporary representation, editing such representation of themselves, and eventually translating the edited form back to machine code [82]. A well-known example is the virus *Simile*, also known as *Etap* or *MetaPHOR*, appeared in 2002. Even more advanced stealth techniques are *permutating malware*, that, instead of generating new instructions like polymorphic programs, modifies existing ones, and thus does not alter their size. The term is used by the Russian virus writer *Zombie* in a set of articles appeared under the title *Total Zombification*.

In 2002, the same Zombie created the *Mistfall engine* using *code integration*, a technique similar to metamorphism applied to cavity viruses. The Mistfall engine decompiles PE files; moves code blocks for creating space; inserts itself; regenerates code and data references, including relocation information; and eventually rebuilds the executable. The virus *Zmist*, or *Zombie.Mistfall*, was the first one to exploit the technique, and scholars defined it “one of the most complex binary viruses ever written” [102].

Most anti-virus programs cope with camouflage resorting to dynamic and static heuristics analysis. In the former ones, a potential malicious application is executed on a virtualized sandbox system, then the modifications brought about by the program are checked relying on heuristic measures or specific triggers. On the contrary, the latter are based on the mere analysis of the code.

While dynamic heuristics may look promising, they require a significant amount of CPU-time and malware authors devised specific counter measures, called *armoring*, against them. Fore instance, some malwares adopt *stealth strategies* to prevent an infected system to report their presence, and made necessary to run the anti-virus tools from a clean environment like by booting a *live* operating system from a portable device. Other anti-anti-virus techniques prevents emulation using undocumented CPU instructions, CPU intensive routines and other tricks, or detects whether a virtualization is in progress and do not execute on emulated systems at all [93]. As a result, effective static heuristics are an essential step in protecting from threats.

4.3.3 Automated Malware Creation

The idea of creating, or optimizing, malware in an automatic way is fascinating. Moreover, starting from the choice of the term “virus”, the connections between computer software trying to penetrate legitimate systems and biological infectious agents that replicate inside the living cells of other organisms are more than evident [96].

Since early 90s, hundreds of malware creation toolkits were introduced, enabling individuals with little programming expertise to create their own customized malware. The *Virus Creation Laboratory* (VCL) is one of the earliest, it was released in 1992 by the *NuKE* hacker group, and featured a nice user interface and documentation. The same year, a felon under the name of *Dark Avenger* distributed a polymorphic toolkit: *The Mutation Engine* (MtE). MtE enabled neophyte programmers to automatically extend their malicious code into a highly polymorphic one. In the following years similar toolkit appeared, like *Dark Angel's Multiple Encryptor* (DAME) written by the felon named Dark Angel, and the *TridenT Polymorphic Engine* (TPE) by Masud Khafir.

An early approach to create malware through an EA was proposed, with noble intents, in [75]. The approach was based on rearranging existing blocks of code, and it was unable to create a virus at the level of a single instruction. Despite its limitation, the very idea of autonomously evolved malware is still frightening [57].

This work proposes to exploit an EA to optimize malware, tackling two of the main problems faced by anti-virus companies: 0-day detection and trojans' infection. Both approaches exploit the open-source EA called μGP [88], which is able of both optimizing numeric values and handling assembly programs, adding, subtracting or changing single instructions.

Tackling 0-day detection, the EA is used to automatically create a new strains of attackers. The new malware is optimized in order not to be detected by existing using static heuristic analyses. The result might provide some insights about the weakness of adopted approaches. Conversely, tackling trojan generation, the EA is given the objective to find weak spots inside a target host software, where malware code can be seamlessly inserted without altering the target's behavior. Such evolved malware could be used to assess anti-intrusion mechanisms of secure environments.

Determining that the generated software is an effective malware and that the full functionalities of an existing program have been preserved are two intractable problems. However, the use of EA enable to find potentially acceptable solutions in a limited amount of time and with limited computational resources.

Evolutionary Algorithms and μGP

EAs are stochastic search techniques that mimic the metaphor of natural biological evolution to solve optimization problems [37]. Initially conceived in the 1960s, the term EAs now embraces different paradigms like *genetic algorithms*, *evolutionary strategies*, *evolutionary programming*, and *genetic programming*. All EAs operate on a population of individuals; underlying, each individual encodes a possible solution for the given problem. The goodness of every solution is expressed by a numeric value called *fitness*, obtained through an evaluator able to estimate how well the solution performs when applied to the problem. An evolutionary step, called generation, always consists of two phases: a stochastic one where some of the best individuals are chosen at random to generate new solutions; and a deterministic one, where solutions are ranked by their fitness and the worst ones are removed from the population. The process is then repeated until a user-defined stop condition is met.

Over the past decade, EAs have been successfully employed as optimization tools in many real-world applications [88]. EAs provide an effective methodology for tackling difficult problems, when no preconceived idea about the optimal solution is available. While it is not usually possible to mathematically guarantee that the optimal solution will be found in a finite amount of time, EAs have been demonstrated able to perform much better than traditional optimization techniques in several practical NP-hard problems.

μGP is a general-purpose EA toolkit developed by the CAD Group of Politecnico di Torino in 2002 and now available under GPL [95]. While the first version was developed specifically to generate assembly language [97], the latest release can be used to tackle a wide range of problems, including numerical optimization.

In the μGP toolkit, the candidate solutions of a problem are represented as graphs, while the problem itself is indirectly modeled as an external script that evaluates each candidate solution and supplies the tool with a measure of its goodness. Due to this loose coupling, μGP can be used on different problems with no modifications needed.

Configuration files in eXtensible Markup Language (XML) describe individuals' structure and all necessary parameters such as population size, stop conditions, number of genetic operators activated at each step. Since in the specific problem individuals map sequences of keys, the related graphs are linear genomes.

Code Generation

In code generation, the EA is used to create a new malware, with the precise intention not to be detected by existing scanners. While it would be theoretically possible to make the EA discover the patterns of malware autonomously, it is far more advantageous to feed the initial population with examples of working software to obtain successful individuals in a far more reasonable amount of time. Thus, the code of several malicious applications can be converted into the EA's internal representation of individuals. The evolution is then started, and the EA rearranges freely freely materials from the individuals provided in the initial population in order to create new malware. Figure 4.8 shows the structure of the proposed framework.

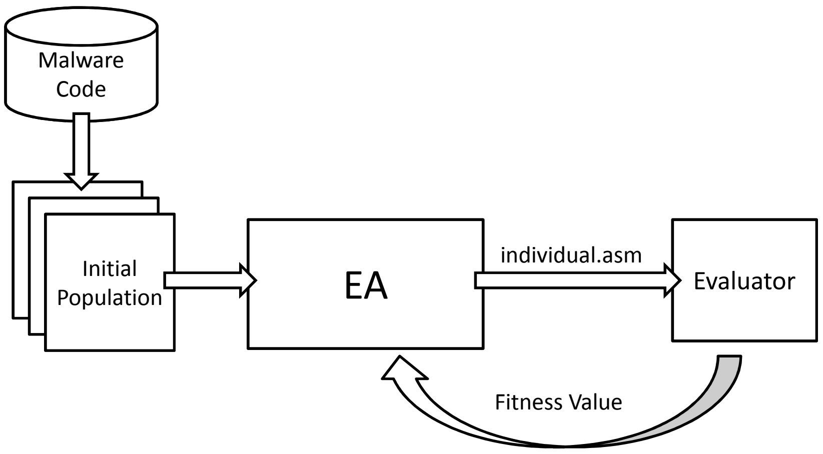


Figure 4.8. Schema of the proposed framework for code generation.

The final goal of the evolution is to create malware not detected by anti-virus applications. But during the evolutionary process it is possible to obtain non-valid programs, unable to compile or not being executed correctly. Moreover, programs that are compiled and executed successfully could lose the characteristics proper of malware, becoming harmless software applications. To drive the evolution towards the creation of malicious applications hard to detect, an individual is awarded a progressively higher fitness value if it satisfies a series of requisites.

Firstly, the assembly code must compile without errors. Then, the obtained executable must run without raising exceptions or fall into infinite loops. Since the system is able to insert any kind of instruction into the code, infinite loops are of course possible: this occurrence is taken into account by forcibly killing candidate programs that do not terminate in a time several orders of magnitude superior to that of the original code. Provided that the program runs correctly, the results of its execution are then checked, to verify that its behavior is still compatible with that of malware. Finally, the candidate malware is analyzed by the scan of a group of anti-virus software. Since the experience is focused on testing the static heuristics only, the chosen anti-virus programs perform the scan without relying on their database. The final fitness value of each individual is proportional to the number of anti-virus application it is able to deceive.

Even more than in other applications of bio-inspired methodologies, the evaluation mechanism closely resembles a synthetic environment. Candidate malware programs represent individuals of a species hunted by anti-virus software. Like living animals, individuals in the EA may evolve features that help them escape the predation. Emerging positive characteristics of this kind are passed on to individuals in the successive generations, thus creating a natural defense strategies against anti-virus predators.

It is interesting to notice that deciding whether an individual still retains the characteristics of malware is a major issue of the evaluator: determining the behavior of a program is in fact a Turing-complete problem [18], thus not approachable in an automated way. As an approximation, a set of heuristics is here used to conclude if a specific program can still be called “malware” with a reasonable probability.

Code Integration

In code integration, the EA is used to determine the optimal position for hiding malicious code inside an existing executable. The goal is to perform the injection preserving both malware’s and host’s functionalities, and with no a-priori information about neither of them.

The EA is used to efficiently explore the search space of possible blocks to replace, probing the target’s code. The potential attacker is interested in finding vulnerable parts as large as possible, and it is important to notice that the search space for blocks of variable size inside a program quickly explodes, even for a relatively small target. Furthermore, finding potential vulnerabilities in compiled software is a task that would involve an intelligent analysis of the target program’s behavior.

In order to empirically evaluate the vulnerability of a certain part of the code, the part is overwritten with other code and the program is run, trying to verify changes in the most common behavior. The injected code is quite simple, few assembly instructions for displaying characters on the screen, nevertheless it is sufficient to discriminate whether and to what extent it is executed.

Every individual in the EA represents a part of the program to be probed, and it is encoded as two integers: the first one (called $offset$) is the offset from the beginning of the compiled code, in bytes; the second (called $size$) is the size of the part, again in bytes (Figure 4.9)

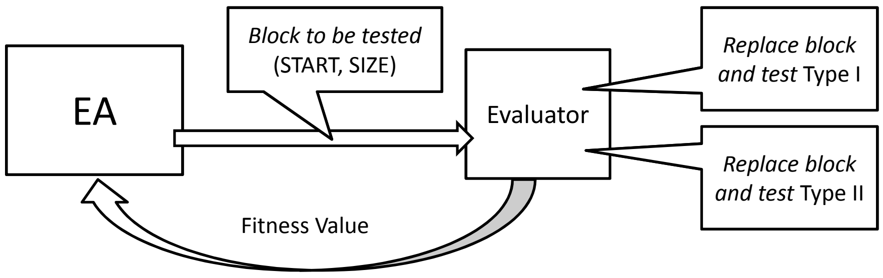

EA
Block to be tested (START, SIZE)
Evaluator
Replace block and test Type I
Replace block and test Type II
Fitness Value

Figure 4.9. Structure of the code integration approach.

The tool distinguishes between two types of areas of potential interest from an attacker's point of view: *Type I* areas represent blocks of code that almost always skipped during a regular execution, like branches after a flow control instructions that are rarely activated; *Type II* areas are usually not processed by the normal flow, and often appear after the end of the main function of the program, like functions that are infrequently called.

The rationale is to use a *Type I* area to inject a vehicle, overwriting the branch with a call – as few as 22 bytes are sufficient to save all registry values, call the malware code, and restore the original values. Then, store the actual malware into one ore more *Type II* areas. The tool does not need any hint, and it is is able to autonomously discriminate between the two types of area observing the behavior of the program after integration.

The EA generates individuals representing blocks, encoded as a vector of integers. The first integer is the starting point of the block, the second is the size. During evaluation, the original block indicated by the individual is overwritten with other code, and the behavior of the program is tested. The evaluator also tries to distinguish between areas of Type I and Type II, with different runs of the code.

4.3.4 Experimental Results

In order to attest their efficacy, the two proposed approaches are experimentally tested. The code generation method is assessed on a real-world malware against different commercial anti-virus software; while the evolutionary code integration is run on two Windows executables.

The two main parameters controlling evolution in μGP are μ and λ. The former is the size of the population: the number of active solutions in each step of the optimization. The latter is the offspring size: the number of new solutions generated in each generation. Almost all other parameters in μGP are self-adapted, that is, the tool tweak the value internally and no user intervention is required.

Code Generation

For the experimental evaluation, the code of the virus *Timid* is inserted into the initial population. Timid is a relatively simple malware, a file infector virus that does not become memory resident, firstly discovered in 1991 and rumored to be an escaped research virus [72]. Each time a file infected with Timid is executed, it copies its own code into other uninfected .COM files in the same directory; if there are no more .COM uninfected files, a system hang occurs. Timid is chosen for the experiment because of several desirable characteristics: despite its age, it still works on the Windows XP operative system; its code is available in the public domain; and its behavior is very predictable and controllable. Thus, checking if the modifications of Timid created by the evolutionary framework still behave as the original malware becomes a relatively straightforward process.

The A86 [58] assembly compiler is chosen for its efficiency in the compiling process. A DOS script kills the individual's process if it takes more than 5 s to complete, thus preventing programs with infinite loops from blocking the evolution. A set of 5 .COM files, taken from those available in the directory `DRIVE:\Windows\system32\`, are used as a test for the infection capability of an individual, their integrity being checked with a md5 cryptographic hash function [84]. An ensemble of 4 different freeware anti-virus applications is selected to verify the ability to escape detection. The chosen anti-virus software shares some desired features: possibility of excluding the database-driven malware detection, performing scans with the heuristic detection only; configurable heuristics, ranging from permissive to distrustful; and a relatively fast scan process. Excluding database-driven detection is a time-saving procedure, since the new malware created by the framework is obviously not included in any database, even if the original code is. The possibility of configuring the heuristics' level of severeness is important to smoothen the fitness landscape, helping the EA to direct the evolution toward the aim. Finally, a fast scan process makes it possible to carry out experiments within a reasonable time limit.

Each experiment, run on a Notebook Intel Centrino running Windows XP Service Pack 2, takes 10 to 15 minutes to complete. Starting from the same initial population, containing only one individual modeling the original code of Timid, 100 runs of the framework are executed. For each experiment, $\mu = 10$, $\lambda = 8$. 96 runs out of 100 reach the maximum possible fitness value, thus producing individuals which compile without errors, are executed correctly, behave like the original malware with regards to several monitored .COM files in the same directory and are not detected by all 4 freeware anti-virus applications. 4 experiments out of 100 are stopped because they produce individuals which delete or corrupt files needed to carry on the evolutionary process.

The successful experiments terminated in an average of 6 generations, with a standard deviation of 2.5. It is worth noticing that the best individual in generation 1 is already able to deceive the heuristics of two of the anti-virus applications; nevertheless, the fitness of the best individual progresses steadily during the generations.

Code Integration

The code integrtion approach is testedon two target executables. Each experiment, run on a Notebook Intel Centrino running Windows XP Service Pack 2, takes about 30 minutes to complete. For each experiment, $\mu = 100$, $\lambda = 30$. The first executable chosen to test the evolutionary code injection approach is `SPLIT.EXE`[3], a small program (46,6 kB) able to split files of any kind into smaller parts or rebuild the original. The EA is set to generate the parameter *offset* in the range (0, 43000), and the parameter *size* in the interval (12, 1000) byte, factually covering all the original code. The first search was stopped after about 10 generations, for a total of 300 individuals evaluated: it reveals 1 zone of Type I and 32 zones of Type II, ranging from 65 to 1511 bytes, thus showing potentially vulnerable positions for an attack.

SPLIT.EXE	
offset interval	(0,43000)
Evaluations	300
Type I (zones found)	1
Type I (largest)	334
Type II (zones found)	32
Type II (largest)	1,511

TESTDISK.EXE			
offset interval	(0,43000)	(0,10000)	(0,2000)
Evaluations	15,000	2,000	300
Type I (zones found)	-	1	1
Type I (largest)	-	33	25
Type II (zones found)	3	4	3
Type II (largest)	179	167	183

Table 4.1: Summary of the experiments for code injections. While SPLIT.EXE shows vulnerabilities even after a first run, several attempts are needed to find exploitable areas in TESTDISK.EXE

[3]See `http://www.iacosoft.com/home/split.txt`

The second executable considered was `TESTDISK.EXE`[4] (315.2 kB), an open-source data recovery software available for different platforms. `TESTDISK` proves to be more resilient to attacks: a first run, with *offset* = (0, 43000) and *size* = (12, 1000) stopped after about 500 generations and 15,000 individuals evaluated, only uncovers only 3 small zones of Type II, ranging from 12 to 179 byte. Since the zones appear to be concentrated in the first part of the executable, subsequent attempts are made, progressively restricting the interested area with *offset*=(0, 10000) and then *offset* = (0, 2000). For these two settings, a zone of Type I if eventually detected. Thus, even if more resilient to attacks, even `TESTDISK.EXE` shows exploitable weak points.

For a complete summary of the experiments, see Table 4.1.

4.4 An Evolutionary Approach for Test Program Compaction

The increasing complexity of electronic components based on microprocessors and their use in safety-critical application - like automotive devices - make reliability a critical aspect. During the life cycle of such products, it is needed to periodically check whether the processor cores are working correctly. In most cases, this task is performed by running short, fast and specialized test programs that satisfies in-field testing requirements. In this work is proposed a method that exploits an evolutionary-computation technique for the automatic compaction of these in-field oriented test programs. The aim of the proposed approach is twofold: reduce execution time and memory occupation, while maintaining the fault coverage of the original test program. Experimental results gathered on miniMIPS, a freely available 5-stage pipelined processor core, demonstrate the effectiveness of the proposed technique.

4.4.1 Introduction

Manufacturers of microprocessor cores included in today embedded systems are not only required to provide their final customers with cheaper, faster, more performant, and less power consuming, but also more reliable devices. This statement is motivated by the fact that microprocessor cores for embedded systems are included in applications that, in these days, are considered safety critical, even if in few years ago the situation was quite different. A paradigmatic case is automotive applications, which are nowadays considered critical applications, since a faulty behavior may provoke a traffic accident leading to several casualties.

[4]See `http://www.cgsecurity.org/wiki/TestDisk`

New standards, such as the ISO 26262 for automotive, and the DO-254 for avionics, provide directions on how to perform periodic tests on microprocessor cores involved in safety-critical applications, in order to increase the dependability of such devices. Clearly, since the periodic testing procedures need to coexist with the actual application, these standards specify constraints regarding execution time, memory occupation, test frequency, and as well as regarding test coverage.

One of the most appropriate strategies for periodically testing processor cores during its normal mode operation (i.e., to perform an *in-field test*) is based on the use of carefully crafted programs. Processor testing through the test programs execution is not a new strategy, in fact, its introduction dates back to 1980 [103], and it is usually referred to as Software-Based Self-Test (SBST). A survey of the most important SBST techniques can be found in [81].

In order to periodically run a SBST test program, the program should be allocated in a memory accessible by the processor core; then, once the test is scheduled to be executed, the processor stops the execution of the actual application, and then runs the test program collecting, at the end the testing results. Interestingly, strategies based on SBST do not require modifying the processor core, and are easily exploited also during the normal mode operation of the processor core.

Several strategies have been proposed for developing test programs, as described in [81]. These are based on different techniques that exploit manual, deterministic, random, and also automatic strategies that usually aim at detecting the most of the processor faults, neglecting however other constraints that deserve more attention when considering in-field testing, such as the actual test program execution time and code size.

In this work, we propose a new compaction procedure able to better fit in-field constraints for processor cores in embedded systems by reducing the execution time as well as the memory occupation of test programs. The main idea behind the proposed strategy is based on the experimental evidence that in most of the test programs there exist significant testing redundancy on the different instructions involved in the test programs. We develop an evolutionary-based strategy able to discriminate which of the instructions in the test program is not significantly contributing on the testing goals, and then, it is possible to remove from the test program without affecting the fault coverage but decreasing in this way the test program execution time and the memory occupation.

In order to experimentally evaluate the proposed approach, we apply our strategy to a set of test programs developed using different strategies and targeting a couple of modules inside a freely available pipelined processor core, miniMIPS [1]. The considered units are the decode unit, and the forwarding logic.

4.4.2 Background

Test of electronic components is a critical phase in order to guarantee the correct behavior of each embedded component. Manufacturing test allows checking the correct fabrication of a circuit: it is applied before beginning its operational mode and permits the detection of faults prior distributing the component and putting it on his final application. However, during the normal chip's operational mode, there is the possibility to come across operational faults; to improve reliability of embedded systems and detect such faults, it is necessary to perform periodic tests during the operational mode. Built-In Self-Test (BIST), as described in [6], is a very powerful technique, able to achieve the twin goals of high fault and error coverage with low error latency. Unfortunately, the BIST techniques require additional hardware to be implemented and, consequently, give back an increase in costs, power consumption and chip area. In addition, BIST techniques are not always suitable for in-field testing.

Software-Based Self-Test (SBST) techniques are alternative BIST ones; SBST are important low-cost test solutions that allow testing of electronic components without hardware overhead, and without increasing the power consumption during the test execution [48]; the efficacy of SBST depends on the quality of the test programs.

In [81], the most important strategies for generating test programs are described. These techniques can be divided in two main groups: *formal* and *simulation-based*. Through the formal approach, test programs are implemented using mathematical techniques to prove some properties considering all possible inputs on the circuit. Simulation-based techniques instead use a set of stimuli to discover any improper behavior of the microprocessor that executes them.

For example, the VERTIS tool [3] is able to generate sequences of assembly instructions for both functional design validation and manufacturing test. The generated test program is based on the enumeration of all the possible combinations for every instruction: this approach achieves good result in coverage of stuck-at faults, but leads to very large test programs.

FRITS [76] is another generator that combines random generation of instruction sequences with pseudorandom data. This tool is implemented through a cache-resident approach; the program has to be loaded within the cache and, to work correctly, it cannot produce any cache miss or bus cycle. FRITS has been implemented on Intel's Itanium processor family and Pentium 4 line of microprocessors achieving 70% stuck-at fault coverage for the entire chip.

Facing the problem of generating high-quality test programs through coverage-driven approaches, exploiting simulation feedback, in [29] [27] authors implement a system based on an evolutionary algorithm: the EA has been used to select instructions defined within the processor's ISA and data values, with the aim of creating test programs with high fault coverage. Experimental results of these works show the achievement of about the 86% of fault coverage on an 8051 microprocessor and that this automatic technique can be successfully applied on complex pipelined architecture such as SPARC v8.

Regarding on-line SBST, in [48], the author describes three precise goals that a test program should pursue to be considered as a good one:

- Small memory footprint;
- Small execution time;
- Low power consumption.

An obvious disadvantage of tests applied during operational mode is the amount of time subtracted to normal running applications and the memory space allocated to contain the test program and its values.

Analyzing an existent test program in order to make it compliant with the previously defined properties is a very difficult task, that has to be performed by experts in a manually and tedious way. As described in [83], the reduction of the test data volume is one of the most significant constraints to minimize the test costs, because it means a reduction of test time and small memory requirements. Consequently, compaction of a test program seems to be an affordable approach to reduce the impact of the test routines on the running application programs.

In [87] authors describe a system based on Genetic Programming with the same aim of reduce time execution and memory occupation of test programs presented on this work, but focusing their approach on compacting test programs written in the form of a loop.

4.4.3 Proposed Approach

Regardless the original generation methods, the most of the test programs contain instructions that do not directly contribute to the program testing goal. For example, in random based approaches as well as in deterministic ones , many redundant instructions are placed in the program. Even though these instructions do not provide any testing feature, these are not removed before the final delivery. Practical evidence shows that in manual test, these instructions are also present, even though the test engineer tries to reduce the number of irrelevant instructions written in the program. As a matter of fact, in a final phase of the test program development, the test engineer can try to reduce the test program size as well as the execution time and memory occupation by identifying and eliminating the aforementioned irrelevant instructions.

The proposed flow is depicted in Figure 4.10. Initially, during the *Test program collection* phase, the *Original processor Test Set* should be sifted in order to only select the test programs that mainly target the module under consideration. Then a fault simulation is performed using the targeted fault list, which includes only the module fault list and considers the selected fault model (e.g., stuck-at faults, or transition delay faults). This phase delivers the *Initial Set* that is composed of a limited number of test programs, and a detailed fault list that gathers all the information regarding the initial fault coverage (FCi %) results.

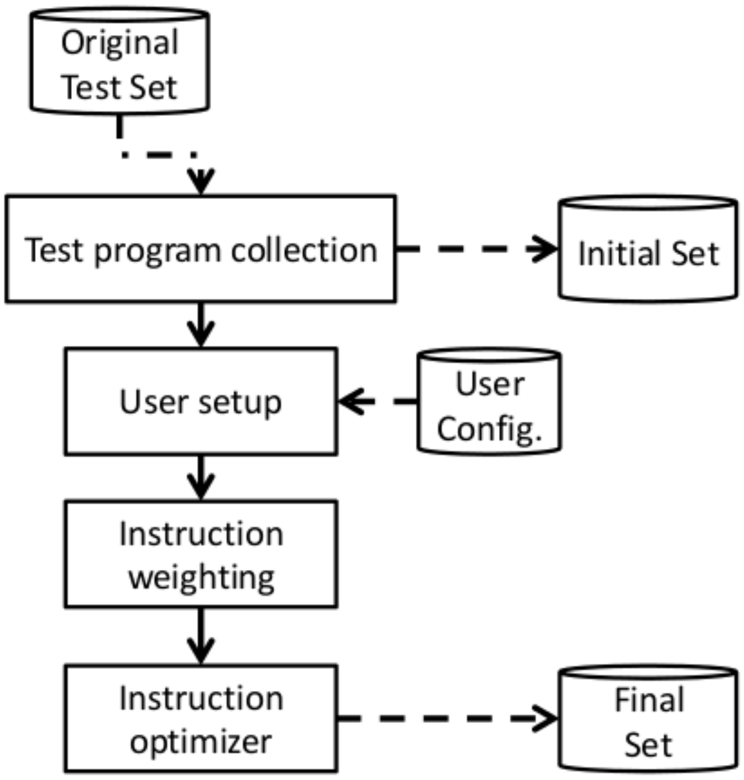


Figure 4.10. Program compaction flow

During the second phase, the *User setup* one, the following parameters are defined by the user:

- Δ *Tolerance*: This parameter defines the number of faults or the percentage of FC that the user may tolerate to lose during the compaction process. In most of the cases this value should be zero; however, in some cases it is affordable to lose a few of faults on the final fault coverage in order to obtain a better compaction. In particular, this parameter allows the optimization process to generate test programs that may cover a slightly lower number of faults than the original one;

- Δ *Instructions*: Minimum number of instructions to be eliminated, defined as a percentage or number of instructions;

- *Time limit* (wall-clock time limit): This parameter defines the maximum time available for performing the optimization process.

The third step consists on providing to every instruction in the test programs with a weight that indicates the probability of one instruction to be eliminated from the test program. For example, memory access related instructions (e.g., LS ST) required to be weighted with a low value, since in most of the cases accesses to the memory represent the most important instructions regarding observation. This step allows the user to identify instructions and accordingly to its personal experience to decide the actual possibilities to be removed from the test set.

The final step, called *Instruction optimizer*, consists in an optimization process in charge of removing instructions from the current test program. In order to ease this process, we associate each instruction of the program with a Boolean variable that represent whether the instruction must be kept or not in the final test program. Thus, every instruction is then described by using two values: the instruction weight, and the instruction availability.

Once the test program is represented as a string of weights and availability, the next step consists in the actual optimization process that will try to eliminate the most of the instructions according to the instruction probabilities while maintaining the expected FC% results.

We used an evolutionary optimizer called μGP (MicroGP). The tool was developed at Politecnico di Torino in 2000, and is freely available under the GNU Public License [88]. μGP represents individuals, in this case formed by a bitstring, as a directed graphs, in which every element can be characterized by one or more tags. Tags are useful both to identify elements, and to add semantic information useful to drive the evolution; for example, in this case, the instruction weight is included as a tag in order to determine the instruction probability to be removed or not. During the first step of the evolution, a random population of individuals is created; afterwards, applying the typical operators of mutation and recombination, individuals are optimized in order to achieve the desired goal. Every new individual is evaluated resorting to an external evaluator that returns the evolutionary optimizer the goodness of the evaluated individual. Then, resorting to these values, individuals are ordered and the best ones are maintained in the population, while the others are discharged from the population. Then, a new evolutionary step starts again trying to improve the remaining individuals in the population.

In [98] author presents a methodology based on μGP to devise assembly programs suitable for a range of on-silicon activities, demonstrating the potentiality of this approach performing experimental evaluation on two high-end Intel microprocessors.

The values suitable to determine goodness in the presented experiments are the following:

- a) **FC%**: the generated individuals must target FCi% - Δ Tolerance
- b) **Number of eliminated instructions**: the minimum target is Δ Instructions
- c) **1 / execution time** (μGP always maximizes the goodness)

The evolutionary optimizer can be configured in such a way to compact test program according to one of the following goals:

- **instructions number**: the optimization is driven in order to reduce as much as possible the number of instructions forming the best optimized program; this strategy can be implemented by selecting the values a) and b) as the goodness to be maximized;
- **execution time**: the optimization process faces the maximum reduction in terms of execution time required by the test program; this goal may differ respect to the one described in the previous point(e.g., when original test programs contains loops); in this case an appropriate goodness consists in the values a) and c).

The strategy selected to compact programs in this work was to optimize the instruction number. During the experiments, we inserted within the random population an individual with all bits set to 1; it means a configuration selecting the whole instructions in the test program. Thus, it was possible to start the evolution process with one candidate solution already yielding the goal desired.

4.4.4 Case Study and Experimental Results

The feasibility of the proposed approach and its compaction capacity was verified applying the strategy described in the previous sections to compact test programs, in which the effort is concentrated on testing two interesting modules of the miniMIPS [1] processor: the forwarding and interlock unit, and the decode unit.

The miniMIPS architecture is based on 32-bit buses and includes a 5-stage pipeline. The RT-level description was synthesized with Synopsys Design Compiler version H-2013.03-SP5-4 targeting an in-house developed library. The resulting gate-level netlist contained 16,303 gates and 1,967 flip-flops, corresponding to 115,508 stuck-at faults.

The framework implemented to allow the use of μGP was configured as follow: the EA tool creates an individual constituted by a vector of boolean mask with as many elements as the number of instructions constituting the original test program. Then, a new program is created including the instructions represented as true. The program obtained by this operation consists on a subset of instructions of the original one, which is ready to be simulated and fault simulated by targeting the faults of the module under consideration.

The fitness values required by the evolutionary optimizer are calculated resorting to a logic simulator and a fault simulator. The logic simulation was performed resorting to ModelSim SE version 6.6d; whereas the fault simulations, determining the FC% achieved by every test program, were performed using TetraMAX version H-2013.03-SP1. The number of faults covered by an individual may increase due to the particular architecture of the modules under test. This particular eventuality is not considered as an improvement, thus, the number of faults covered does not include the faults that overtake the fault coverage of original test program.

For every one of the processor modules we targeted in this work, three different set of test programs were gathered. These test set were obtained through three different generation approaches:

- genetic programming (GP)
- manual optimization (M)
- random (RND)

Considering that instructions performing memory access are the most important ones to observe values on buses permitting to check the correctness of signals in transit, we set parameters in such a way to impose a low probability to eliminate them. In particular, we set the probability to remove LS and ST instructions to 0.25; removal probability for the rest of instructions was set to 0.75.

Experiments described in this work were executed on a workstation based on 2 Intel Xeon E5450 CPUs running at 3.00 GHz, equipped with 8 GB of RAM. On this machine, the duration of a complete simulation in terms of wall-clock time is between 6 and 12 seconds for each run, mostly dedicated to fault simulation; to speed-up the process, experiments were parallelized on all the 8 available cores. As usually happens in industrial applications, the real time limit is defined by the wall-clock; according to this constraint, we set the time limit parameter for each compaction process to 10 hours.

Forwarding and Interlock Unit

The forwarding and interlock unit is a hardware solution to deal with data hazards; the methods employed to avoid data hazards mainly consist in checking if there is a data dependency between two instructions simultaneously present in different pipeline stages, and take suitable counteractions accordingly; example:

```
(1) ADD R1, R2, R3 (2) ADD R4, R1, R5
```

For clarity purposes, let us consider the example above where we call the first instruction entering the pipeline instruction (1) and the instruction arriving later, which has a data dependency with the first one, instruction (2). In this case forwarding must be activated: the input data for instruction (2), instead of coming from the register file, is directly taken from the stage where instruction (1) produces it. In case instruction (1) is not yet in that stage, interlock is used. Interlock implies the pipeline is stalled until instruction 1 reaches the stage in which the data is produced. At that moment, forwarding is used to send the result of instruction (1) to the stage where instruction (2) needs its operands.

The forwarding and interlock unit inside the miniMIPS processor is a unit used within the execution stage, consisting in a combinatorial block of 551 gates, counting 3,738 stuck-at faults.

The test programs used as benchmark for this unit are characterized by similar fault coverage values, but a different number of instructions due to different approaches used for producing them: the *GP* program, generated through genetic programming, covers 3,237 faults (86.6 %) counting with 456 instructions, and requiring 568 clock cycles to execute the test program. The *M* program, obtained by an expert test engineer through manual optimization, covers 3,240 faults (86.7%) using 243 instructions; to be executed, this test programs needs 382 clock cycles. And finally, we created a third test set using a random approach that produces the *RND* test program that counts with 2,000 instructions, covers 2,861 faults (76.5%); due the greater number of instructions within it, the running time is 2,325 clock cycles.

Table 4.2 presents the results obtained by applying the compaction approach (*EC Compactions*) described in the previous sections, targeting the test programs for testing the forwarding unit module. For this first experiment, we set a Δ Tolerance equal to zero, with the aim of compacting test program without losing any faults excited by the original version; Δ Instructions parameter instead was set equal to 10%. The value of minimum number of instructions to be eliminated was chosen not too strict, due that compaction possibilities of a test program are not known at the beginning of optimization. For the sake of comparison, we implement also a random approach that eliminates instructions in a random fashion, and the results on these experiments are also shown the Table 1 lines labeled as *Random Compaction*.

The reader may notice while observing Table 4.2, that the evolutionary-computation technique has a high capacity of compaction, and improves the quality of the test programs reducing number of instructions and the needed time to execute them, without affecting the final fault coverage.

		GP Program	*M Program*	*RND Program*
Fault Coverage %	Original TP	86.6	86.7	76.5
	EC Compaction	0	0	0
	Random Compaction	0	0	0
Size [instructions]	Original TP	456	243	2000
	EC Compaction	-16.0%	-46.1%	-26.1%
	Random Compaction	-0.4%	-5.4%	-0.8%
Duration [cc]	Original TP	569	382	2324
	EC Compaction	-13.7%	-38.5%	-26.4%
	Random Compaction	-0.5%	-3.7%	-0.7%

Table 4.2: Compaction of test programs for the forwarding unit

Interestingly, the random approach shows low compaction capacities, due also to the strong constraints to conserve the same faults coverage of the original test programs; this obligation makes valid only a small part of programs obtained by casual compaction, despite the creation of the same number of programs generated during the evolutionary compaction approach.

Figure 4.11 shows program size and program duration evolution for all the three experiments related to the forwarding unit. Even though our primary goal is to minimize the program size, as a side-effect a similar trend is obtained also for program duration in all the three cases.

Decode Unit

The decode unit is a module present in microprocessor that interprets the instruction codes and figure out what are operations to be applied to data; it is a module working within the decode stage and, on the implementation used of the miniMIPS processor, it counts 7,502 stuck-at faults that refer to 977 gates and 203 flip-flops.

Similarly to the approach described in the previous section for the forwarding unit, benchmark test programs implemented for testing the decoding hardware are similar by faults coverage. Test programs adopted in this work are: program *GP*, obtained through genetic programming, that covers 5,971 faults (79.6%) with 447 instructions and 1,231 clock cycles of execution time; program *M*, handwritten by experts, that covers 5,792 faults (77.2%) using 258 instructions and 2,328 clock cycles of running period, and a random one formed by 2000 instructions covering 5,699 faults (76%), called program *RND*, that needs 2,325 clock cycles for executing it.

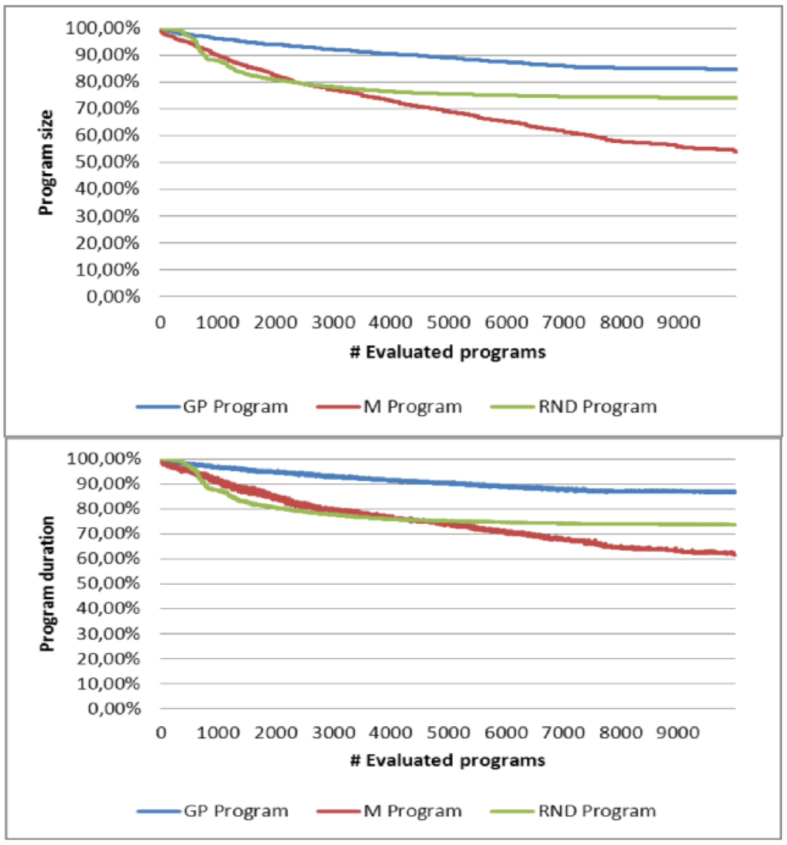


Figure 4.11. Forwarding and interlock unit program size and duration evolution

The application of compaction techniques to the decode module was a quite hard task. Application of random and evolutionary compaction techniques it has proved an unsuccessful process: with both approaches it was not possible to compact programs without losing in number of covered faults. To obtain significant results, it was enough to insert a tolerance in total number of faults; the Δ Tolerance parameter was set to 1%, in order to admit a small leak in coverage and permitting a satisfying reduction in number of instructions. Thus, similarly to what we did in the previous experiments for the forwarding unit program compaction, the Δ Instructions parameter was set to 10%. Table 4.3 shows compaction achieved with this new constraint.

		GP Program	*M Program*	*RND Program*
Fault Coverage %	Original TP	79.6	77.2	76.0
	EC Compaction	-0.8	-0.7	-0.4
	Random Compaction	-0.6	-0.7	-0.8
Size [instructions]	Original TP	447	258	2000
	EC Compaction	-27.3%	-70.9%	-52.0%
	Random Compaction	-1.8%	-24.0%	-28.4%
Duration [cc]	Original TP	1231	2328	2325
	EC Compaction	-11.4%	-37.1%	-51.0%
	Random Compaction	-1.1%	-71.8%	-27.8%

Table 4.3: Compaction of test programs for the decode unit, with 1% of faults lost

Values in Table 4.3 confirms the method suitability; moreover, due to the introduction of the Δ Tolerance parameter, the size compaction results are slightly better than the ones obtained previously. The Evolutionary-Computation based compaction algorithm works better than a random compaction and was able to reduce test programs length up to 70% in the case of handwritten program.

Figure 4.12 depicts the evolution of the program size and duration concerning the decode unit experiments. The reader can observe a sharp improvements for the *RND* program; this is a typical trend of evolutionary optimizers, which represents an escape from local maximum values.

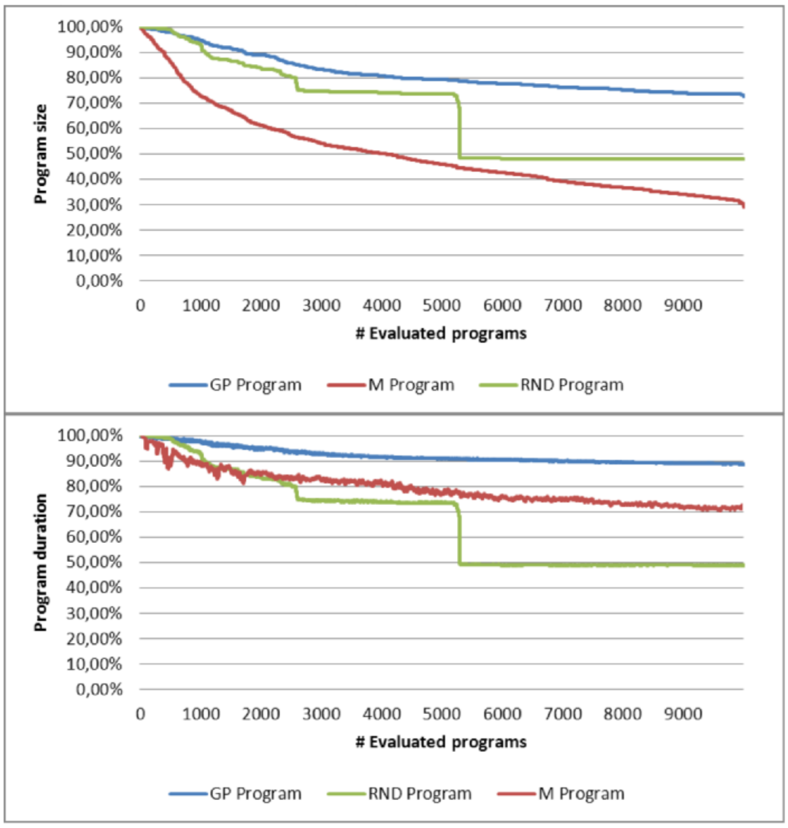


Figure 4.12. Decode unit program size and duration evolution

Chapter 5

Conclusions

The goal of this thesis is to design and implement new EA technologies for improving the μGP tool in its industrial-grade optimization capabilities. In more details, this thesis show new approaches for the selection of optimal genetic operators during different phases of the evolution; the definition of a new metric for calculating distance between individuals and able to guarantee the diversity preservation within the population; and a new evolutionary approach that use a cooperative evolution to group individuals in order to optimize a set of sub-optimal solutions. Then, the thesis reports four different successful industrial scenarios: generation of on-line test programs; compaction of on-line test programs; design of wetlands. Finally, a prototypical application is shown: automatic creation of malware.

The extensive experimental evaluation demonstrates that the proposed approach to a cooperative evolution for the generation of functional test programs is effective in achieving a high fault coverage on the targeted modules, while maintaining essential characteristics for on-line testing. The approach consists in an automatic strategy for the generation of functional test programs oriented to on-line testing of processor cores. The proposed strategy requires low human intervention, as well as low knowledge about the processor core or the modules under test.

In the experiments on group evolution, two common modules available in a pipelined processor core were tackled: the address calculation module and the forwarding unit. Both of the modules were approached using the proposed strategy, and the final coverage results experimentally corroborate the suitability of the proposed approach.

Experiments on the distance for Linear Genetic Programming show a prominent correlation between the proposed method and phenotypic problem-specific distance measurements in two samples where individuals are radically different, NK-landscapes and Assembly language generation. The methodology is then successfully applied to two experiments with fitness sharing, showing again results comparable to more complex and computationally more demanding phenotypic fitness metrics. Following the same general principles, a similar distance metric could also be defined for classical Genetic Programming, using for example a node and its children in place of n-grams. However, further studies are needed to assess the general validity of the proposed approach. Variations of the methodology need to be conceived in order to tackle individuals composed of both real values and constants, or individuals in combinatorial problems. Also, since the proposed approach relies upon the number of symbols encoded in each individual, its use for diversity preservation might implicitly benefit larger individuals, thus possibly contributing to the known issue of *bloating*.

Experiments performed for the extension of the Dynamic Multi-Armed Bandit approach, for the genetic operators selection, shows that was possible to combine the performance benefits of the standard DMAB strategy, with the added benefit of simplifying its usage, removing the need to select operators beforehand, and relaxing the constraints on operator definitions, as operators can fail sporadically. Our simulation and experimental results on two benchmarks demonstrate that the extension indeed yields in all setups a performance at least equal to the DMAB approach for specific problems and clearly better in other cases.

Moreover, applications discussed in Chapter 4 clearly show the adaptability of evolutionary algorithms in different optimization contexts, due to the significant differences between the fields of application.

In the application on wetlands optimization, results achieved using evolutionary algorithm as optimizer are encouraging. Optimizing the wetland design is an extremely complex task, and it is currently carried on by experts using a trial-and-error approach on the basis of fluid-dynamics simulations. In this application, an evolutionary algorithm is applied to the wetlands design problem. Each candidate solution is evaluated by a state-of-the-art fluid-dynamics simulator, on the basis of several relevant metrics. Th application of evolutionary algorithm based optimization show a performance comparable with human-devised designs, despite the absence of human intervention during the optimization process.

Slightly different is the application of EAs in anti-malware research. While the research is still at an early stage of development, its potential is apparent. An evolutionary toolkit was used first to devise a new malware, modifying and tweaking existing code to escape detection, then to find holes in an existing executable to hide malicious code in. Future works will focus on the development of an automated procedure to model malware code in the internal representation of the EA; on the use of even more commercial anti-virus programs to improve the final applications produced by the framework; and on the development of general heuristics able to determine whether the programs produced by the framework still retain the malicious characteristics of the original software source, even when more distinct malware applications are present in the first population. Finally, the two approaches will be merged in a single one, able to automatically create new code injectors.

The last application described in Chapter 4, instead, presents a new automatic approach to perform test program compaction. Applying compaction technique based on an evolutionary algorithm, it is possible to decrease execution time and memory requirements of existing test programs without losing in fault coverage.

Results presented on this work show the efficacy of the approach, underlining the concept that it is possible to compact test programs generated through different techniques: genetic programming, manual and random. In addition, the approach based on EA is compared with a fully random one, showing higher compaction performance, confirming that our proposed method is more effective than a purely random one.

Observing obtained results, compared with those obtained by previous versions of μGP , the potential of new technologies is encouraging. Starting from this point, I intend to focus my future research activities belonging the following directions: *Group Evolution*: this technology, presented in this thesis, is still under development. Many inspections on the potential of the genetic operators are still required; *Aging*: The idea is to apply some theories of aging of humans in the evolutionary algorithms field, by allowing crossovers among young individuals and mutations on older ones; in this way could be possible to force exploration and exploitation phases on each individual.

Appendix A

Acronyms

ALU: Arithmetic and Logic Unit

BIST: Built-In Self-Test

CAD: Computer-Aided Design

CCEA: Cooperative Co-Evolutions Algorithms

CISC: Complex Instruction Set Computer

CPU: Central Processing Unit

DAME: Dark Angel's Multiple Encryptor

DIS: Draft International Standard

DMAB: Dynamic Multi-Armed Bandit

DOS: Disk Operating System

EA: Evolutionary Algorithms

EC: Evolutionary Computation

EP: Evolutionary Programming

ES: Evolutionary Strategies

FC: Faults Coverage

FRITS: Functional Random Instruction Testing at Speed

GA: Genetic Algorithms

GNU: GNU's Not Unix

GP: Genetic Programming

ISA: Instruction Set Architecture

ISO: International Organization for Standardization

IT: Information Technology

LGP: Linear Genetic Programming

LRU: Least Recentlty Used

MAB: Multi-Armed Bandit

MIMD: Multiple Instruction Multiple Data

MtE: The Mutation Engine

NSGA: Nondominated Sorting Genetic Algorithm

NP: Nondeterministic Polynomial time

OS: Operating System

PDMAB: Parallelized Dynamic Multi-Armed Bandit

PE: Portable Executable

RAM: Random Access Memory

RISC: Reduced Instruction Set Computer

ROM: Read Only Memory

RTL: Register Transfer Level

SBST: Software-Based Self-Test

SPARC: Scalable Processor ARChitecture

TGP: Tree-based Genetic Programming

TPE: TridenT Polymorphic Engine

UCB: Upper Confidence Bound

UID: Universal Information Distance

VCD: Value Change Dump

VCL: Virus Creation Laboratory

XML: eXtensible Markup Language

Appendix B

List of Publications

- M. Gaudesi, M. Jenihhin, J. Raik, E. Sanchez, G. Squillero, V. Tihhomirov, R. Ubar *Diagnostic Test Generation for Statistical Bug Localization using Evolutionary Computation.* In: EVOSTAR - The Leading European event on Bio-Inspired Computation, Granada, Spain, 23-25 April 2014.

- Di Carlo S., Gaudesi M., Sanchez E., Sonza Reorda M. (2014) *A Functional Approach for Testing the Reorder Buffer Memory.* In: JOURNAL OF ELECTRONIC TESTING, vol. 30 n. 4, pp. 469-481. - ISSN 0923-8174

- M. Gaudesi, S. Saleem, E. Sanchez, M. Sonza Reorda, E. Tanowe (2014) On the In-Field Test of Branch Prediction Units using the Correlated Predictor mechanism. In: IEEE 17th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, Warsaw, Poland, April 23-25.

- Gaudesi M., Piccolo E., Squillero G., Tonda A. (2014) *TURAN: Evolving non-deterministic players for the iterated prisoner's dilemma.* In: 2014 IEEE Congress on Evolutionary Computation (CEC), Beijing, China, 06 - 11 July, 2014. pp. 21-27

- Cani A., Gaudesi M., Sanchez E., Squillero G., Tonda A. (2014) *Towards Automated Malware Creation: Code Generation and Code Integration.* In: 29th Symposium on Applied Computing - SAC 14, Gyeongju, Korea, March 24 - 28, 2014. pp. 157-158

- Marco Gaudesi, Giovanni Squillero, Alberto Tonda (2014) *Universal information distance for genetic programming.* In: GECCO '14 - Genetic and Evolutionary Computation Conference, Vancouver, BC, Canada, July 12-16, 2014. pp. 137-138

- Gaudesi M., Squillero G., Tonda A. (2013) *An Efficient Distance Metric for Linear Genetic Programming.* In: GECCO 2013 - Genetic and Evolutionary Computation Conference, Amsterdam, The Netherlands, July 06-10, 2013. pp. 925-932

- Gaudesi M., Marion A., Musner T., Squillero G., Tonda A. (2013) *An Evolutionary Approach to Wetlands Design.* In: 11th European Conference on Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics, EvoBIO 2013, Vienna (Austria), 3-5 April 2013. pp. 177-187

- M. Gaudesi, A. Marion, T. Musner, G. Squillero, A. Tonda (2013) *Evolutionary Optimization of Wetlands Design.* In: 28th Annual ACM Symposium on Applied Computing, SAC 2013, Coimbra, Portugal, 18 - 22 Marzo 2013. pp. 176-181

- Ciganda L., Gaudesi M., Lutton E., Sanchez E., Squillero G., Tonda A. (2012) *Automatic Generation of On-Line Test Programs through a Cooperation Scheme.* In: 13th International Workshop on Microprocessor Test and Verification (MTV), 2012, Austin TX, USA, 10-13 Dec. 2012 . pp. 13-18

Definitely Accepted – Currently in Press

- R. Cantoro, M. Gaudesi, E. Sanchez, P. Schiavone, G. Squillero *An Evolutionary Approach for Test Program Compaction.* In: Latin American Test Symposium (LATS), 2015.

- N. Palermo, V. Tihhomirov, T. S. Copetti, M. Jenihhin, J. Raik, S. Kostin, M. Gaudesi, G. Squillero, M. Sonza Reorda, F. Vargas *Rejuvenation of Nanoscale Logic at NBTI-Critical Paths Using Evolutionary TPG.* In: Latin American Test Symposium (LATS), 2015.

Awaiting Final Decision

- M. Gaudesi, E. Piccolo, G. Squillero, A. Tonda *Exploiting Evolutionary Modeling to Prevail in Iterated Prisoner's Dilemma Tournaments.* In: Transactions on Computational Intelligence and AI in Games, 2015 (minor review, re-submitted).

- J. Belluz, M. Gaudesi, G. Squillero, A. Tonda *Operator Selection using Improved Dynamic Multi-Armed Bandit.* In: GECCO 2015 - Genetic and Evolutionary Computation Conference, Madrid, Spain, July 11-15, 2015).

- R. Cantoro, M. Gaudesi, E. Sanchez, G. Squillero *Exploiting Evolutionary Computation in an Industrial Flow for the Development of Code-Optimized Microprocessor Test Programs.* In: GECCO 2015 - Genetic and Evolutionary Computation Conference, Madrid, Spain, July 11-15, 2015).
- M. Gaudesi, M. Sonza Reorda, I. Pomeranz *On Test Program Compaction.* In: ETS 15 - 20th IEEE European Test Symposium, Cluj-Napoca, Romania, May 25-29. 2015.

Bibliography

[1] miniMIPS processor, available at http://opencores.org/project,minimips.

[2] Symmetric Difference, in: E. J. Borowski, J. M. Borwein (eds.), The Harper-Collins Dictionary of Mathematics, HarperCollins, 1991.

[3] Native mode functional test generation for processors with applications to self test and design validation (Oct. 1998).

[4] M. Ahluwalia, L. Bull, Co-evolving Functions in Genetic Programming: Dynamic ADF Creation using GLiB, in: V. W. Porto, N. Saravanan, D. Waagen, A. E. Eiben (eds.), Evolutionary Programming VII: Proceedings of the Seventh Annual Conference on Evolutionary Programming, vol. 1447 of LNCS, Springer-Verlag, Mission Valley Marriott, San Diego, California, USA, 1998, pp. 809–818.

[5] C. Akratos, V. Tsihrintzis, Effect of temperature, HRT, vegetation and porous media on removal efficiency of pilot-scale horizontal subsurface flow constructed wetlands, Ecological Engineering 29 (2) (2007) 173–191.

[6] H. Al-Asaad, B. T. Murray, J. P. Hayes, Online BIST for Embedded Systems, Journal IEEE Design & Test vol. 15 (4) (1998) 17–24.

[7] F. Arega, B. F. Sanders, Dispersion Model for Tidal Wetlands, Journal of Hydraulic Engineering 130 (8) (2004) 739–754.

[8] P. Auer, N. Cesa-Bianchi, P. Fischer, Finite-time analysis of the multiarmed bandit problem, Machine learning 47 (2) (2002) 235–256.

[9] D. Augustijn, F. Huthoff, E. Velzen, Comparison of vegetation roughness descriptions.

[10] J. Bagley, The behavior of adaptive systems which employ genetic and correlation algorithms, Ph.D. thesis, University of Michigan (1967).

[11] W. Banzhaf, P. Nordin, R. E. Keller, F. D. Francone, Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications (The Morgan Kaufmann Series in Artificial Intelligence).

[12] J. Bauer, J. Michel, Y. Wu, ITU Study on the Financial Aspects of Network Security: Malware and Spam, ICT Applications and Cybersecurity Division, International Telecommunication Union, Final Report.

[13] G. Bendoricchio, S. E. Jorgensen (eds.), Fundamentals of Ecological Modelling,

Third Edition, 3rd ed., Elsevier Science, 2001.

[14] P. Bernardi, L. Ciganda, M. D. Carvalho, M. Grosso, J. Lagos-Benites, E. Sanchez, M. S. Reorda, O. Ballan, On-Line Software-Based Self-Test of the Address Calculation Unit in RISC Processors, in: European Test Symposium (ETS), 2012 17th IEEE, 2012, pp. 1–6.

[15] P. Bernardi, M. Grosso, E. Sanchez, O. Ballan, Fault grading of software-based self-test procedures for dependable automotive applications, in: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011, IEEE, 2011, pp. 1–2.

[16] G. E. P. Box, Evolutionary operation: A method for increasing industrial prouctivity, Applied Statistics VI, no. 2 (1957) 81–101.

[17] T. Bradley, Zero Day Exploits: Holy Grail Of The Malicious Hacker, http://netsecurity.about.com/od/newsandeditorial1/a/aazeroday.htm.

[18] W. S. Brainerd, L. H. Landweber, Theory of Computation, John Wiley & Sons, Inc., New York, NY, USA, 1974.

[19] M. F. Brameier, W. Banzhaf, Linear genetic programming, Springer, 2007.

[20] H. J. Bremermann, Optimization through Evolution and Recombination, Spartan Books, 1962.

[21] A. Cani, M. Gaudesi, E. Sanchez, G. Squillero, A. Tonda, Towards Automated Malware Creation: Code Generation and Code Integration, in: Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC '14, ACM, New York, NY, USA, 2014, pp. 157–160.

[22] W. D. Cannon, The Wisdom of the body, W.W.Norton, 1932.

[23] R. Cantoro, M. Gaudesi, E. Sanchez, P. Schiavone, G. Squillero, An Evolutionary Approach for Test Program Compaction, in: (in press), 2015.

[24] T. M. Chen, J. Robert, The Evolution of Viruses and Worms, in: Statistical Methods in Computer Science, 2004.

[25] L. Ciganda, M. Gaudesi, E. Lutton, E. Sanchez, G. Squillero, A. Tonda, Automatic Generation of On-Line Test Programs through a Cooperation Scheme, in: 13th International Workshop on Microprocessor Test and Verification (MTV), 2012, pp. 13–18.

[26] F. Cohen, Computer viruses : Theory and experiments, Computers & Security 6 (1) (1987) 22–35.

[27] F. Corno, E. Sánchez, M. S. Reorda, G. Squillero, Automatic Test Program Generation: A Case Study., IEEE Design & Test of Computers 21 (2) (2004) 102–109.

[28] F. Corno, E. Sánchez, G. Squillero, Evolving assembly programs: how games help microprocessor validation, Evolutionary Computation, IEEE Transactions on 9 (6) (2005) 695–706.

[29] F. Corno, M. Sonza Reorda, G. Squillero, M. Violante, On the test of microprocessor IP cores, in: Proceedings of Design, Automation and Test in Europe

Conference and Exhibition 2001, IEEE Press, Munich, Germany, 2001, pp. 209–213.

[30] G. Corriveau, R. Guilbault, A. Tahan, R. Sabourin, Review and Study of Genotypic Diversity Measures for Real-Coded Representations, IEEE transactions on evolutionary computation 16 (5) (2012) 695–710.

[31] L. M. Cowardin, Classification of Wetlands and Deepwater Habitats of the United States, DIANE Publishing, 1979.

[32] L. DaCosta, Á. Fialho, M. Schoenauer, M. Sebag, Adaptive Operator selection with dynamic multi-armed bandits, in: G. Rudolph, T. Jansen, S. M. Lucas, C. Poloni, N. Beume (eds.), PPSN, vol. 5199 of Lecture Notes in Computer Science, Springer, 2008.

[33] R. Dawkins, The Selfish Gene, Oxford University Press, 1982.

[34] K. Deb, D. E. Goldberg, An investigation of niche and species formation in genetic function optimization, in: Proceedings of the 3rd International Conference on Genetic Algorithms, Morgan Kaufmann Publishers Inc., 1989, pp. 42–50.

[35] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, Evolutionary Computation, IEEE Transactions on 6 (2) (2002) 182–197.

[36] J. Doe, J. Roe, J. Bloggs, Evolutionary Optimization of Wetlands Design, in: 28th Symposium On Applied Computing (SAC), 2013.

[37] A. E. Eiben, J. E. Smith, Introduction to evolutionary computing, vol. 2, Springer Berlin, 2010.

[38] J. Elder, The dispersion of marked fluid in turbulent shear flow, J. Fluid Mech 5 (4) (1959) 544–560.

[39] Á. Fialho, L. D. Costa, M. Schoenauer, M. Sebag, Analyzing bandit-based adaptive operator selection mechanisms., Ann. Math. Artif. Intell. 60 (1-2) (2010) 25–64.

[40] H. Fischer, Mixing in inland and coastal waters, Academic Pr, 1979.

[41] L. J. Fogel, Autonomous Automata, Industrial Research 4 (1962) 14–19.

[42] L. J. Fogel, Toward Inductive Inference Automata, in: Proceeding of the International Federation for Information Processing Congress, 1962, pp. 395–400.

[43] A. S. Frazer, Simulation of genetic systems by automatic digital computers (part 1), Australian Journal of Biological Science 10 (1957) 484–491.

[44] R. M. Friedberg, A learning machine: Part I, IBM Journal of Research and Development 2 (1) (1958) 2–13.

[45] J. C. Galland, N. Goutal, J. M. Hervouet, TELEMAC: A new numerical model for solving shallow water equations, Advances in Water Resources AWREDI, 14 (3).

[46] M. Gaudesi, A. Marion, T. Musner, G. Squillero, A. P. Tonda, An Evolutionary Approach to Wetlands Design., in: L. Vanneschi, W. S. Bush, M. Giacobini (eds.), EvoBIO, vol. 7833 of Lecture Notes in Computer Science, Springer, 2013, pp. 177–187.

[47] M. Gaudesi, G. Squillero, A. P. Tonda, An efficient distance metric for linear genetic programming., in: C. Blum, E. Alba (eds.), GECCO, ACM, 2013, pp. 925–932.

[48] D. Gizopoulos, Low-cost, on-line self-testing of processor cores based on embedded software routines, Microelectronics Journal vol. 35 (5) (2004) 443–449.

[49] D. Goldberg, Genetic algorithms in search, optimization, and machine learning.

[50] J. Green, J. Garton, Vegetation lined channel design procedures, Transactions of the American Society of Agricultural Engineers 26 (2) (1983) 437–439.

[51] R. Hamming, Error detecting and error correcting codes, Bell System technical journal 29 (2) (1950) 147–160.

[52] J. M. Hervouet, J. L. Hubert, J. M. Janin, F. Lepeintre, E. Peltier, The computation of free surface flows with TELEMAC: an example of evolution towards hydroinformatics, Journal of Hydraulic Research 32 (S1) (1994) 45–64.

[53] D. V. Hinkley, Inference about the change-point from cumulative sum tests, Biometrika 58 (3) (1971) 509–523.

[54] J. H. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, 1975.

[55] J. H. Holland, Adaptation in natural and artificial systems, MIT Press, Cambridge, MA, USA, 1992.

[56] P. Husbands, F. Mill, Simulated Co-Evolution as the Mechanism for Emergent Planning and Scheduling., in: R. K. Belew, L. B. Booker (eds.), ICGA, Morgan Kaufmann, 1991, pp. 264–270.

[57] D. Iliopoulos, C. Adami, P. Szor, Darwin inside the machines: Malware evolution and the consequences for computer security, in: Proceedings of Virus Bulletin Conference, 2008, pp. 187–194.

[58] E. Isaacson, A86/A386 assembler, http://www.eji.com/a86/ (2006).

[59] ISO/DIS26262, Road vehicles - functional safety (2009).

[60] R. Kadlec, The inadequacy of first-order treatment wetland models, Ecological Engineering 15 (1-2) (2000) 105–119.

[61] R. Kadlec, S. Wallace, Treatment wetlands, CRC, 2009.

[62] S. Kauffman, E. Weinberger, The NK model of rugged fitness landscapes and its application to maturation of the immune response, Journal of theoretical biology 141 (2) (1989) 211–245.

[63] J. R. Koza, Genetic Programming, On the Programming of Computers by Means of Natural Selection. A Bradford Book, MIT Press, Cambridge, MA, USA, 1992.

[64] T. Lai, H. Robbins, Asymptotically Efficient Adaptive Allocation Rules, Advances in Applied Mathematics 6 (1985) 4–22.

[65] V. I. Levenshtein, Binary codes capable of correcting deletions, insertions and reversals, in: Soviet physics doklady, vol. 10, 1966, p. 707.

[66] M. A. Ludwig, The giant black book of computer viruses, American Eagle Publications, 1998.

[67] C. J. Martinez, W. R. Wise, Analysis of constructed treatment wetland hydraulics with the transient storage model OTIS, Ecological Engineering 20 (3) (2003) 211–222.

[68] J. Maturana, Á. Fialho, F. Saubion, M. Schoenauer, M. Sebag, Extreme compass and Dynamic Multi-Armed Bandits for Adaptive Operator Selection., in: IEEE Congress on Evolutionary Computation, IEEE, 2009, pp. 365–372.

[69] J. Maturana, F. Saubion, A Compass to Guide Genetic Algorithms, in: G. Rudolph, T. Jansen, S. M. Lucas, C. Poloni, N. Beume (eds.), PPSN, vol. 5199 of Lecture Notes in Computer Science, Springer, 2008, pp. 256–265.

[70] M. Mauldin, Maintaining diversity in genetic search, in: Proceedings of the national conference on artificial intelligence (AAAI conference on artificial intelligence), vol. 247, 1984, p. 250.

[71] E. W. Mayr, Toward a new Philosophy of Biological Thought: Diversity, Evolution and Inheritance, Belknap, Harvard, 1982.

[72] McAfee, Threat Center: TIMID entry, http://vil.nai.com/vil/content/v_1240.htm (1991).

[73] A. Merentitis, et al., Directed Random SBST Generation for On-Line Testing of Pipelined Processors, in: On-Line Testing Symposium, 2008. IOLTS '08. 14th IEEE International, IEEE, 2008, pp. 273–279.

[74] R. Moir, Defining Malware: FAQ, http://technet.microsoft.com/en-us/library/dd632948.aspx (October 2003).

[75] S. Noreen, S. Murtaza, M. Z. Shafiq, M. Farooq, Evolvable malware, in: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, GECCO '09, ACM, New York, NY, USA, 2009, pp. 1569–1576.

[76] P. Parvathala, K. Maneparambil, W. Lindsay, FRITS - a microprocessor functional BIST method, in: Test Conference, 2002. Proceedings. International, 2002, pp. 590–598.

[77] A. Paschalis, D. Gizopoulos, Effective software-based self-test strategies for on-line periodic testing of embedded processors, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on vol. 24 (1) (2005) 88–99.

[78] J. Persson, N. Somes, T. Wong, Hydraulics efficiency of constructed wetlands and ponds, Water Science & Technology 40 (3) (1999) 291–300.

[79] R. Poli, A simple but theoretically-motivated method to control bloat in genetic programming, in: Genetic Programming, Springer, 2003, pp. 204–217.

[80] M. A. Potter, K. A. D. Jong, A Cooperative Coevolutionary Approach to Function Optimization, in: Y. Davidor, H. Schwefel, R. Männer (eds.), 3rd International Conference on Parallel Problem Solving from Nature (PPSN III), Springer-Verlag, Berlin, 1994, pp. 249–257.

[81] M. Psarakis, et al., Microprocessor Software-Based Self-Testing, IEEE Design & Test of Computers vol. 27 (3) (2010) 4–19.

[82] B. B. Rad, M. Masrom, S. Ibrahim, Camouflage in malware: from encryption to metamorphism, IJCSNS International Journal of Computer Science and Network Security 12 (8) (2012) 74–83.

[83] J. Rajski, J. Tyszer, M. Kassab, N. Mukherjee, R. Thompson, K.-H. Tsai, A. Hertwig, N. Tamarapalli, G. Mrugalski, G. Eide, J. Qian, Embedded Deterministic Test for Low-Cost Manufacturing Test., in: ITC, IEEE Computer Society, 2002, pp. 301–310.

[84] R. Rivest, The MD5 Message-Digest Algorithm, RFC 1321.

[85] C. D. Rosin, R. K. Belew, New methods for competitive coevolution, Evolutionary Computation 5 (1) (1997) 1–29.

[86] E. Sánchez, M. S. Reorda, G. Squillero, Test Program Generation from High-level Microprocessor Descriptions, in: M. S. Reorda, Z. Peng, M. Violante (eds.), System-level Test and Validation of Hardware/Software Systems, Springer London, 2005, pp. 83–106.

[87] E. Sanchez, M. Schillaci, G. Squillero, Enhanced Test Program Compaction Using Genetic Programming, in: G. G. Yen, L. Wang, P. Bonissone, S. M. Lucas (eds.), Proceedings of the 2006 IEEE Congress on Evolutionary Computation, IEEE Press, Vancouver, 2006, pp. 3207–3212.

[88] E. Sanchez, M. Schillaci, G. Squillero, Evolutionary Optimization: the μGP toolkit, Springer, 2011.

[89] E. Sanchez, G. Squillero, A. Tonda, Group Evolution: Emerging sinergy through a coordinated effort, in: Evolutionary Computation, IEEE Congress on, 2011, pp. 2662–2668.

[90] E. Sanchez, G. Squillero, A. Tonda, Industrial Applications of Evolutionary Algorithms, vol. 34 of Intelligent Systems Reference Library, Springer, 2012.

[91] B. Sareni, L. Krahenbuhl, Fitness sharing and niching methods revisited, Evolutionary Computation, IEEE Transactions on 2 (3) (1998) 97–106.

[92] H.-P. Schwefel, Cybernetic Evolution as Strategy for Experimental Research in Fluid Mechanics (Diploma Thesis in German), Hermann F?ttinger-Institute for Fluid Mechanics, Technical University of Berlin, 1965.

[93] J. Segura, Malware Diaries: Malware armoring is now the norm, http://blogs.paretologic.com/malwarediaries/index.php/2008/04/14/malware-armoring-is-now-the-norm/ (April 2008).

[94] J. Shen, J. A. Abraham, Synthesis of Native Mode Self-Test Programs, Journal of Electronic Testing: Theory and Applications vol. 13 (2) (1998) 137–148.

[95] SourceForge, Home of μGP3, http://sourceforge.net/projects/ugp3/.

[96] E. H. Spafford, Computer Viruses as Artificial Life, Journal Of Artificial Life 1 (1994) 249–265.

[97] G. Squillero, MicroGP – An evolutionary assembly program generator, Genetic Programming and Evolvable Machines 6 (3) (2005) 247–263.

[98] G. Squillero, Artificial evolution in computer aided design: from the optimization of parameters to the creation of assembly programs, Computing 93 (2-4) (2011) 103–120.

[99] G. Squillero, A. P. Tonda, A novel methodology for diversity preservation in evolutionary algorithms, in: Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation, ACM, 2008, pp. 2223–2226.

[100] C. Suen, N-gram statistics for natural language understanding and text processing, Pattern Analysis and Machine Intelligence, IEEE Transactions on (2) (1979) 164–172.

[101] Symantec, Understanding and Managing Polymorphic Viruses, http://www.symantec.com/avcenter/reference/striker.pdf.

[102] P. F. . P. Szö, Zmist Opportunities, in: VIRUS BULLETIN MARCH 2001 / VIRUS ANALYSIS 1, 2001.

[103] S. M. Thatte, J. A. Abraham, Test Generation for Microprocessor, IEEE Transaction on Computers vol. C-29 (6) (1980) 429–441.

[104] A. Tonda, E. Lutton, G. Squillero, A benchmark for cooperative coevolution, Memetic Computing 4 (4) (2012) 263–277.

[105] A. P. Tonda, E. Lutton, G. Squillero, Lamps: A Test Problem for Cooperative Coevolution., in: D. A. Pelta, N. Krasnogor, D. Dumitrescu, C. Chira, R. I. Lung (eds.), NICSO, vol. 387 of Studies in Computational Intelligence, Springer, 2011, pp. 101–120.

[106] A. M. Turing, Computing Machinery and Intelligence, Mind 9 (1950) 433–360.

[107] J. Von Neumann, A. W. Burks, et al., Theory of self-reproducing automata.

[108] E. Weinberger, NP completeness of Kauffman nk model, a tuneably rugged fitness landscape, Santa Fe Institute Technical Reports.

[109] J. M. Whitacre, T. Q. Pham, R. A. Sarker, Use of statistical outlier detection method in adaptive evolutionary algorithms, in: M. Cattolico (ed.), GECCO, ACM, 2006, pp. 1345–1352.

[110] B. White, H. Nepf, Scalar transport in random cylinder arrays at moderate Reynolds number, Journal of Fluid Mechanics 487 (25) (2003) 43–79.

[111] A. Worman, V. Kronnas, Effect of pond shape and vegetation heterogeneity on flow and treatment performance of constructed wetlands, Journal of Hydrology 301 (1-4) (2005) 123–138.

[112] W. Wu, Computational river dynamics, CRC, 2007.

[113] I. You, K. Yim, Malware obfuscation techniques: A brief survey, in: Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on, IEEE, 2010, pp. 297–300.