Università degli Studi di Trieste
Sede Amministrativa del Dottorato di Ricerca

XX Ciclo Del
Dottorato di Ricerca in
INGEGNERIA DELL'INFORMAZIONE

New Strategies for Efficient and Practical Genetic Programming

(Settore scientifico-disciplinare ING-INF/05)

Dottorando
Cyril FILLON

Coordinatore del collegio dei docenti
Chiar.mo Prof. Alberto BARTOLI
Università degli studi di Trieste

Firma:_____

Relatore
Chiar.mo Prof. Alberto BARTOLI
Università degli studi di Trieste

Firma:_____

Contents

List of Figures

List of Tables

Riassunto

Negli ultimi anni, ingegneri e progettisti hanno espresso un interesse crescente nello sviluppo di nuovi metodi di simulazione e di modellazione per comprendere e predire il comportamento di diversi fenomeni sia in ambito scientifico che ingegneristico. Molti di questi fenomeni vengono descritti attraverso modelli matematici che ne facilitano l'interpretazione. A questo fine, i metodi più comunemente impiegati sono, le tecniche basate sui Reti Neurali, Simulated Annealing, gli Algoritmi Genetici, la ricerca Tabu, ecc. Questi metodi vanno a determinare i valori ottimali o quasi ottimali dei parametri di un modello costruito a priori. E evidente che in tal caso, si dovrebbe conoscere in anticipo un modello idoneo. Quando ciò non è possibile, il problema deve essere considerato da un altro punto di vista: l'obiettivo è trovare un programma o una rappresentazione matematica che possano risolvere il problema. A questo scopo, la fase di modellazione è svolta automaticamente in funzione di un criterio qualitativo che guida il processo di ricerca.

Il tema di ricerca di questa tesi è la programmazione genetica ("Genetic Programming" che chiameremo GP) e le sue applicazioni. La programmazione genetica si può definire come un metodo automatico per la generazione di programmi attraverso una simulazione artificiale dei principi relativi all'evoluzione naturale basata sui contributi originali di Darwin e di Mendel.
La programmazione genetica ha dimostrato di essere un potente mezzo per affrontare quei problemi in cui trovare una soluzione e la sua rappresentazione è difficile. Però la sua applicabilità rimane severamente limitata da diversi fattori. In primo luogo, il metodo GP è inerentemente un processo stocastico. Ciò significa che non garantisce che una soluzione soddisfacente sarà trovata alla fine del ciclo evolutivo. In secondo luogo, le prestazioni su un dato problema dipendono fortemente da una vasta gamma di parametri, compresi il numero di variabili impiegate, la quantità di dati per ogni variabile, la dimensione e la composizione della popolazione iniziale, il numero di generazioni e così via.
Al contrario, un utente della programmazione genetica ha due aspettative: da una parte, massimizzare la probabilità di ottenere una soluzione accettabile, e dall'altra, minimizzare la quantità di risorse di calcolo per ottenerla.

Nella fase iniziale di questo lavoro sono state considerate delle applicazioni particolarmente innovative relative a diversi campi della scienza (informatica e meccanica) che hanno contributo notevolmente all'esperienza acquisita nel campo della programmazione genetica.

In questa tesi si propone un nuovo procedimento con lo scopo di migliorare le prestazioni della programmazione genetica in termini di efficienza ed accuratezza. Abbiamo testato il nostro approccio su un ampio insieme di benchmarks in tre domini applicativi diversi. Si propone inoltre una tecnica basata sul GP per la regressione simbolica di data-set multivariati dove il fenomeno di fondo è caratterizzato da una funzione discontinua.

Questi contributi cercano di fornire una comprensione migliore degli elementi chiave e dei meccanismi interni che hanno consentito il miglioramento dell'algoritmo originale.

Abstract

In the last decades, engineers and decision makers expressed a growing interest in the development of effective modeling and simulation methods to understand or predict the behavior of many phenomena in science and engineering. Many of these phenomena are translated in mathematical models for convenience and to carry out an easy interpretation. Methods commonly employed for this purpose include, for example, Neural Networks, Simulated Annealing, Genetic Algorithms, Tabu search, and so on. These methods all seek for the optimal or near optimal values of a predefined set of parameters of a model built a priori. But in this case, a suitable model should be known beforehand. When the form of this model cannot be found, the problem can be seen from another level where the goal is to find a program or a mathematical representation which can solve the problem. According to this idea the modeling step is performed *automatically* thanks to a quality criterion which drives the building process.

In this thesis, we focus on the *Genetic Programming* (GP) approach as an automatic method for creating computer programs by means of artificial evolution based upon the original contributions of Darwin and Mendel.
While GP has proven to be a powerful means for coping with problems in which finding a solution and its representation is difficult, its practical applicability is still severely limited by several factors. First, the GP approach is inherently a stochastic process. It means there is no guarantee to obtain a satisfactory solution at the end of the evolutionary loop. Second, the performances on a given problem may be strongly dependent on a broad range of parameters, including the number of variables involved, the quantity of data for each variable, the size and composition of the initial population, the number of generations and so on.
On the contrary, when one uses Genetic Programming to solve a problem, he has two expectancies: on the one hand, maximize the probability to obtain an acceptable solution, and on the other hand, minimize the amount of computational resources to get this solution.

Initially we present innovative and challenging applications related to several fields in science (computer science and mechanical science) which participate greatly in the experience gained in the GP field.

Then we propose new strategies for improving the performances of the GP approach in terms of efficiency and accuracy. We probe our approach on a large set of benchmark problems in three different domains. Furthermore we introduce a new approach based on GP dedicated to symbolic regression of multivariate data-sets where the underlying phenomenon is best characterized by a discontinuous function.

These contributions aim to provide a better understanding of the key features and the underlying relationships which make enhancements successful in improving the original algorithm.

Contributions and Thesis Overview

This thesis brings four significant contributions to the Genetic Programming research, enclosed in four different chapters.

- In **Chapter 3** we introduce a new genetic programming system, *Evolutionary Design* (ED), designed and implemented during this thesis to carry out possible complex genetic programming experiments, including the ones discussed later in this thesis. ED provides a modular experimental framework and a rich set of built-in genetic programming features. This software is currently used in the academic world, with the University of Cambridge and Ecole Centrale de Lyon but also in the industrial world with SKF Engineering Research Centre.
- **Chapter 4** presents three nontrivial and difficult real-world problems solved with a GP approach, in a multi-disciplinary context. All of these applications have been published in [24][26][62].
  (i) In the first case study, we evolve a Round Trip Time estimator for improving the TCP protocol. The solutions found performed surprisingly well against a hand-coded estimator used for years up to now.
  (ii) The second case study addresses the problem of web defacement and involves the ED kernel for classifying a set of web pages monitored by a tool developed at the University of Trieste [7]. In this context our GP approach delivered competitive results and is able to detect web defacements with a good sensibility.
  (iii) In the last problem we try to approximate the stiffness matrix for a deep groove ball bearing put in particular conditions.
- **Chapter 5** and **Chapter 6** are the key-stones of the thesis. They present a set of new strategies which encapsulate the GP process for improving efficiency and accuracy. **Chapter 5** is decomposed in two main sections.
  (i) The first part reviews existing issues in scaling genetic programming and the current solutions available in the literature.
  (ii) In the second part we present a new approach called *Reduction & Differentiation* (R&D) Strategy which aims to partition the search space in smaller regions that are explored independently of each other.

– In **Chapter 6** we propose another strategy for discovering discontinuous functions from multivariate data-sets. We identify the portions of the input space that require different approximating functions by means of a new algorithm that we call *Hyper-Volume Error Separation* (HVES).

  Both strategies have been described in [23][25].

Our contributions are prefaced by two chapters which provide a basis for a better understanding of the concepts presented in our contributions.

Chapter 1 introduces optimisation problems, meta-heuristics, evolutionary algorithms and in general terms the genetic programming technique. Following this introductory chapter, **Chapter 2** describes more in details the genetic programming algorithm. The representation and each step of the genetic programming process are discussed. This chapter concludes with a presentation of common benchmark problems.

Finally, the last chapter of this thesis, discusses conclusions, lists the contributions and gives some leads for future work. This concluding chapter is followed by an extensive bibliography.

Acknowledgments

First and foremost I wish to thank my thesis advisor Prof. Alberto Bartoli. Supervising a Ph.D. thesis is a very delicate job: one has to be a scientific guide but also has to be open-mind in front of new ideas. Prof. Alberto Bartoli taught to me how to lead a research project and how to write a research paper, but he always let me explore new ideas. The best I can wish for myself is just to have the opportunity to collaborate with him in the future.

I would like to thank Prof. Carlo Poloni, not only for convincing me to stay in Italy and to try to obtain a Ph.D. at the University of Trieste, but also for his insistent enthusiasm and his support in many occasions.

I sincerely acknowledge my colleagues with whom I shared so much time supporting me with patience and kindness: Paolo Vercesi, Eric Medvet, Valentino Pediroda, Lucia Parussini, Mattia Ciprian, Marzio Lettich.

Several of my fellow postgraduate students contributed in some way to my research. Carlo Gasprotich helped me to implement several plug-ins of the software tool developed during this period, Rodolfo Pelizzola achieves useful experiments for RTT prediction. Finally Vincenzo Gulisano stimulates me with his communicative enthusiasm.

I would like to thank my friends who helped me through the last few years in more ways than they know. Thank you Paolo Geremia, Filippo Soffia, Pierre & Valérie Heckendorn, Vincent Héritier.

The Ph.D. and this thesis were funded by the Marie-Curie RTD network AI4IA, EU contract MEST-CT-2004-514510 (December 14th 2004). This funding is greatly appreciated.

Chapter 1

Introduction to Evolutionary Computation

"Advances are made by answering questions. Discoveries are made by questioning answers."

Bernard Haisch - Astrophysicist

Finding algorithms able to solve more and more complex problems remains one of the most challenging issue in science and engineering. Very often, the problem to solve may be translated in an *optimisation problem* defined with some independent variables and an objective (also called cost) function, to minimise or maximise, according to the problem domain. This very first problem definition may be surrounded by one or several constraints. These constraints are useful to discard the candidate solutions which have parameter's values outside the boundaries fixed by the constraints.

1.1 Meta-Heuristics for Difficult Problems

The complexity growth in the optimisation problems gave the birth to a novel class of algorithms called *"Meta-Heuristics"*. The goal of these algorithms is to find solutions equal to or *close* to the ideal solution. These methods have many principles in common:

- They are *stochastic*: by introducing randomness in their internal mechanisms, they can face a huge search space.
- They are inspired by *analogies*: with physic (Simulated Annealing – SA [44]), ethology (Ant Colonies Algorithms – ACO [15], Particle Swarm Optimisation – PSO [42]) and biology (Evolutionary Algorithms – EA, Neural Networks – NN, Estimation of Distribution Algorithms – EDA [64]).

Unfortunately, they share also the same disadvantages: these methods have many parameters, difficult to set in an optimal way, and may require a large computational effort. However these methods remain competitive when compared with classical iterative methods (for instance gradient based methods) because they are able to escape from a local minima.

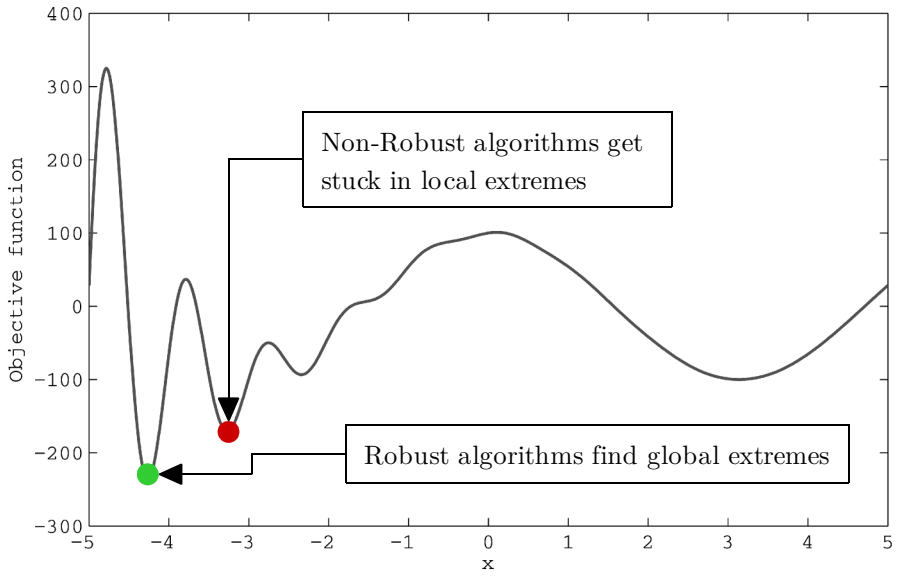


Figure 1.1 - Robustness.

This property of the meta-heuristics, called *robustness*, defines the ability of an optimisation algorithm to reach the absolute extreme of an objective function without to be trapped on a local extreme. Meta-heuristics may escape from local extremes because they accept a temporary loss of performance in order to find further better solutions. A mechanism (specific for each meta-heuristic) controls this performance loss in order to maintain the convergence of the algorithm.

In the following sections, we define more formally what is an optimisation problem. Then we introduce some concepts specific to multi-objective optimisation. These concepts constitute only a basis for a better understanding of this thesis.

1.2 Optimisation Problem

An optimisation problem may be defined as the problem of finding the absolute minimum[1] within the search space Ω containing all candidate solutions related to the given problem. Each candidate solution $\vec{x} \in \Omega$ is a vector which contains n decision variables. The decision variables are the free parameters, i.e. the quantities that the designer can vary or the choices the designer can make. These variables are denoted as x_i , with $i = 1, 2, \ldots, n$. A vector x of n decision variables is represented by:

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{1.1}$$

In many optimisation problems, some *constraints*, related to the domain of the problem in exam, are applied on the candidate solutions. These constraints are the quantities imposed to the candidate solutions, i.e. restrictions and limits that the designer must meet due to norms, functionalities, etc. Each constraint must be satisfied in order to consider that a certain solution is acceptable and belongs to a feasible region. Constraints may be expressed like mathematical inequalities:

$$g_i(\vec{x}) \geq 0 \quad i = 1, 2, \ldots, m \tag{1.2}$$

or equalities:

$$h_i(\vec{x}) = 0 \quad i = 1, 2, \ldots, p \tag{1.3}$$

Note that p, the number of *equality constraints* must be less than n, the number of decisions variables, because if $p \geq n$ the problem is said *over constrained*, since there are no degrees of freedom left for optimising. The number of degrees of freedom is given by $p - n$.

In order to know the performance of a candidate solution, it is necessary to use some criteria to evaluate it. These criteria, called *objective functions*, are expressed as computable functions of the decision variables. Although simple optimisation problem are associated with only one objective function, real world problems may be associated with several objectives, often in conflict with each other, where some objectives should be minimised and others maximised.

1 Or maximum, since $min\{f(\vec{x})\} = -max\{-f(\vec{x})\}$

1.3 Multi-Objective Optimisation Problem

In a *Multi-objective Optimisation Problem* (MOP) we want to find a candidate solution $\vec{x} \in \Omega$ which optimises a vector of k objective functions: $\vec{f}(\vec{x})=[f_1, f_2, \ldots, f_k]$. The very same nature of a MOP implies that there may be many points in Ω representing practically acceptable solutions. It is often the case that the objective functions are in conflict with each other [14], e.g., a solution $\vec{x}$ could be better than another solution $\vec{y}$ for some of the k objectives while the reverse could be true for the remaining objectives.
An important definition for reasoning about solutions of a MOP is the Pareto dominance relationship:

Definition 1 (Pareto dominance). A solution $\vec{u} \in \Omega$ is said to dominate $\vec{v} \in \Omega$ if and only if:

$$\forall\, i \in 1,2,\ldots,k,\ \ f_i(\vec{u}) \leq f_i(\vec{v}) \ \wedge\ \exists i \in 1,2,\ldots,k,\ \ f_i(\vec{u}) < f_i(\vec{v}) \tag{1.4}$$

In other words, a solution $\vec{u}$ dominates another solution $\vec{v}$ (denoted $\vec{u} \succcurlyeq \vec{v}$) if $\vec{u}$ is better than $\vec{v}$ on at least one objective and no worse than $\vec{v}$ on all the other objectives. The Pareto optimal set P_S consists of the set of non dominated solutions: a solution $\vec{u}$ belongs to P_S if there is no other solution which dominates $\vec{u}$. A Pareto optimal front P_f contains all objective function values corresponding to the solutions in P_S (i.e., each point in P_S maps to one point in P_f). Of course, usually, we only consider an approximation of the Pareto optimal set since P_S is not known. In the following we will not mention any further that our notions of P_S and P_f are approximations of their unknown optimal counterparts.

1.4 Evolutionary Algorithms: a Brief Taxonomy

Evolutionary Algorithms (EA) are heuristic search techniques based on an artificial simulation of the mechanisms underlying the evolution of living beings: *natural selection* and *genetic*. The concept of natural selection has been described for the first time by Charles Darwin in "Origin of Species by Means of Natural Selection" [18].

> *"... if variations useful to any organic being do occur, assuredly individuals thus characterized will have the best chance of being preserved in the struggle for life; and from the strong principle of inheritance they will tend to produce offspring similarly characterized. This principle of preservation, I have called, for the sake of brevity, Natural Selection."*

The biological evolution has generated living beings, autonomous, extremely complex, able to solve difficult problems like a continuous adaptation to an environment, uncertain and in permanent evolution. The huge variety of environments in which the life achieves its integration shows that the evolutionary process is robust and can solve difficult problems. Indeed, EA have been applied successfully on difficult problems where other classical optimisation algorithms are not able to produce satisfying results.

All evolutionary algorithms are based on three main concepts:

- Selection: individuals whose variations adapting themselves better to the environment are likely to have more offspring.
- Heredity: offspring are similar but never identical to their parents.
- Variability: slight variations in the offspring may affect significantly the chance of survival.

In the EA context, *individuals* submitted to the evolution laws are candidate solutions providing an answer, more or less efficient, to a particular problem. These solutions belong to a search space bounded by the problem in exam.

The EA process consists in an iterative stepwise refinement of the performance of the individuals. The first step is the creation of a new *population* composed of N individuals randomly generated. Then a *fitness function* evaluates and assigns to each individual a performance measure, or *fitness value*. The definition of the fitness function depends on the objective function and is not always a trivial task. Then this population evolves for a number of iteration called *generation* until to satisfy a termination criterion. At each generation, individuals may reproduce, survive or disappear from the population according to the action of two *selection* operators:

- Selection for *reproduction* which ensures that solutions that have above average performance will participate more often to the reproduction step. According to the Darwin's idea, better is an individual and better is its likelihood of survival. The selected solutions (copies) are subsequently processed by the variation operators.
- Selection for *replacement* (or simply replacement) determines which individuals will have to disappear from the population in order to keep the population size constant, or sometimes to control this size according to a particular policy.

In order to find better solutions than those present in the current population, it is necessary to transform the individuals by applying a set of *variation* operators (or genetic operators).
The two main variation operators in EAs are *mutation* and *reproduction* (also called *recombination* or *crossover*). Mutation changes a small part of the individual's structure while crossover exchanges some features of a set (usually two) individuals to create offspring which are a combination of theirs parents. These operators are inspired by the knowledge gained in the genetic field. The mutation operator may be considered like an innovation operator, since it introduces some new genetic material inside the population. On the other hand, crossover is a conservation operator since it only redistribute preexistent genetic material in the population. These variation operators aim to produce better individuals from stochastic modifications in the structure of the individuals.

The syntactical structure of an individual is often called *genotype*, it is the support for the genetic information. The genotype is decoded in a form called *phenotype* which may interact with its environment, i.e. the problem to solve. Selection acts on the phenotype adjusting the probability of scattering the genetic information in the next generations according to the degree of adaptation. On the contrary the variation operators, mutation and recombination act on the genotype.

The evolution is guided towards an adaption to the environment. The evolutionary scheme is made up an interaction between two statistical phenomena, *randomness* (mutation, recombination operators) and *selection* (which directs the choice of the genotypes involved in the next generation). Randomness is source of disorder whereas selection conditions order.

The general evolutionary algorithm scheme is illustrated in Figure 1.2.

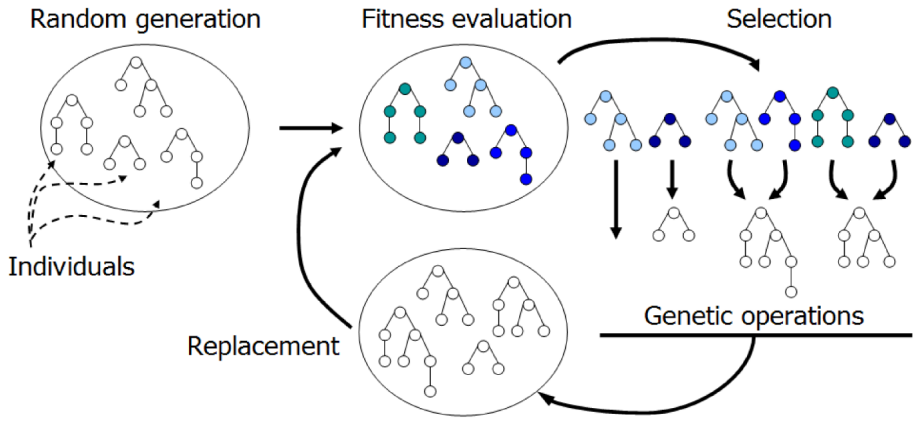


Figure 1.2 - Generic scheme of an evolutionary algorithm.

Usually the EA are divided in four main groups, presented from the oldest to the most recent in the following subsections: *Evolutionary Programming*, E*volution Strategy*, *Genetic Algorithm* and *Genetic Programming* [5].
The main difference between these different approaches lies in the representation of the individuals and consequently the definitions of the genetic operators working on them.

1.4.1 Evolutionary Programming

Evolutionary Programming (EP) has been initially developed by Lawrence Fogel in the sixties [28] in order to evolve finite automata for predicting binary time series. The original scheme has been modified by his son David Fogel in the nineties [27] for parameters optimisation.
The EP approach emphasizes the relationship between parents and offspring rather than a simulation of genetic operators. Unlike to the other evolutionary algorithms, EP does not use a specific representation but provides only a high-level abstraction of the evolutionary process coupled with a genotype representation and a mutation operator directly related to the problem to solve. Indeed there is no recombination mechanism (such as crossover) because recombination does not occur between individuals.
The EP works as follows, a population of μ candidate solutions is randomly generated. Each individual i produces λ offspring according to a mutation operator based on a statistical distribution which weights minor variations in fitness as highly probable and substantial variations as increasingly unlikely. Then, a selection procedure is applied in order to generate a new population composed of μ individuals. Mutation

and selection are repeated in sequence until to find an acceptable solution.

1.4.2 Evolution Strategy

Evolution Strategy (ES) owes its origin to the work of Rechenberg and Schwefel [74] [80] at the Technical University of Berlin. During their researches on the optimal shapes of bodies in a flow, the classical attempts with the coordinates and the well-known gradient-based strategies were unsuccessful. Thus, they proposed the idea of trying random changes in the parameters defining the shape, following the example of natural mutations.

Individuals are represented as real-valued vectors which corresponds to a set of characteristics of the candidate solution. The first version has been called *two membered* ES, or (1+1)-ES by Rechenberg [74]. In this configuration there is only one parent individual and one descendant. Offspring is produced by adding normally distributed random numbers. Then the best of both individuals is selected for the following iteration. This configuration has been extended by the *multi-membered* ES, or (μ+1)-ES, where $\mu > 1$ parents can be recombined by crossover for producing a new individual. The replacement step removes the worst individual, either the offspring itself or one of the parents, from the population before the next generation. The mutation continues to operate in the same way as for the (1+1)-ES.
These models have been generalised by Schwefel [81] with the introduction of the (μ +λ) and the (μ, λ)-ES. In the (μ+λ)-ES, μ parents produce λ offspring, then the selection procedure keeps the best μ individuals amongst parents and children for the next generation. Thus parents survive until they are overcome by better offspring. In the (μ, λ)-ES, only offspring may be selected, i.e. the life time of every individual is limited to one generation.

1.4.3 Genetic Algorithms

Genetic Algorithms (GA) have been invented by Holland in the early 1970s [36]. Their success at searching complex non-linear spaces and their general robustness led to their use in a number of practical problems such as scheduling, financial modeling and parameters optimisation.

In GA, the individuals are encoded with strings of characters of fixed length L. Indeed, the genotype is built as a string of characters taken in a set of n possible characters. The search space (i.e. the set of all possible individuals) is thus composed by n^L different strings. In the canonical genetic algorithm, the available characters are

only 1 or 0 (this structure is called bit-string), and as a consequence the search space size is 2^L.
The GA process starts with the creation of a new population composed of individuals randomly generated. The number p of individuals in the population is usually fixed once for all by the practitioner and will not change during evolution. Then the process performs the following steps until to reach an acceptable solution or a predefined number of generations.

(i) Each individual is evaluated and assigned a fitness value. Actually, the bit-string encoding is mapped to a representation that can be evaluated and assigned a fitness by the heuristic evaluation function.

(ii) The selection procedure collects one individual from the current population and copy this individual in an *intermediate population* (also called *mating pool*). This process is repeated p times, so that, at the end of the selection process, the intermediate population is composed of p individuals.

(iii) The variation operators transform the individuals of the intermediate population to create the next population.

The selection mechanisms used in GAs are very similar to those used in Genetic Programming. Thus, they will be introduced in Chapter 2, when the Genetic Programming approach will be presented more in-depth.

1.4.4 Genetic Programming

We have seen up to now that for a given problem, the methods presented in the previous sections all seek for the optimal or near optimal values of a predefined set of parameters of a model built a priori. But in this case, a suitable model should be known beforehand. When the form of this model cannot be found, the problem can be seen from another level where the goal is to find a program or a mathematical representation which can solve the problem.
According to this idea the modeling step should performed *automatically* thanks to a quality criterion which will drive the building process.
The *Genetic Programming* (GP) [46] approach popularized by Koza in the early 1990s is a robust method for coping with problems in which finding a solution and its representation is difficult but evaluating the performance of a candidate solution is reasonably simple [6][46]. Many engineering problems exhibit this feature and may thus greatly benefit from Genetic Programming techniques.

The Genetic Programming process shares many working principles with the other variants of Evolutionary Algorithms presented above (especially with the GA scheme). However, unlike the traditional methods previously mentioned, the genetic programming process automatically optimises both the functional form and the coefficient values. The most relevant difference between the GP and the GA approaches lies in the representation of the individuals. Indeed, in GP, the fixed length vectors are replaced by programs of variable size.
It is worth to note that genetic programming provides competitive results (some of them are already patented [48]) in a large range of fields: time series prediction [89], classification tasks [68], robot control [40] and so on.

In the following chapter we will enter more in details on the GP approach since this thesis focuses on this particular method and its applications.

Chapter 2

Genetic Programming

"It occurred to me that perhaps you could combine genetic algorithms with the basic thrust of AI, which was to get computers to do things automatically - that perhaps you could evolve a population of programs."

John R. Koza - Computer scientist

Genetic Programming is based on a simple idea: exploit the dynamic of evolution to generate programs automatically. However implementing this idea requires a fairly complex system. The key components of this system are described in details in this chapter.

2.1 Representation of a Candidate Solution

A program subject to modifications by the action of the genetic operators can not be represented by a sequence of instructions which follows a rigid and complex syntax, as in the common programming languages. These languages have been conceived to reject all programs which do not respect the syntactic rules. For instance, if a program written in C was represented like a string of characters, the variation operators which modify randomly some characters will have almost no chance to generate a valid program. Thus the representation chosen must respect the property of *syntactic closure* and support the application of genetic operators without compromise the correctness of the programs.

In the first implementation of GP, Koza used the Lisp programming language to represent the candidate solutions as Lisp S-expressions. S-expressions, or symbolic expressions, are the basic objects in Lisp and are naturally represented as *abstract syntax trees*, where an internal node is an element from a *functions set* which may contain mathematical functions (such as *exp, log, cos, sin,* $+$, $-$), boolean operations (*not, or, and, xor*), conditional operations (*if-then-else*) or relational operations ($<$, $>$,

=). A leaf node of the tree is instead an element from a *terminals set*, which usually contains variables, constants and functions with no arguments. The arity of a node is the number of arguments it expects to receive.
Although the distinction between terminals and functions is useful for tree generation and modification, it does not take in account the semantic of the elements inside these sets and depends only on the arity of the considered elements. Indeed, with this classification a *terminal set* contains variables, constants but also functions without parameters. However variables and functions have different meanings.

In computer science and mathematics, a variable is a symbolic representation used to denote a quantity or expression. In mathematics, a variable often represents an "unknown" quantity which has the potential to change; in computer science, it represents a place where a quantity can be stored. The term has a similar meaning in the physical sciences and in engineering: a variable is a quantity whose value may vary over the course of an experiment, across samples.

The mathematical concept of a function expresses dependence between two quantities, one of which is given (the independent variable, argument of the function, or its "input") and the other produced (the dependent variable, value of the function, or "output"). A function associates a single output with every input element drawn from a fixed set.

We rely on these definitions in order to fill the gap. Therefore we propose to introduce a *relations set* containing functions with zero or more arguments and a *variables set* containing variables and constants.

The classic way for evaluating a syntax tree is a depth-first walk inside the tree where a node is evaluated only after each of its arguments have been evaluated. Thus, the syntax tree evaluation algorithm may be seen as a recursive call on the root node of the tree, which in turn evaluates each of its children, typically from left to right. In the domain of mathematical functions, the functions set may consist of the arithmetic operators, or trigonometric functions. The terminals set may consist of the independent variables and constants fixed according to the preliminary knowledge of the problem, or randomly generated (Ephemeral Random Constants ERC [46]) during the generation of the tree. An example of abstract syntax tree for a mathematical expression is presented in Figure 2.1.

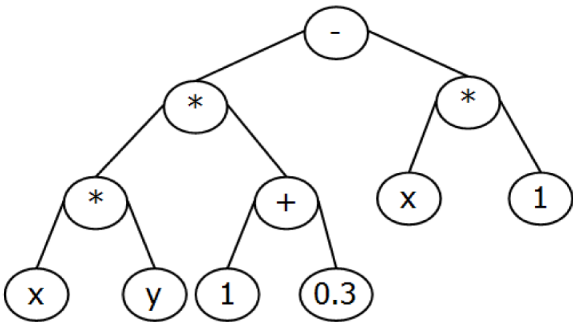


Figure 2.1 - A tree representation for the mathematical expression $x \times y \times 1.3 - x$

Other representations such as linear genetic programming [6] and grammatical evolution [67] are also used and provide alternatives to the abstract syntax tree.

In the GP approach, the tree structure corresponds to the genotype of an individual while the actual performance, i.e., how well it performs when compared with the desired solution, corresponds to the phenotype. In general the mapping from the genotype and phenotype is a non-uniform many to one mapping [93]. That is say, for a given functions and terminals set, there are many ways to express the same function.
Consider a functions set $\{+\}$ and a terminals set $\{a, b\}$ and trees up to a maximum size of 3. All possible trees (six in total) are shown in the left ellipse of Figure 2.2. This set of trees maps onto a set of possible functions listed in the central ellipse. The mapping between the space of trees and the space of functions they represent is independent of the problem and depends only on the functions and terminals set. The mapping between these two sets is many to one and is non-uniform. All functions are represented once except the function $a + b$ which is represented twice (by the trees $a + b$ and $b + a$). No test, based on the functionality of these two trees, will differentiate them as they are functionally equivalent, i.e. for all inputs they will produce the same output. The mapping between the space of functions (central ellipse) and the fitness function (right ellipse) is problem dependent. Typically, the value this function returns is called the fitness value in GP. The actual values returned by the fitness function depend on the test cases.
As a result, the success of a GP search is conditioned by its capacity to maintain a high level of diversity in the population.

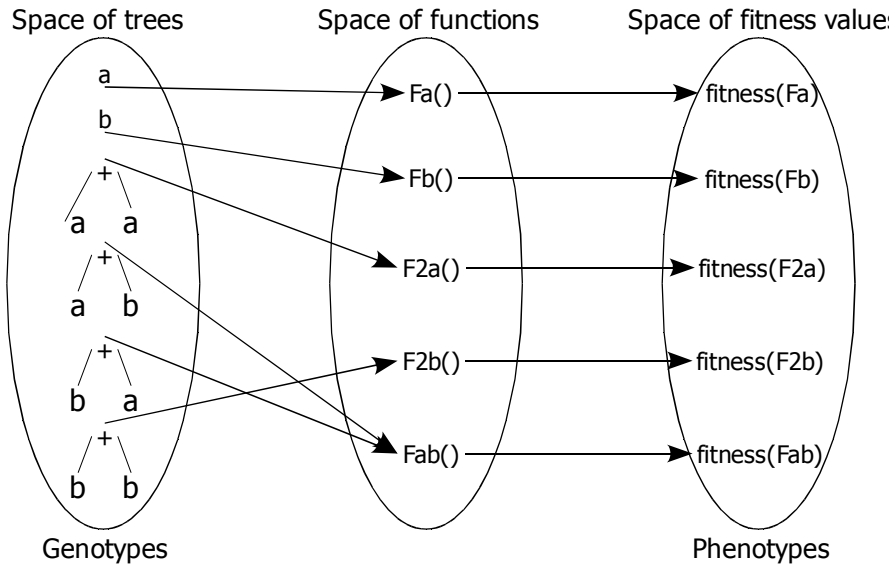


Figure 2.2 - Many to one mapping between trees (left ellipse) and the functions they compute (middle ellipse). One to one mapping between the functions they represent and the fitness values (right ellipse).

In the following, we will consider only the tree-based GP form since this structure is easy to interpret, to manipulate and can capture many properties and features of a modern program.

2.2 Problem Definition

The size and the dimensions number of the search space are strongly dependent on the choice of the components of a program (i.e., variables and functions). As a consequence these elements and the choice of the fitness function largely determine how difficult and ultimately how successful the search will be. In the following section we present these parameters which are inherently dependent of the problem domain.

2.2.1 Choice of Primitives

The choice of the elements which should be part of the *functions set* and *terminals set* is closely related to the problem domain. In other words, the user specifies in advance the building blocks that can be used for constructing an individual – i.e., a solution to the problem. Identification of the elements to include in the *functions set* and *terminals set* should be done carefully by the user in order to provide variables

and functions which are sufficient to express a solution to the problem (*sufficiency* property). Moreover, these elements should be the most significant as possible for the problem domain considered. For instance, if the task is performing a symbolic regression from a numerical data sample, it is necessary to include the independent variables in the *terminals set*, and mathematical functions in the *functions set*. If the problem is related to the robotic field, sensor information and side-effecting operations could be used.

2.2.2 Typing Constraints with Strongly Typed Genetic Programming

An issue arises on how to handle different data types in a tree. Montana in [63] addresses this problem with a *Strongly Typed Genetic Programming* (STGP) system which adds type constraints to the return values and child arguments of nodes. Typically in the STGP approach, each node is assigned a type signature which contains the return type of the node and the type of each argument, if any. For instance, a subtree rooted at node A may serve as a sub-expression at argument position x into node B, only if the return type for A is compatible with the type of the argument at x in B. Similarly, trees have a specific type: a node A may serve as the root of the tree only if its return type is compatible with the tree's type. These type constraints are provided by the user to restrict which kinds of nodes may serve as children of which other nodes, and in which position. All algorithms manipulating the trees must conform to these constraints.

2.2.3 Fitness Function for Assessment

A fitness function provides a performance measure of the individual ability to solve a particular problem. Often the fitness function returns a single scalar value, but more than a single performance criterion can be used as well for multi-objective problem. The fitness function depends on the problem considered and is provided by the user. Fitness is usually evaluated over a set of *fitness cases*. These fitness cases provide a basis for evaluating the fitness of the candidate solutions over a number of different representative situations sufficiently large. The fitness cases are often a small finite sample of the entire domain space. The fitness value is used as a criterion to select individuals that will be used to generate a new population. For this reason a fitness function should be carefully designed in order to deliver a performance measure as a fine-grained discrimination between competing solutions as possible. Binary responses

(perfect solutions, or not) will not work because the selection algorithm will be not sufficiently informed of the closeness to the solution, considering that at the beginning of the evolutionary process it is unlikely to discover a perfect solution.
For example, consider the problem of finding a mathematical expression which approximates the following function $e^{1-x}\sin(2\pi x)+100\cos(x)$. The only explicit knowledge on the target function is a table of x values and corresponding t values. Each input/target output pair is a fitness case. While the fitness metric could answer whether or not the formula submitted gets all fitness cases correct, this level of detail is insufficient for the selection algorithm. Indeed, at the beginning of the evolutionary process it is highly unlikely to find a solution which approximates perfectly all the target data points.

Since the fitness cases and the output estimated by an individual are both numeric, we can define as fitness function the sum of the distances over N fitness cases between the expected values t in the table and values y returned by an individual i.

$$f(y_i,t)=\sum_{j=1}^{N}|y_{ij}-t_j| \tag{2.1}$$

In this work, three fitness functions are commonly used in the experiments involving symbolic regression problems. One is the Mean Absolute Error (MAE), defined as:

$$MAE(y_i,t)=\frac{1}{N}\sum_{j=1}^{N}|y_{ij}-t_j| \tag{2.2}$$

We also used the Root Mean Squared error (RMS error RMSE) to obtain a performance measure stated in the same unit as the target variable.

$$RMS(y_i,t)=\sqrt{\frac{1}{N-1}\sum_{j=1}^{N}(y_{ij}-t_i)^2} \tag{2.3}$$

2.3 Evolutionary Process

Once the elements of the functions and terminals set have been selected according to the problem domain, and when a fitness function for discriminating good candidate programs from bad ones has been chosen, then candidates solutions can evolve as illustrated in Figure 1.2 presented in the Section 1.4.

In the following part we describe each problem-independent step of the GP approach.

2.3.1 Creation of a New Population

The initial population is composed of individuals randomly generated from the elements of the functions and terminals set. A good generation algorithm should provide the most uniform and random distribution of trees as possible since the composition of the initial population has a crucial influence in the evolutionary process. A common initialization algorithm to generate trees in GP is the *ramped half-and-half* method [46]. This method generates trees of different depths, (inside an interval specified by the user) and shapes. The *ramped half-and-half* produces trees generated for one half by the *grow* method and for the second half by the *full* method.

In the *grow* algorithm, a tree of arbitrary depth d is generated from elements randomly chosen in the functions set and terminals set, denoted respectively F_S and Ts, with the following algorithm:

(i) A function is picked from F_S according to a uniform probability distribution to be the root of the tree.

(ii) Let n be the arity of the selected function. Then n nodes are selected from the set $F_S \cup T_S$ and inserted as children of the root node.

(iii) The algorithm is recursively applied on each function amongst these n children nodes, i.e. each child node of arity n which belongs to F_S receives n nodes randomly chosen from the set $F_S \cup T_S$, unless this child node has a depth equal to $d-1$. In the latter case, this node receives n nodes taken from T_S according to a uniform probability distribution.

In other words, the root is selected from F_S according to a uniform probability distribution, thus a tree composed by a single node can not be created initially. Nodes with depth between 1 and $d-1$ are selected with uniform probability from $F_S \cup T_S$, but once a branch contains a terminal node, that branch has ended, even if the specified depth has not been reached. Finally, nodes at depth d are chosen with uniform probability from T_S.

With the *full* algorithm (so named because it generates full trees) a tree is generated by choosing only functions for the nodes until a specified depth is reached beyond which only terminals are selected. As a consequence all leaves of the tree are placed at the specified depth.

Where the *full* method generates trees of a specific depth and shape, the *grow* method allows for the creation of trees of varying size and shape.

Alternative methods have been used to create different distributions of initial trees, for instance in [56], variants of the *grow* algorithm are presented which allow to provide user-defined probabilities of appearance of functions within the tree.
The reader is referred to [59] for a description and an empirical comparison of these methods.

In the research presented in this thesis, the *ramped half-n-half* method will be used for initialisation in order to facilitate further comparisons between our work and the related work found in the literature.

2.3.2 Fitness Evaluation

After the generation of a new population, each individual is evaluated according to a fitness function. The fitness function provides a fitness value which is an index measuring the individual ability to solve the problem. This fitness value, called *raw fitness* as defined by Koza [46] is "the measurement of fitness that is stated in the natural terminology of the problem itself". In other words, the raw fitness value is the direct result of the problem-specific fitness function. This raw fitness may be transformed in a *standardized fitness* in such a way that numerical values close to zero are always considered as better. Still, it may occur that the most natural measure of fitness increases as the program performs better, but in many cases, it is more convenient to make the best value of standardized fitness equal zero. If for a particular problem the upper bound of the raw fitness value f_R^{max} is known, then the standardized fitness f_S of an individual i can be defined as:

$$f_S(i) = f_R^{max} - f_R(i) \quad \text{where } f_R(i) \text{ is the raw fitness of } i. \tag{2.4}$$

The standardized fitness value can be adjusted to obtain the *adjusted fitness* value which lies between 0 and 1 with 1 as the best value. The adjusted fitness value f_A of the individual i in the population is given by:

$$f_A(i) = \frac{1}{1 + f_S(i)} \tag{2.5}$$

Finally the adjusted fitness value can be normalized by:

$$f_N(i) = \frac{f_A(i)}{\sum_{i=1}^{M} f_A(i)} \quad \text{with } M \text{ the number of individuals in the population.} \tag{2.6}$$

The *normalized fitness* values range between 0 and 1 and their sum is 1. The normalized fitness is used by the fitness proportionate selection presented in the next section.
Fitness evaluation is computationally expensive, although it is highly dependent of the number of variables and the quantity of data.

2.3.3 Selection Step

The main idea of selection is to apply the survival-of-the-fittest mechanism by analogy with natural selection on the candidate solutions. Many selection procedures have been proposed to serve this idea. In the following subsections, the most common selection algorithms are introduced.

2.3.3.1 Fitness-Proportional Selection or Roulette Wheel Selection

This selection method due to [36] uses the normalized fitness values as defined in the previous section.
The expected number of selection λ_i of an individual i is proportional to its normalized fitness value $f_N(i)$. The Table 2.1 gives the expected number of descendants for a population of six individuals.

Individuals	$f_N(i)$	λ_i
$X1$	0.24	1.44
$X2$	0.38	2.28
$X3$	0.04	0.24
$X4$	0.1	0.6
$X5$	0.12	0.72
$X6$	0.12	0.72

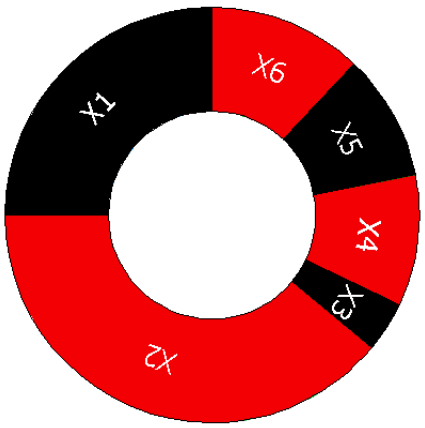


Table 2.1 - Expected number of descendants in a population of 6 individuals.

However the actual number of offspring must be only an integer number. Therefore it is necessary to use a stochastic sampling method for selecting the individuals. This technique is called the *Roulette Wheel Selection* (RWS), as a metaphor with the roulette wheel in a casino. A virtual wheel is created and contains as many sections as the number of individuals in the population. Each section is associated with an individual and its size is proportional to the fitness value performed by this individual. The selection is performed by randomly choosing a section according to a uniform distribution probability. The major drawback of this selection scheme is its high variance. Differences between "bad" and "good" individuals is high in a population, it may happen that only the individuals with the highest fitness values are selected, decreasing the population diversity.

2.3.3.2 Ranking Selection

To address this issue, another selection method has been proposed in [33][91]. This alternative approach ranks the individuals according to their fitness values, with the first individual being the worst and the last individual being the best. Each individual i is then selected according to a probability p_i based on a linear function of its rank r_i:

$$p_i = \frac{1}{M} \times \left(2 - b + (2b - 2) \frac{r_i - 1}{M - 1} \right) \quad \text{with} \quad 1 \leq b \leq 2 \tag{2.7}$$

M is the size of the population and b is the selection bias where higher values of b enforces the selection on the better individuals. Thus, the best individual in the population is selected with a probability $\frac{b}{M}$ and the worst individual with the probability $\frac{2-b}{M}$.

Unlike to the fitness proportional selection, this technique ignores the difference of fitness values between the individuals, so that, individuals with low fitness values which are not the last ranked may participate to the generation of the next population, contributing to its diversity. Moreover, the method does not require an exact knowledge of the fitness function but only to be able to sort the individuals from each other.

2.3.3.3 Tournament Selection

This technique consists in picking randomly a pool of k individuals from the population. An individual may be eventually chosen more than once. Then the tournament selection keeps the individual with the highest fitness value in this pool. If k=1 then the method selects individuals totally at random. On the other hand, if k is assigned a larger value, then the method will select only the individuals with the highest fitness values. This operation is repeated M times, where M is the population size. The tournament selection is popular because it is easy to implement and does not require a centralized fitness comparison between all individuals. Popular values for the tournament size k include 2 and 7. In this thesis we used the tournament selection with k=7 which is a setting widely used in the genetic programming literature.

2.3.4 Variation Operators

Variation or genetic operators act on the structure of the trees and on the content of the nodes or leaves, by combining or modifying two or more parental individuals to create new, possibly better solutions (i.e., offspring). The goal is to combine parental traits with the hope to obtain an individual with an improved fitness. The genetic operators should be able to assume two important functions for the search:

- Exploration of the search space in order to discover the most promising regions, which are more likely to contain ideal solutions.
- Exploitation of the regions previously discovered, for finding the optimal solutions if they are present.

The simplest variation operator in GP is duplication (also called reproduction) since it does only an exact copy of an individual without any change. This operator allows to keep a copy of some selected individuals in the population during artificial evolution.

In GP, recombination of two parents individuals is operated by subtree crossover. This genetic operator uses two individuals provided by the selection procedure. The crossover operation starts with two parental trees and produces two children.

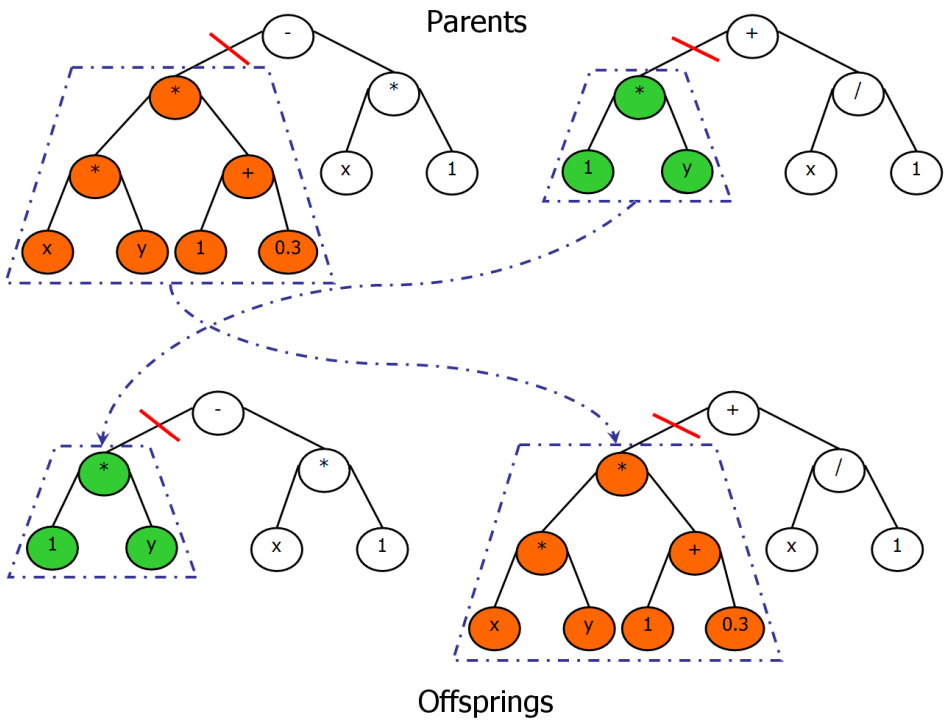


Figure 2.3 - Subtree crossover.

The operation begins by independently selecting one random point (called crossover point) in each parent, using a uniform probability distribution. Then, the operator swaps the subtrees rooted to the points chosen before. This mechanism is illustrated in Figure 2.3.

Choosing randomly a crossover point from all available subtrees implies some bias towards selecting smaller subtrees. Indeed if we consider a tree composed of binary functions, slightly more than 50% of the subtrees are leaves. Therefore choosing randomly nodes in the tree will result that in more than half of the cases in choosing a leave. As a consequence, the uniform selection of crossover points leads the crossover operator to exchange a minimal amount of information (i.e. small trees or leaves). To counter this, Koza [46] suggests the widely used approach of selecting internal nodes (functions) 90% of the time and leaves (terminals) 10% of the time. It is worth to note that when a leave of one parent and the root of another are located at the crossover point, the crossover operation will generate offspring with deeper trees. It is one of the cause of the growth of the code size of the individuals. This phenomenon called *bloat* will be discussed in Chapter 5 of this work. For this reasons other forms of crossover

have been proposed in the litterature [35][61]. For example, *Homologous* and *Size Fair crossover* [49] attempt to preserve tree structures and the size of exchanged subtrees.

While recombination operates on two parental individuals, mutation tranforms locally a single individual. For instance, the *one point-mutation* [71] operator exchange a single node in the tree structure of the individual. The mutation operation begins by selecting a node at random within the tree, and then the mutation operation replaces the function or the terminal node with a node randomly chosen of same arity. This operation is useful to introduce diversity without modify the shape of the tree.

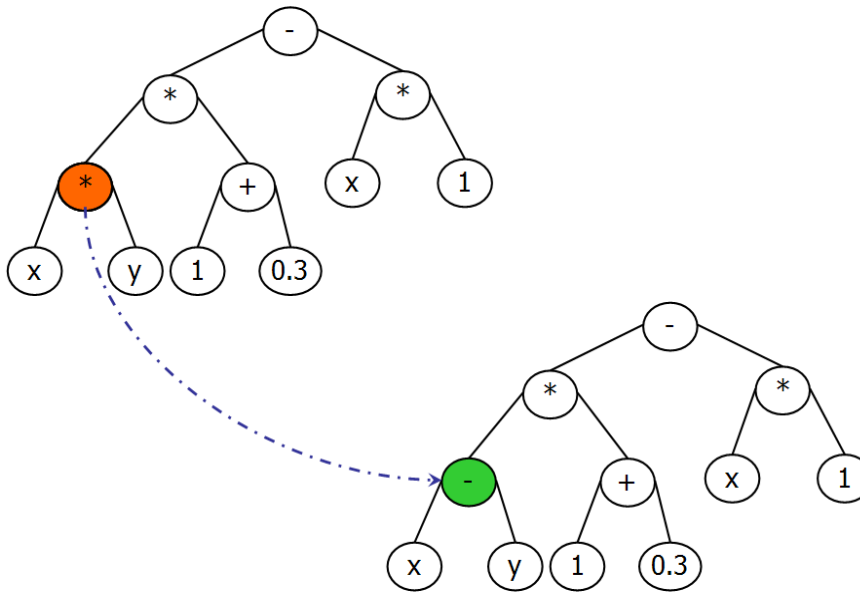


Figure 2.4 - One point mutation.

There are many variants of mutation available in the literature [11]. For example *permutation* (or *swap mutation*) that exchanges two arguments of node or *promote mutation* that takes a node B whose parent is A, and replaces the subtree rooted at A with the subtree rooted at B.

The probability rate to use crossover for variation is currently higher than for mutation since it is believed the crossover operator allows a quicker convergence than mutation but importance of crossover versus mutation is still an open debate.

2.3.5 Replacement Policies

2.3.5.1 Generational Replacement

It is the most simple replacement mechanism since the individuals in the population at the generation g are completely replaced by their offspring at generation g. This cycle is repeated for a predefined number of generation or until to discover an ideal individual. The generational replacement policy requires two user-provided parameters: M which determines the number of individuals in the population and G the maximum number of generations allocated to the evolutionary process. It is the most common replacement policy used with the genetic programming approach.

2.3.5.2 Steady-state Replacement

The *steady-state* replacement policy replaces only a few individuals at each generation. As the whole population is never replaced in once, the children may compete directly with parents. The choice of the individuals to replace depends on various criteria. Parents to replace may be chosen randomly or according to their performance: the worst are replaced. An additional parameter $r < M$ defines the number of individuals replaced at each evolutionary iteration. With a steady-state replacement, a generation is usually defined as the number of iterations necessary to evaluate M individuals. This replacement policy is especially useful when the representation of a solution is shared out several individuals.

2.3.5.3 Elitism

Elitism consists in keeping in the population the best or a few best individuals discovered so far from a generation to another. Thus, the best fitness value is monotonous with the increasing number of generation. Elitism may improve greatly the performance of the evolutionary process for some kind of problems but performs worse by increasing premature convergence on others.

2.3.6 Termination Criterion

The termination criterion is usually defined as the maximum number of generations to be run, eventually coupled with a problem-specific success predicate. When it is possible to obtain an *ideal solution*, then the ideal fitness value can serve as a termination criterion.

2.4 Benchmarks for Genetic Programming

This section introduces a suite of standard benchmark problems divided in three problem domains, which is extensively used in the GP litterature. In addition, we complete this test suite originally composed of five benchmarks by further eleven other benchmarks.

Of course this list is not exhaustive, but it can be considered as a good set of benchmarks to test our approaches since it covers problems of different nature exhibiting different behaviors.

First we introduce the symbolic regression problem in the continuous, and discontinuous case. Then we present several problems in the boolean domain with the *Even-k-Parity*, *k-Bits Multiplexer* and *k-Majority-on* problems. Finally, we describe a last benchmark related to path planning problems.

2.4.1 Numerical Data Modeling

The induction of mathematical models from experimental data is a crucial task for understanding or predicting the behavior of many phenomena in science and engineering. Indeed, mathematical models are more convenient to use and to carry out an easy interpretation. This task is called *symbolic regression* in GP, to emphasize the fact that the object of search is a description of a model in the symbolic language of mathematics, and not only a set of coefficients for a model built a priori. The object of search is then a composition of the input variables, coefficients and primitive functions able to minimise the error of the function with respect to the expected output variable. The shape and the size of the solution to discover are not specified beforehand (however an upper bound is specified for the size). In the same way, the number of coefficients to use and their respective values are determined automatically during the evolutionary process. The genetic programming process is also free to exclude certain input variables from the candidate equation, performing a kind of dimensionality reduction.

More formally, symbolic regression aims to discover a mathematical expression in symbolic form that approximates a target output, dependent variable t, from a vector of inputs, independent variables $\vec{x}=(x_1, x_2, \ldots, x_n)$, according to the following equation.

$t = f(\vec{x}) + \varepsilon$ where ε represents the errors made on the approximation. (2.8)

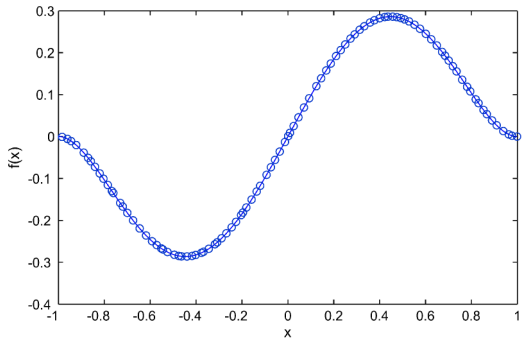
Three classes of symbolic regression problems may be defined according to the nature of the relationships between the independent variables and the dependent variable.

- If the function to discover is completely defined on the intervals described by the independent variables, then the goal is to discover a continuous function.
- If the data points in the sample are not related to a unique function, then the symbolic regression task should associate different expressions with different regions of the input space, i.e., the approximating function should be discontinuous.
- If the dependent variable and the independent variables are related to the same values but mapped according to a different time delay, then the problem is called time series prediction (or forecasting) and the goal is to find a model for predicting future data points from an historical data sample.

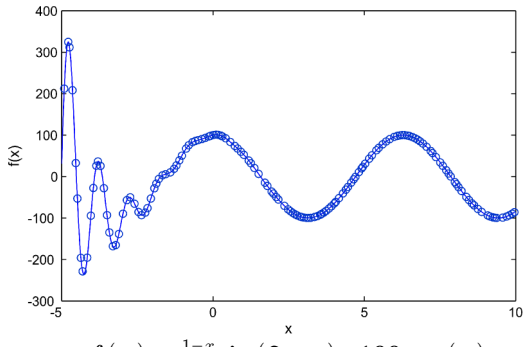
2.4.1.1 Continuous Functions

We used a set of four continuous functions as benchmarks, two are univariate functions, the others have two input variables. All benchmark functions are shown in Table 2.2 and Table 2.3. For each one we report a short description, the data-set characteristics and the functions and terminals sets which are commonly used in the literature.

Univariate functions



Quintic: $f(x)=x^5-2x^3+x$



F_1: $f(x)=e^{1-x}\sin(2\pi x)+100\cos(x)$

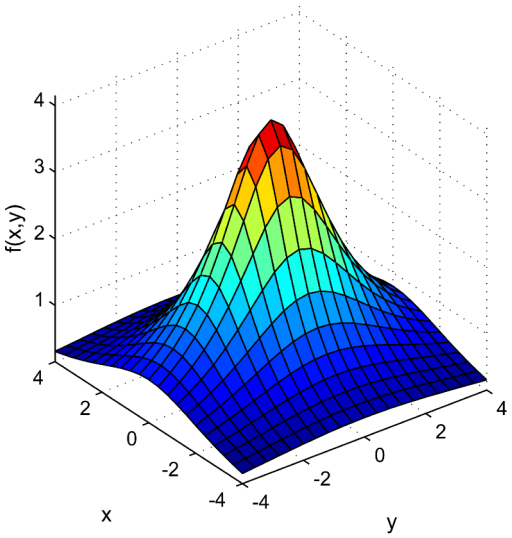
The quintic function is a polynomial function frequently used as benchmark in the GP field [46]. Function F_1 is more difficult to approximate since it simulates a transitory stage in the range [-5, 0] followed by a periodic behavior.

The data-set for the quintic is composed of 100 points drawn in the interval [-1, 1]. For F_1, the data-set is composed of 150 points drawn in the interval [-5, 10]. All data points have been generated according to a uniform probability distribution.
Circles on the plots represent the data points recorded in the data-set.

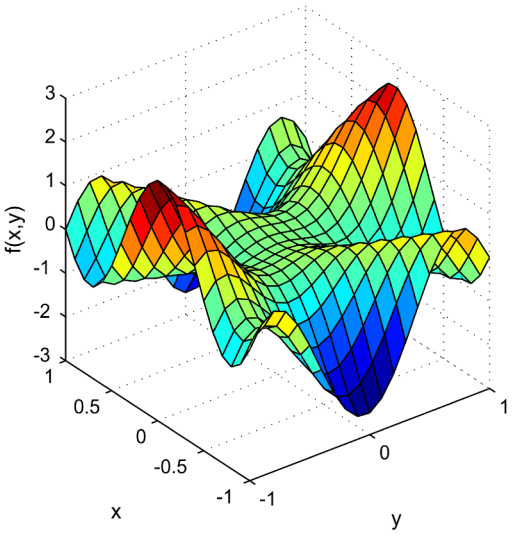
Functions and terminals sets:
- F_S={+, -, ×, /, *sin*, *cos*, *exp*, *log*}
- T_S={x, 1, 0.1, 10}

Table 2.2 - Univariate continuous benchmark functions.

Multivariate functions



MF_1: $f(x,y)=8\left(2+x^2+\frac{y^2}{2}\right)^{-1}$



MF_2: $f(x,y)=3\cos(3(x+y))(\sin(x^2-y^2))$

MF_1 is a multivariate polynomial function. For the data-set we used 441 points corresponding to all combinations of the values in [-4, 4] by step of 0.4.

MF_2 is a multivariate periodic function. For the data-set we used 441 points corresponding to all combinations of the values in [-1, 1] by step of 0.1.

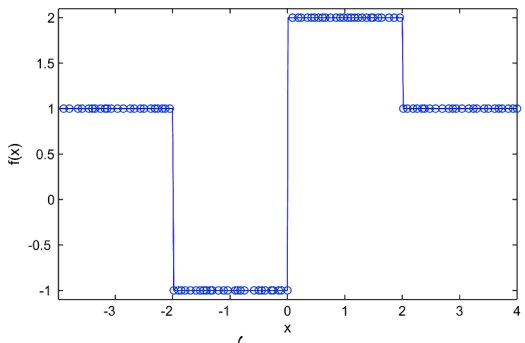
Functions and terminals sets:
- F_S={+, -, ×, /, *sin*, *cos*, *exp*, *log*}
- T_S={x, y, 1, 0.1, 10}

Table 2.3 - Multivariate continuous benchmark functions.

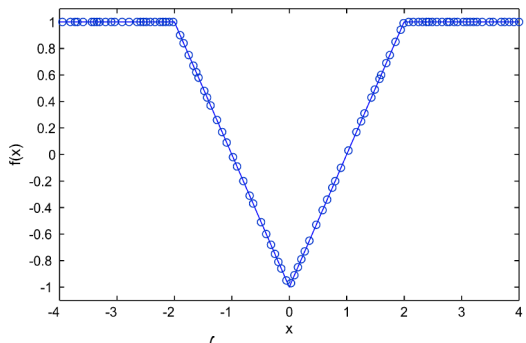
2.4.1.2 Discontinuous Functions

We used a set of eight discontinuous functions as benchmarks, six are univariate functions, the others have two input variables. Functions DF_1 to DF_4 have been used in [83]. All benchmark functions are shown in Table 2.4 and Table 2.5. For each one we report a short description, the data-set characteristics and the functions and terminals sets which are commonly used in the literature.

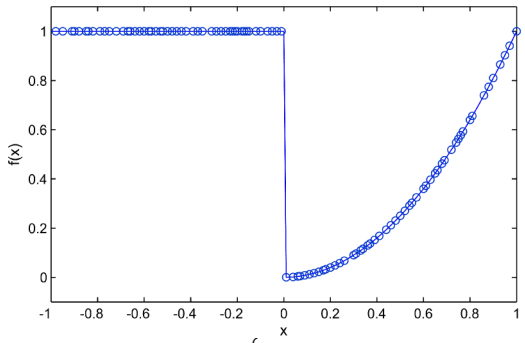
Univariate discontinuous functions



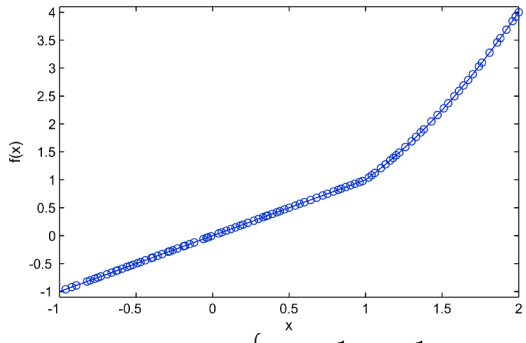
$$DF_1:\ f(x)=\begin{cases}1 & -4\le x\le -2\\ -1 & -2< x\le 0\\ 1 & 0< x\le 2\\ 0 & 2< x\le 4\end{cases}$$



$$DF_2:\ f(x)=\begin{cases}1 & -4\le x\le -2\\ -(x+1) & -2< x\le 0\\ x-1 & 0< x\le 2\\ 1 & 2< x\le 4\end{cases}$$



$$DF_3:\ f(x)=\begin{cases}1 & -1\le x\le 0\\ x^2 & 0< x\le 1\end{cases}$$



$$DF_4:\ f(x)=\begin{cases}x & -1\le x\le 1\\ x^2 & 1< x\le 2\end{cases}$$

DF_1 is a step function with three discontinuities. DF_2 is the inverse of the roof function with three discontinuities.

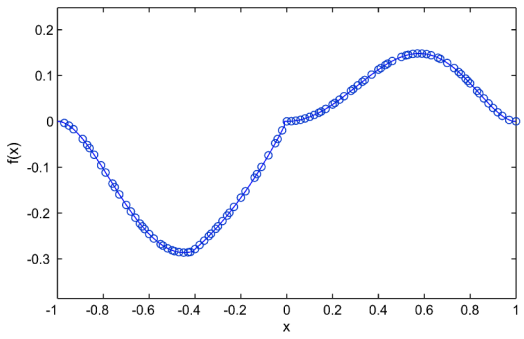
DF_3 and DF_4 contains only one discontinuity.

For each function, the data-set is composed of 100 points drawn in the interval [-4, 4], [-1, 1] and [1, 2] respectively, according to a uniform probability distribution. Circles on the plots represent the data points recorded in the data-set.

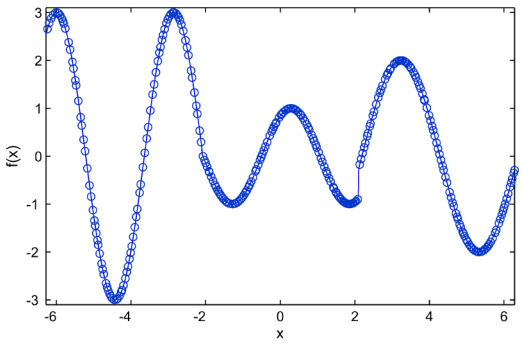
Functions and terminals sets:
- F_S={+, −, ×, /, ≤, ≥, *if-then-else*, *or*, *and*, *not*}
- T_S={x, 1, *true*}

Table 2.4 - Discontinuous functions proposed in [83].

Univariate discontinuous functions



$$DF_5:\ f(x)=\begin{cases} x^5-2x^3+x & -1\le x<0.4 \\ x^4+x^3+x^2+x & -0.4\le x<0 \\ x^6-2x^4+x^2 & 0\le x\le 1 \end{cases}$$



$$DF_6:\ f(x)=\begin{cases} 3\sin(2x+1) & -2\pi\le x<-\frac{2\pi}{3} \\ \sin(2x+1) & -\frac{2\pi}{3}\le x<\frac{2\pi}{3} \\ 2\sin(1.5x+3) & \frac{2\pi}{3}\le x\le 2\pi \end{cases}$$

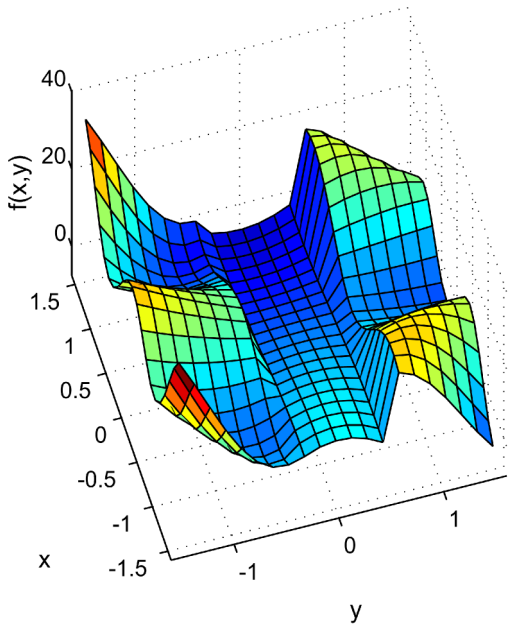
DF5 and DF6 are univariate discontinuous functions defined in such a way that the discontinuities are not very pronounced.

For DF5 we generated a data-set composed of 100 points drawn according to a uniform probability distribution in the interval [-1, 1]. For DF6 we used 300 data points drawn from the interval $[-2\pi, 2\pi]$.

Functions and terminals sets:
- F_S={+, −, ×, /, ≤, ≥, *if-then-else*, *or*, *and*, *not*}
- T_S={x, 1, *true*}

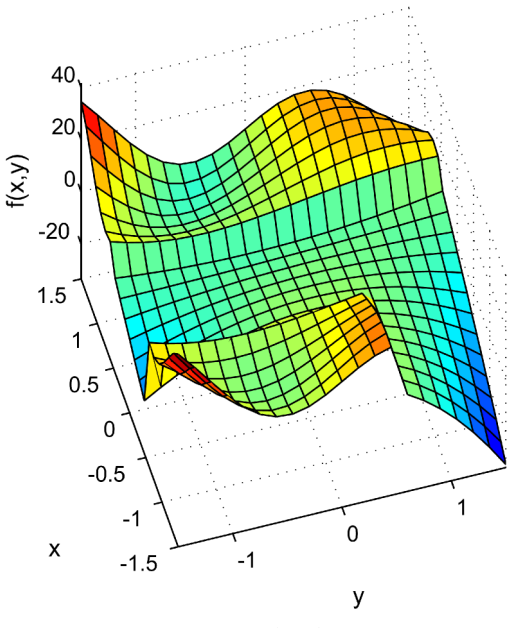
Table 2.5 - Additional discontinuous benchmark functions.

Multivariate discontinuous functions



DMF_1: $f(x,y)=$

$$\begin{cases} f_a(x,y) & -1.5\le x\le 1.5, -0.4\le y\le 0.6 \\ f_b(x,y) & -1.5< x\le 1.5, -1.5\le y< 0.4 \\ f_b(x,y) & -1.5< x\le 1.5,\ \ 0.6< y\le 1.5 \end{cases}$$



DMF_2: $f(x,y)=$

$$\begin{cases} f_b(x,y) & 0.5\le x\le 1.5,\ \ -1.5\le y\le 1.5 \\ f_b(x,y) & -1.5< x\le -0.5, -1.5\le y\le 0.6 \\ f_a(x,y) & -0.5< x\le 0.5,\ \ -1.5\le y\le 0.6 \\ f_a(x,y) & -1.5< x\le 0.5,\ \ 0.6< y\le 1.5 \end{cases}$$

Multivariate functions with two discontinuities. They are a combination of two functions reported below:

$$f_a(x,y)=x^3-3y^2+11xy^2-7x+y+3$$
$$f_b(x,y)=0.1(y-x^2)^2+(1-x)^2+2(2-y)^2+17\sin(1.5x)\sin(1.7xy)$$

For the data-set we used 441 points corresponding to all combinations of the values in [-1.5, 1.5] by step of 0.15.

Functions and terminals sets:
- F_S={+, -, ×, /, *cos*, *sin*, ≤, ≥, *if-then-else*, *or*, *and*, *not*}
- T_S={x, y, 1, *true*}

Table 2.6 - Multivariate discontinuous benchmark functions.

2.4.2 Symbolic Regression in Boolean Domain

2.4.2.1 Even-k-Parity Problem

In this problem, the goal is to discover a boolean function of k boolean inputs that returns *true* if an even number of inputs are *true*, and that returns *false* otherwise. For example for 2 inputs D_0 and D_1, a correct boolean function solving the problem returns *true* when an even number (0 or 2 in this case) of inputs is *true*, and *false* when an odd number (1 in this case) is *true*, as reported in the truth Table 2.7.

D_0	D_1	**Even-2-parity**
false	*false*	*true*
false	*true*	*false*
true	*false*	*false*
true	*true*	*true*

Table 2.7 - Truth table for even-2-parity.

A candidate solution is evaluated over 2^k fitness cases which represent the number of all possible permutations of the values of the k inputs. The fitness function counts the number of differences between the output provided by the candidate solution and the expected output for each combination of inputs. Thus a perfect solution has a fitness 0, while the worst individual has a fitness equal to 2^k.

This problem uses the following functions and terminals sets:

- F_S={*and, or, nand, nor*}. The standard benchmark uses only four boolean functions, which are sufficient to solve the problem.
- T_S={D_0, D_1, . . . , D_k}, the k boolean input variables.

Solving the even-k-parity problem is still recognised as a difficult problem for the genetic programming approach [51] and for this reason they are widely used as benchmarks. Firstly, the function is extremely sensitive to changes in the value of its inputs, since flipping a single bit reverses the output. Secondly, the function set, {*and, or, nand, nor*}, that is usually used by GP researchers attempting to induce it, has an inbuilt bias against parity problems since it omits the building block functions *eq* and *xor*, either of which can be used to construct more parsimonious solutions. The difficulty of the problem rises sharply with the number of inputs k. Koza [46] estimated that the number of evaluations necessary for the canonical form of GP to

solve even-k-parity problems increased by about an order of magnitude for each increment of k.

2.4.2.2 k-Majority-on Problem

In the *majority-on* problem, the aim is to evolve a program that is capable of determining whether the majority of its boolean inputs are set to *true*. Thus, in the 5 inputs version, a solution will deliver *true* if three or more inputs are *true*, and *false* otherwise. The functions and terminals sets for the problem are:

- F_S={*and, or, not*}.
- T_S={D_0, D_1, . . ., D_k}, the k boolean input variables.

2.4.2.3 k-Bits Multiplexer Problem

The task in the k-Bits multiplexer problem is to induce a boolean function which performs multiplexing over a l bits address. In the 6-Bits Multiplexer, there are two boolean-valued address inputs (A_0 and A_1) and four corresponding boolean-valued data inputs (D_0, D_1, D_2, D_3). A correct multiplexer function must return the value of the data input at the address described by the binary values of A_0 and A_1.
For example, if A_1 is true and A_0 is false, the address is 2 (binary 10), and in this case, a solution must return the value stored in D_2. For the 6-Bits multiplexer problem there are six boolean inputs altogether which can be combined in 64 permutations and hence 64 fitness cases.
In the experiments, we also consider the 11-Bits multiplexer which has three address inputs (A_0, A_1, A_2), and thus has eight data inputs (D_0, D_1, . . . , D_7). In this case, there are eleven variables altogether, and so there are 2048 fitness cases. Functions and terminals sets are reported below for the 6-Bits and 11-Bits Multiplexer.

- F_S={*and, or, not, if-then-else*}.
- T_S={A_0, . . . , A_l} $\cup$ {$D0$, $D1$, . . ., D_{2^l-1}}, the l address inputs and the 2^{l-1} data inputs.

The standardized fitness is the number of fitness cases for which the individual returned a wrong value for the data input expected, given the current setting of the address inputs.

2.4.3 Artificial Ant on the Santa Fe Trail

In the Artificial Ant problem, the goal is to find a simple navigation algorithm for an artificial ant which tries to find and eat the most food pellets which are arranged along a path on a two dimensional grid within a predefined number of time steps. The ant may move forward, turn left, and turn right. When moving in a square which contains a pellet, it eats it. The ant has also the capacity to sense the presence of a pellet in the square directly in front of it. We choose to assess the artificial ant on the "Santa Fe trail". This irregular trail is composed of 145 squares, with 21 turns, on a 32 by 32 grid square area. Figure 2.5 shows the "Santa Fe trail" where the 89 food pellets distributed along the trail are represented with solid black squares and the gaps with solid gray squares. This world is toroidal: walking off an edge moves the ant to the opposite edge.

The functions and terminals associated with the ant actions are described in the Table 2.8.

Function syntax	**Description**
If-food-ahead(a, b)	If the ant faces a food pellet then evaluate a, else evaluate b.
progn2(a, b)	Simple sequencing function: evaluate a, then b.
progn3(a, b, c)	Simple sequencing function: evaluate a, b then c.
move	Move the ant forward by one square per step.
right	Turn the ant left by 90 degrees.
left	Turn the ant right by 90 degrees.

Table 2.8 - Functions used in the ant problem.

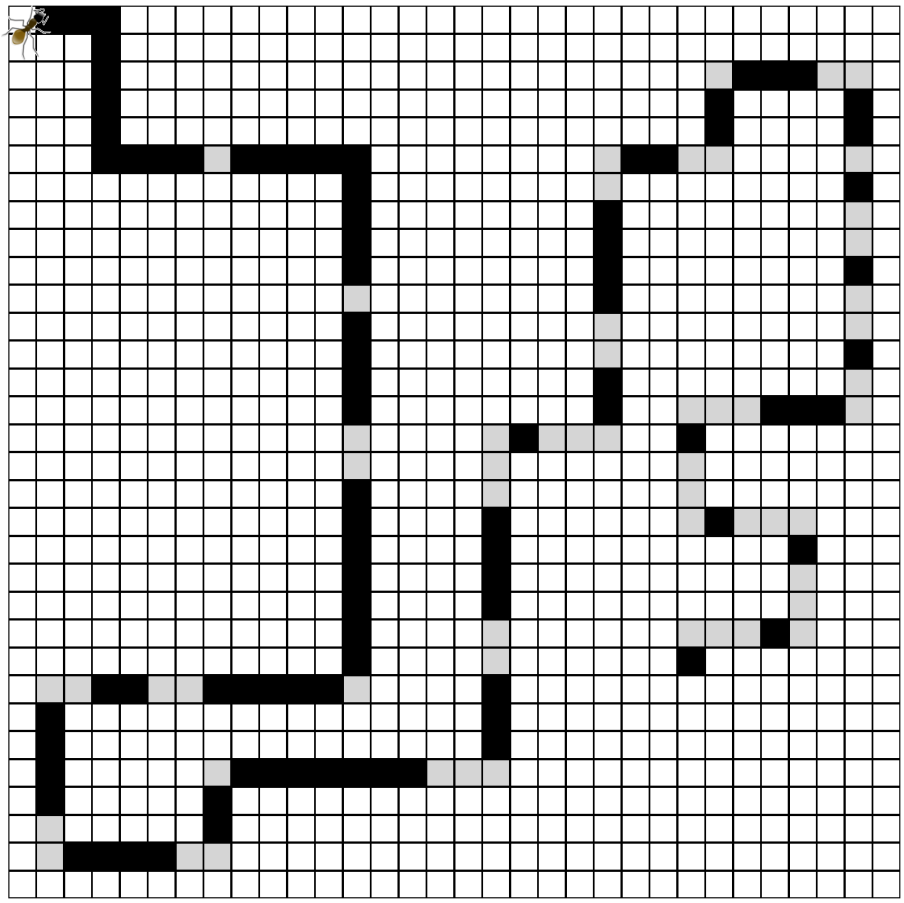


Figure 2.5 - Santa Fe Trail for the artificial ant problem.

An individual is evaluated as follows. The artificial ant starts out on the trail in its initial position and orientation (usually in the North-west corner, facing East). The tree is executed again and again until the ant has eaten all the pellets in the grid or when the 400 time steps have been consumed. This time-out limit prevents a random walk of the ant to cover all the 1024 squares. Each movement counts as one time step. The fitness function is the total number of food pellets lying on the trail minus the amount of food picked up by the ant.

The Artificial Ant domain is quite different from the numerical modeling and the boolean domains since the return value of each tree node is ignored. The only important result is the node's actions in the environment, that is, each node's side effect: moving the ant, turning it, etc. It means that in Artificial Ant, the environment state depends on the order in which the nodes are executed. In symbolic regression problems, it does not really matter in which order the subtrees are evaluated, or if both the *then* and *else* branches of an *if* statement are evaluated but only one is used, because the return values of these branches are their only contributions to the individual. But in Artificial Ant, evaluating a branch means potentially changing the environment state. It means that some optimisation in the individual evaluation stage, like the vectorized evaluation presented in Chapter 3, cannot be used,

The Artificial Ant problem is considered as a highly deceptive problem [50]. There are many possible solutions which are symmetrical. These symmetries lead to essentially the same solutions appearing to be the opposite of each other. E.g. either a pair of *right* or pair of *left* terminals at a particular location may be important. If the search technique does not recognise them as the same thing it may spend a lot of effort trying to decide between them.

2.5 Summary

This chapter has described in details the genetic programming algorithm through three main topics:

- The underlying structures used to represent a program allowing to modify a solution randomly without compromising its correctness.
- The problem definition where a user may bring some knowledge on the problem to solve to the genetic programming process. Moreover the Strongly Typed Genetic Programming approach allows to introduce typing constraints in the problem.
- Each step of the evolutionary process, i.e. creation of a new population, evaluation of the candidate solution, selection algorithms, variation operators and replacement policies are described thoroughly.

This chapter provided few concepts that will be used in subsequent chapters. In addition, we introduce three common problem domains we use in our experiments.

Chapter 3

A Genetic Programming Software Tool

"If you cannot grok the overall structure of a program while taking a shower, you are not ready to code it."

R. Pattis – Computer scientist

To lead a research project in the GP field it is necessary to have software tools able to ease the experimental procedure and to exploit efficiently the computational resources.
Unfortunately, the number of tools dedicated to the GP field are less than a handful. Moreover, only a minority of GP applications have been created with the aim to reduce the number of steps between the experimental design and the exploitation of the results. Very often these tools are only API (Application Programming Interface) and need to be partially reprogrammed even for simple and usual tasks like symbolic regression or time series forecasting for instance. This trend slows down the productivity of evolutionary approaches and reduces researchers and engineers interest for these techniques.

For our research context we look for a tool able to provide the following features:

- platform independence,
- easy extension mechanism,
- simple and quick procedure for the configuration and running steps,
- detailed information available, covering any aspects of a GP run through a complete set of log file records,
- effective implementation of the GP kernel, independent of the problem in exam,
- a set of tree generation algorithms, selection algorithms and genetic operators ready to use,

- multi-objective support,
- time series regression support,
- data extraction from data-set file records.

Developing a software attempting to meet these requirements has proven to be a complex task from a software engineering point of view. In this section we will present a new software tool called "*Evolutionary Design*" (ED) written with the Java programming language.
The architecture and the fundamentals components have been designed and developed by myself. Further, some plug-ins have been added by students I followed during their engineering thesis. The software is used internally at the University of Trieste, but not only, it has been adopted at the University of Cambridge (UK) [21] and the Ecole Centrale de Lyon (France) [45].

First of all, a brief review of the literature on the software available for GP purposes is proposed. The second section introduces the ED architecture and the design choices. In Section 3.2.3 we deal with some implementation issues. Then we describe the configuration and running steps in an experimental context. Finally conclusions are drawn.

3.1 Literature Review

In this brief review we will consider only non commercial products essentially because they do not support modifications of the code for copyright reasons.
Indeed researchers use either public domain software provided by other researchers or developed in house. We focus on three implementations which are popular and widely used in the GP community:

- *lil-GP* (*Little GP*) is written in ANSI C and provides a GP kernel using a variable length tree structure for individual representation. He is currently in version 1.1 and downloadable from http://garage.cps.msu.edu/software/lil-gp/lilgp-index.html.
- *ECJ* is also a generic tool developed in Java at George Mason University. It provides a system for evolving both variable length tree and linear individual representation. He is currently available in version 16 at http://www.cs.umd.edu/projects/plus/ec/ecj/.

– *Open BEAGLE*[2] is a generic framework written in C++ able to handle several kind of Evolutionary Algorithms. This framework is freely available on the project's Web page at http://www.gel.ulaval.ca/~beagle.

The two first softwares have been already reviewed in [92], and a short description of the last one may be found in [31].

In *Lil-gp* all GP parameters are hold in a text file called *input.file*. Thus modifications of parameters such as population size and definition of the selection and genetic operators do not require a recompilation. The problem in exam must be described in five source files. Templates of these files may be found in the application directories which contain the initial five benchmark problems.
As a consequence defines a problem even relatively close to the examples provided is a time consuming task and requires some programming skills. Another problem of the application is that no interface exists for reading data files. Finally, the results of the genetic programming process are reported through six reporting files.

To use *ECJ* on a new application, it is necessary to write additional Java code. Typically the user begins by programming methods for the appropriate classes and overriding default values of parameters. There are four tutorials to guide the user through this process. However, even with the help of this documentation a considerable learning curve exists before a sufficient understanding of the relationships between the numerous packages and classes is attained. As *lil-gp*, *ECJ* is not able to read data files and as a consequence users must write their own code. *ECJ* provides detailed information on the state of the system via three reporting mechanisms:

(i) output, warnings, and errors are typically printed to the screen;

(ii) a text file containing statistical information such as best fitness, average fitness, and size of best individual for both each generation and overall best individual;

(iii) an optional LaTeX file providing a graphical description of the tree representing the best individual.

Open BEAGLE provides a generic framework implementing basic mechanisms and structures for designing finely tuned Evolutionary Algorithms (EA).
It comprises three main components: a vivarium, an evolution system, and an evolver.

2 This recursive acronym means Beagle Engine is an Advanced Genetic Learning Environment

The vivarium is a container for populations of generic individuals. The individuals themselves are specified by an abstract genotype.
In *Open BEAGLE* the parameters are distributed in several XML files. These configuration files are read by a register which centralizes the information.

Open BEAGLE is the most recent mature project, but as the others, it is necessary to put hands on the code for doing GP experiments. As a consequence, we decide to develop a new tool, able to shorten the learning curve and accessible for engineers non specialist in evolutionary computing.

3.2 Evolutionary Design: an Evolvable API

3.2.1 Evolutionary Design Architecture

We have seen in the Chapter 2 that there are many variations of the algorithms for each step of the GP search available in the literature: generation of the individuals, selection procedure, recombination operators and so on. Moreover, it is often necessary to test and experiment the new variants proposed by the researchers. Creating a software which integrates all variations of the algorithm is impossible. For this reason, we design an API based on a highly modular architecture where each step of the GP process may be exchanged with a plug-in. Thus the API is composed of many plug-ins turning on a core as shown in Figure 3.1.

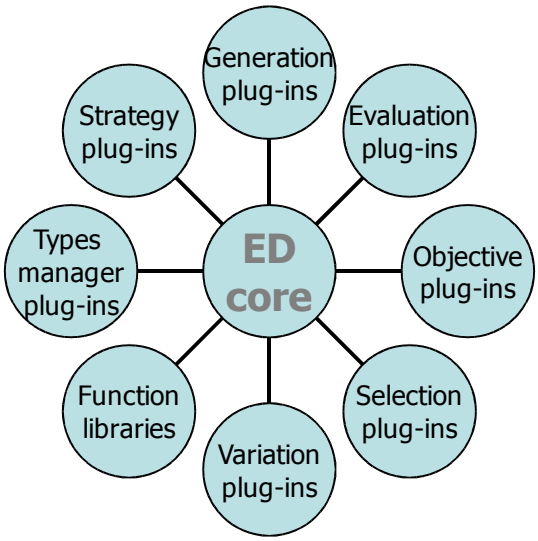


Figure 3.1 - Plug-ins Architecture in Evolutionary Design.

This core provides I/O primitives, plugin loader, code for handling the problem in exam, essential data structures like tree nodes and so on...

Around this core we find a set of eight plug-ins, each category of plug-in corresponds to a specific task. Each plug-in is associated with an interface describing its behavior and an abstract class providing a partial implementation and guidelines for the creation of a concrete plug-in. In such a way, we obtain a very low coupling between the core and a plug-in, and between a plug-in category and another. It means that finding and correcting possible errors in the code is easer. This separation allows many developers to add or extend different modules concurrently without increasing the API complexity or compromising the stability of the software. Another advantage lies in the reduction of the memory footprint since the plug-ins class descriptions are listed when the application starts, and then they are loaded only when needed.

A brief description of the role of each plug-in is given below:

- "Generation plug-ins": generate new individuals from the elements found in the functions and variables set. Three standard algorithms are currently available: Grow, Full and Ramped Half & Half [46]
- "Evaluation plug-ins": evaluate each individual in the population in order to estimate an observed variable from the independent ones. Five plug-ins are available: scalar evaluation, vectorized evaluation described later in Section 3.2.4 and three specialized plug-ins, the first one is dedicated to the Ant problem described in Section 2.4.3 and the others to the RTT-RTO prediction as explained in Section 4.1.1.
- "Objective plug-ins": calculate a fitness index by comparing the estimated values found by a candidate solution with the expected values. More than ten objectives plug-ins are provided to the user.
- "Selection plug-ins": select the best individuals found in the population. Three algorithms are available: Proportionate selection, Ranking based selection and Tournament selection.
- "Variation plug-ins": introduce stochastic variations in the individuals structures and semantics. Standard crossover, one point mutation and promote mutation are available.
- "Function libraries": contain all functions or operators necessary for individuals evaluation. Arithmetic operators, elementary functions, trigonometric functions, hyperbolic functions and actions for the ant problem are available.

- "Types manager plug-ins": work like object factories and allow to create, clone, and merge variables associated with a type chosen by the experimenter.
- "Strategy plug-ins": orchestrators of the evolutionary process, they manage and schedule the execution of the generation, evaluation, selection and variation steps.

It is worth to notice that nearly 60% of the code is embedded in the plug-ins.
The current version of ED offers more than 50 plug-ins which allow a practitioner who has no specific requirements to proceed with no need to write code.

3.2.2 Configuration and Running Steps

Every scientific software need to be finely tunable and configurable, thus in ED practically every feature of the system is determined at runtime from a parameter defined by an user. Parameters define the plug-ins to use, their initial runtime values, the problem definition and so on.

XML (eXtensible Markup Language) is a description language (http://www.w3.org/XML/) especially suitable for modeling data with a standard and versatile structure readable by humans and easily modified. Moreover, any XML file format may be transformed into another XML file format by using XSLT (eXtensible Stylesheet Language Transformations). For instance transforming an XML file into the XHTML format for data visualisation in a Web browser. It is an important feature because it allows an ascendant compatibility when files formats change, and interoperability with other systems using XML files.

ED provides classes for reading an XML file which contains all the information for the configuration of the API.

Before to write a configuration file, a user should answer to several questions in order to give a sufficient description of the problem in exam. This procedure is illustrated in Figure 3.2.

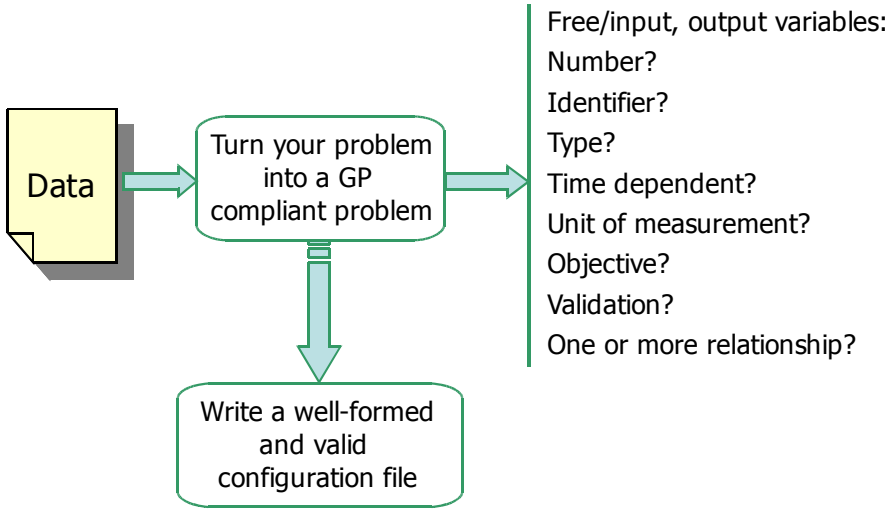


Figure 3.2 - Engineering Process Flow.

The syntax of the configuration file follows the rules defined in a DTD (Document Type Definition) file which should be mentioned in the configuration file. A DTD defines in an unique way the correct structure of the configuration document, by describing the exact sequence of elements and attributes available in the XML file.

The configuration file is divided in two distinct sections:

- The first part (<*parameters*>) is dedicated to the control parameters of the GP process. For example the number of individuals in the population, the identifiers of the plug-ins involved in the experiment, the parameters assigned to each plug-in, the termination criterion and so on...
- In the second part (<*problem_definition*>) are present parameters which describe the problem in exam. For instance the name and type of the variables, the operation list operating on these variables and so on...

Reading and modifying some parameters in a XML file is an easy operation requiring few minutes only. Moreover many templates are already available for various kind of problems.

When ED starts, a bootstrap class called *Ed*, checks the syntax and the semantic of the submitted XML configuration file and loads the control parameters section in an instance of the *ControlParameters* class. Using this first parameter repository, *Ed* asks

to the *PluginsFactory* to provide the strategy plug-in defined by the experimenter. Then, *Ed* tells it to "setup" itself. First the strategy plug-in reads the problem definition in the XML file and then it calls subsidiary plug-ins (such as selection and crossover plug-ins) and tells them to "setup" themselves by reading their respective parameters in the same configuration file. This procedure continues down the chain until the entire system is setup and ready to run.

3.2.3 Implementation Issues

Evolutionary Design is a 100% Java application and should run correctly on any operating systems that has a fully compliant Java 1.5 (http://java.sun.com/) implementation:

- Unix (Solaris, Linux, MacOS/X)
- Windows (2000, XP, Vista)

The choice of Java as programming language is motivated by its independence from the operating system (Java programs runs on a Virtual Machine, "compile once, run everywhere"), and by the large spread of the language in the academic and industrial worlds.
Implementation of the API follows the guidelines of the Object Oriented Design approach. Abstraction and extension of classes for code reusing are intensively used in the implementation of ED. Part of the design has been realized with the (Unified Modeling Language) especially for defining the plug-in architecture. We also used extensively *design patterns* for dealing specifically with problems at the level of software design. Software patterns which are the most common in the API are the *abstract factory*, *singleton* and *strategy* patterns. Of particular interest, the *strategy* pattern allows programmers to define a family of algorithms, encapsulate each one, and make them interchangeable. This is especially suitable for our application which arranges plug-ins together to form a complex overall behavior. For example, the algorithms by which the candidate solutions are assembled, evaluated, selected and modified can be implemented and tested separately, and finally gathered together to form a whole genetic programming process. Please note that in ED the *strategy* pattern is also implemented as a plug-in.

Evaluation of the candidate solutions is the most time consuming task for non trivial applications. A comparative study of several implementations for individual evaluation may be found in [41]. The paper focuses on how to make traversing the tree as fast as possible since a typical evaluation function requires that the parse tree is traversed multiple times. Another interesting approach is subtree sharing [77]. It was experimentally shown that using subtree sharing allows to reduce the amount of memory required to store a population and can be significantly reduce evaluation time.

3.2.4 Vectorized Evaluation

This method of evaluation requires only a single pass through the tree, regardless of the amount of data points. However, this method has several strong constraints. First of all, functions must have no side-effects. Second, the fitness cases must be bounded and fixed during the evaluation of an individual. This rules out problems such as Round Trip Time prediction in which fitness case i is a function of fitness case $i-1$.
In the numerical computation literature it is known as vectorized evaluation. It has been successfully used in [34] to cache previously performed computations. Implementation of this technique is possible for practically any GP system using a tree-based representation for the individuals.

Consider a regression task where all fitness cases are independent from each other and where the functions which process the data points have no side effects.
With the usual approach a tree is evaluated on a data point by recursively go down in the tree for each single case. However using recursion is fairly slow compared to iteration. By vectorizing the evaluation, an entire dataset is evaluated with a single recursive traversal through the tree. The tree is then recursively traversed only once and the evaluation will return the output for all data points. However this method increases the memory footprint because the evaluation is performed at a cost of keeping a number of vectors in memory proportional to the depth of the tree.

The vectorized evaluation is expected to speed up considerably the evaluation step on problems that has a large number of fitness cases and a limited number of conditional branching instructions. Moreover, since the tree is traversed only once per evaluation, it is not necessary to give special attention for improving the tree traversal routine.

3.2.5 Conclusions

Not only ED serves our needs in the research activities related to GP, but it has been used with success in several collaborations either locally with the "Laboratorio Rete dei Calcolatori del DEEI" [62] or in international partnerships with the University of Cambridge [21], Ecole Centrale de Lyon [45] and SKF Engineering Research Centre [26]. These partnerships brought innovative applications for the GP field in a multi-disciplinary context.
Moreover, the first release candidate of the ED project has been validated for further integration in an industrial CAD-CAE (Computer Aided Design-Computer Aided Engineering) tool called Orpheus and maintained by the SKF Engineering Applications department. Several criteria have been examined during this evaluation, for each one we cite the associated comments picked in the resulting technical report [88] provided by SKF:

- **Architecture**: "the overall architecture is setup in such a way that it can cover a wide range of problems in the GP area. The plugin architecture is a flexible means to allow different extensions to be added."
- **Design**: "by observing the overall design a very balanced package structure has been chosen in order to facilitate development on parts of the GP-code (possibly by different teams)..." "The selection of generally available Logging facilities and configuration options (based on XML) greatly simplifies the exchange and integration with other software."
- **Maintainability**: "the code base uses a consistent style and is very well documented. Using the Eclipse development platform and a number of carefully selected plug-ins will make it easy to navigate through the ED-code base (almost 11.000 lines of Java code!) and to preserve a level of program-quality."
- **Stability**: "the ED-program did not crash during the various tests performed and when it fails due to some erroneous parameter configurations a number of Log-files can be consulted."
- **Usability**: "the ED-code can be run very easily on different kinds of platforms with the standard available command line-scripts. It can be installed anywhere on your system and to get things running, minimal changes have to be made (e.g. only your JAVA_HOME variable) in the example scripts. In order to get the best results out of the software, the novice user should have a basic understanding of GP principles. By applying e.g. the workshop held in Trieste, very useful (introductory) presentations were given to get you quickly up-and-running."

Chapter 4

Applications in a Real Context

"The value of an idea lies in the using of it."

Thomas Alva Edison - Inventor

A Genetic Programming approach may be especially useful when the problem in exam exhibits one or more of the following characteristics:

- interrelationships among the relevant variables are unknown or poorly understood,
- finding the size and shape of the ultimate solution to the problem is a major part of the problem,
- good simulators to test the performance of candidate solutions are available, but there are no methods to directly obtain good solutions,
- conventional mathematical analysis cannot provide analytic solutions,
- an approximate solution is acceptable.

In this section, we present three new applications for the GP approach. These applications adress several engineering problems in computer science and mechanical science:

- Automatic synthesis of network delay predictors published in [24].
- Detection of web defacement published in [62].
- Stiffness estimation for deep groove ball bearings published in [26].

4.1 Automatic Synthesis of Network Delay Predictors

Accurate methods for predicting the delay involved in network-related operations are useful in a large variety of scenarios and at several levels of the protocol stack. For example, TCP *(Transmission Control Protocol)* implementations use a formula for setting the *Retransmission Time-Out* (RTO) that is based on another formula for predicting the *Round-Trip Time* (RTT). Both formulas are due to Jacobson and were developed nearly two decades ago [39]. The performance of TCP is heavily dependent on the quality of its round-trip time predictor [39][3], i.e., the formula that predicts dynamically the delay experienced by packets along a network connection.

Techniques for constructing network delay predictors can be roughly classified in two categories. *Model-based*, in which one constructs a model of the system and then derive a formula from that model. *History-based*, in which one uses traces of observed delays and then attempts to find a formula that matches the observations accurately. In practice, once a delay predictor has been constructed and validated for a given workload and environment, that very same predictor will be used everywhere, i.e., with any other workload and in any other environment.

In this work we explore a different approach and describe a methodology for constructing delay predictors in an automatic way. We apply two techniques for multi-objective optimisation in genetic programming to construct a round-trip time predictor for TCP. The construction of an RTT predictor is particularly challenging for genetic programming because, as we shall see more in detail, an RTT predictor must satisfy two conflicting requirements, there are many solutions that are optimal for only one of the two requirements and any such solution performs poorly for the other one.

We evaluate the performances of RTT predictors constructed via multi-objective genetic programming on real traces collected at the mail server of our University. The formulas that we found outperform the RTT predictor used in all existing TCP implementations, including those in Windows 2000/XP, Linux, Solaris and so on.
This result could lead to several interesting developments and applications of genetic programming in the networking field.

The next section presents the RTT prediction problem in detail and introduces the formulas currently in use. Then several multi-objective strategies are defined. Section 4.1.3 presents the experimental procedure used to discover new formulas which

predicts RTT. Section 4.1.5.1 discusses the behavior of the different multi-objectives policies as well as the performances of the formula found by Genetic Programming.

4.1.1 Round Trip Time Prediction Problem

The Transmission Control Protocol (TCP) provides a transport layer, base of many other protocols used in the most common Internet applications, like HTTP (i.e., the Web), FTP (files transfer) and SMTP/POP3 (e-mail). TCP was defined in [76][[75]. We provide in the following only the necessary background for this work. More details on the TCP implementation can be found in many places, for example, in [1].

TCP provides applications with a *reliable* and *connection-oriented* service. This means that two remote applications can establish a connection between them and that bytes inserted at one end will reach the other end reliably, that is, without losses, in order and without duplicates.

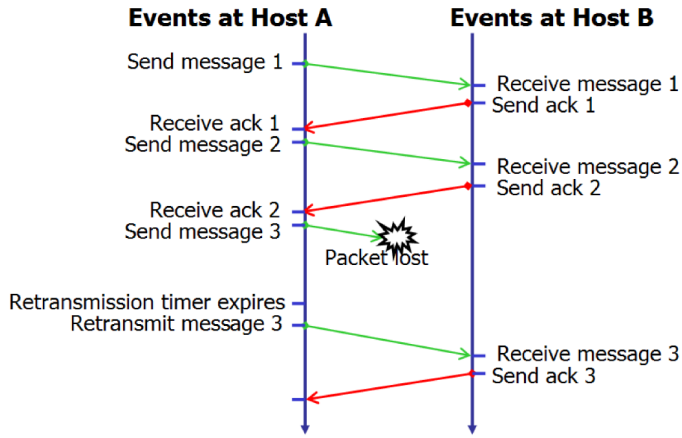


Figure 4.1 - An example of retransmission for the case (i).

To ensure reliable delivery of packets in spite of packet losses, that may occur at lower levels of the Internet protocol stack, the TCP implementation employs internally a retransmission scheme based on *acknowledgments* as follows. Consider a connection between hosts A and B. Whenever either of them, say A, sends a packet B to the other, it sets a *Retransmission Time-Out* (RTO). Whenever B receives a packet S, it responds with another packet *ack(S)* for notifying the other end that S has been indeed received. If A does not receive *ack(S)* before RTO expires, then A resends S. Note that when RTO expires only one of the following is true, but A cannot tell which one:

(i) S has been lost;

(ii) S was received by B but *ack(S)* is lost;

(iii) neither S nor *ack(S)* was lost and RTO expired too early.

The fact that A resends S whenever RTO expires means that the TCP implementation *assumes* that case (i) always holds. The three cases described above are illustrated in Figure 4.1 and Figure 4.2.

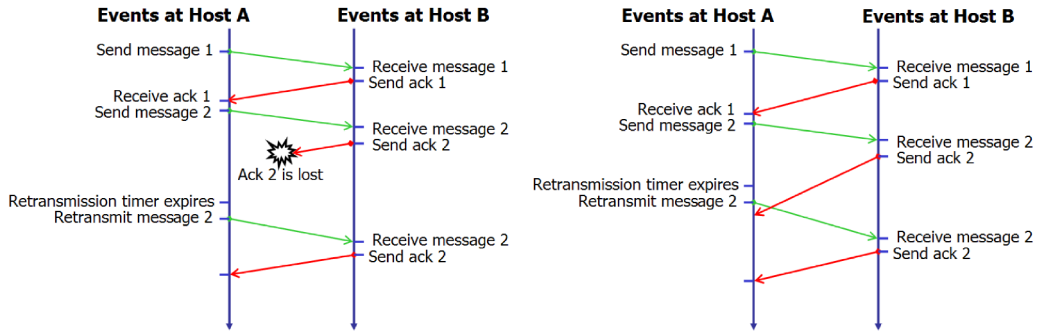


Figure 4.2 - Examples of retransmission for case (ii) and (iii).

Each TCP implementation selects RTO on a per-connection basis based on a formula that depends on the *Round-Trip Time* (RTT) time, i.e., the time elapsed between the sending of a packet S and the receiving of the corresponding acknowledgment *ack(S)*. RTO should be larger than RTT to not incur in case (iii) above too often, which would waste resources at the two endpoints and within the network. On the other hand, RTO should not be much larger than RTT, otherwise it would take an excessively long time to react to case (i) which would result in a high latency at the two connection endpoints.

RTT varies dynamically, due to the varying delays experienced by packets along the route to their destination. Moreover, when sending packet S_i the corresponding RTT value $measuredRTT_i$ is not yet known. The TCP implementation thus maintains dynamically, on a per-connection basis, a *predicted* RTT and selects RTO for S_i based on the current value for $predictedRTT_i$. This component of TCP has a crucial importance on performance of TCP [3].

Virtually *all* the TCP implementations – including those in Linux, Windows 2000/XP, Solaris and so on – maintain $predictedRTT_i$ according to an algorithm due to Jacobson [39].

This algorithm constructs $predictedRTT_i$ based on the previous prediction $predictedRTT_{i-1}$ and the previous value actually observed $measuredRTT_{i-1}$:

$$predictedRTT_i = (1-k_1)\, predictedRTT_{i-1} + k_1\, measuredRTT_{i-1} \tag{4.1}$$

Constant k_1 is set to 1/8 allowing an efficient implementation using fixed-point arithmetic and bit shifting. Initially, *predictedRTT* is set to the first available *measuredRTT*. Another component of the Jacobson algorithm, not shown here for space constraints, constructs RTO_i based on $predictedRTT_i$.

In this work we are concerned with RTT prediction only, i.e., we seek for methods for predicting RTT that are different from Equation 4.1 and hopefully better. The construction of *predictedRTT* has two conflicting objectives to optimise. One would like to minimise the number of underestimates (which may cause premature time-out expiration) while at the same time minimising the average error (which may cause excessive delay when reacting to a packet loss). The problem is challenging because optimising the former objective is very simple – any very large estimation would work fine – but many excellent solutions from that point of view are very poor from the point of view of the average error.

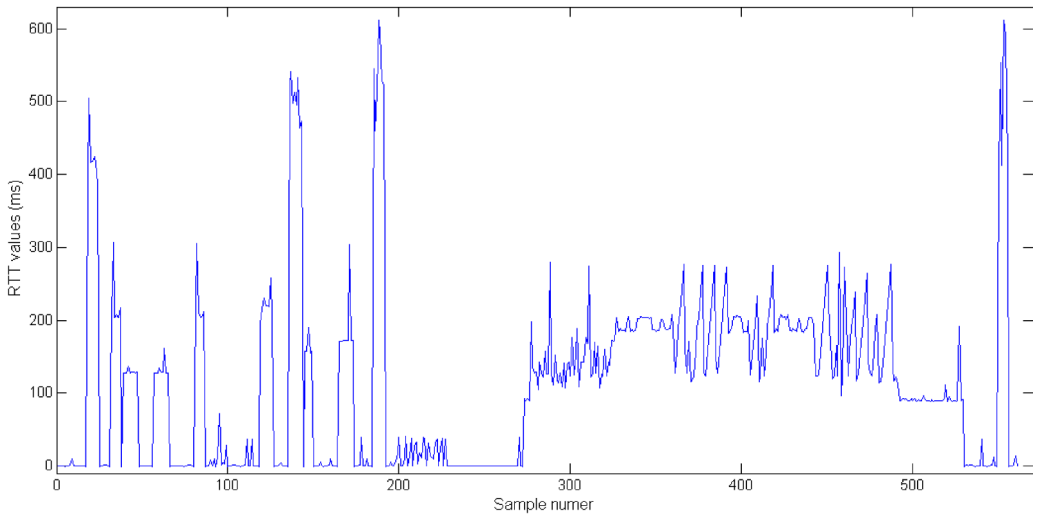


Figure 4.3 - Sample of RTT values for consecutive connections.

An example of the sequence of RTT values measured in TCP connections is given in Figure 4.3 above. It is easy to realize that predicting the next value of RTT based on the past measurements, with a small error and few underestimates, is hard.

A more complete characterization of real RTT traces can be found, for example, in [1].

4.1.2 Multi-objective Approaches for RTT Prediction

With respect to our RTT prediction problem, we define two objective functions:

- *Mean Absolute Error*($\vec{u}$) as the average of the absolute distances between the sequence of *predictedRTT* constructed by solution $\vec{u}$ and the corresponding sequence of the *measuredRTT* actually observed.
- *Number Of Underestimates*($\vec{u}$) as the number of times in which the *predictedRTT* constructed by $\vec{u}$ is lower than the corresponding *measuredRTT*.

Both functions are evaluated on a set of training data collected as described in Section 4.1.3.1. The ideal value for each of the two objective functions is zero. In the following subsections we describe the approaches that we have applied to this MOP.
In all the approaches we kept the non-dominated solutions found during the evolutionary search. That is, at each generation we perform the following steps:

(i) Store in an external archive all the individuals non-dominated by any other individual in the current population;

(ii) drop from the archive individuals dominated by some other member of the archive.

4.1.2.1 Scalarization

This approach consists in combining the k objective functions in a single scalar objective F_{ws} to be minimised. The combination consists of a weighed sum of the objective functions with weights fixed a priori [14]:

$$F_{ws} = \sum_{i=1}^{k} w_i f_i \tag{4.2}$$

Since the relative importance of the objectives cannot be determined univocally – i.e., with one single choice of weights – we explored several combinations of weights between [0.0, 2.0] varied in steps of 0.25. In the extreme cases we give weight 2 to one of the objectives and weight 0 to the other. Note that the sum of weights is equal to the number of objectives.

4.1.2.2 Pareto Dominance

We applied the Pareto dominance technique with a tournament selection scheme close to that in [22]. In the cited paper an individual is randomly picked from the population and then compared with a comparison set. Individuals that dominate the comparison set are selected for the reproduction. We modified this scheme according to the classical tournament selection, in which a group of n ($n \geq 2$) individuals is randomly picked from the population and the one with the best fitness is selected. Our scheme works as follows:

(i) A tournament set of n ($n \geq 2$) individuals is randomly chosen from the population.

(ii) If one individual from the set is not dominated by any other individual, then it is selected.

(iii) Otherwise an individual is chosen randomly from the tournament set.

4.1.3 Experimental Procedure

4.1.3.1 RTT Traces

We collected a number of RTT samples on the mail server of our University. This server handles a traffic in the order of 100.000 messages each day (see http://mail.units.it/mailstats/). We intercepted the SMTP traffic at the mail server for 10 minutes every 2 hours for 12 consecutive days (SMTP is the application-level protocol for sending email messages). The *tcpdump* software intercepted the network packets. The output of this tool was processed by the *tcptrace* software which constructed the *measuredRTT* data for each connection. We then dropped connections with less than 5 RTT values. The tools that we used are freely available on the web, at http://www.tcpdump.org/ and http://www.tcptrace.org/ respectively.

The resulting trace consists of 396109 RTT measures in 41521 TCP connections. These data are grouped in 78 files, for convenience. We chose one of these files as *training set*, consisting of 5737 RTT measures on 611 TCP connections. The file selected as training set exhibits a large variety of scenarios: small and large variations, abrupt changes and so on. We used the remaining files, consisting of 390372 RTT measures on 40910 TCP connections, as *cross validation set*. The generalization capabilities of the solutions found on the training set have been evaluated on the cross validation set.

4.1.4 On Setting the GP Process

The terminal set and the function set is shown in Table 4.1. We used only arithmetic operators and a power of 2 as constant in order to obtain formulas that can be computed efficiently (this is a key requirement in TCP implementations and is necessary for fair comparison with the RTT estimator developed by Jacobson that is currently used). We allow the resulting formula to include the last measured RTT ($measuredRTT_{i-1}$) and the last prediction ($predictedRTT_{i-1}$). We did not include values more far away in the past because the autocorrelation of RTT traffic is known to decrease very quickly [60].

Terminal set	½, 1, $measuredRTT_{i-1}$, $predictedRTT_{i-1}$
Function set	+, -, ×, /

Table 4.1 - Terminals and functions sets.

For the first multi-objective approach we performed 25 independent executions for each combination of weights for a total of 225 runs, for the Pareto tournament scheme we performed the same amount of runs in order to carry out a fair comparison. Each execution starts with a different seed for the random number generator. We allocate 50 generations for each test. All others parameters are summarized in the Table 4.2 below.

Parameter	**Setting**
Population size	500
Selection	Tournament of size 7
Initialization method	Ramped Half-and-Half
Initialization depths	2-6 levels
Maximum depth	6
Internal node bias	90% internals, 10% terminals
Elitism	5
Crossover rate	80%
Mutation rate	20%

Table 4.2 - Parameters settings.

4.1.5 Results

4.1.5.1 Comparison of the Multi-objective Approaches

In this section we compare the Pareto fronts generated by multi-objective genetic programming search based on the scalarization method and the Pareto-based tournament selection. A thorough comparison between the two approaches would require several indicators as those described in [97][9]. We use a much simpler comparison because both approaches exhibit significantly better performance than our baseline solution – the original algorithm by Jacobson.
We evaluated the performance on the cross validation dataset of the Jacobson algorithm and of each solution found with GP and belonging to the Pareto set. The results are summarized in Figure 4.4.

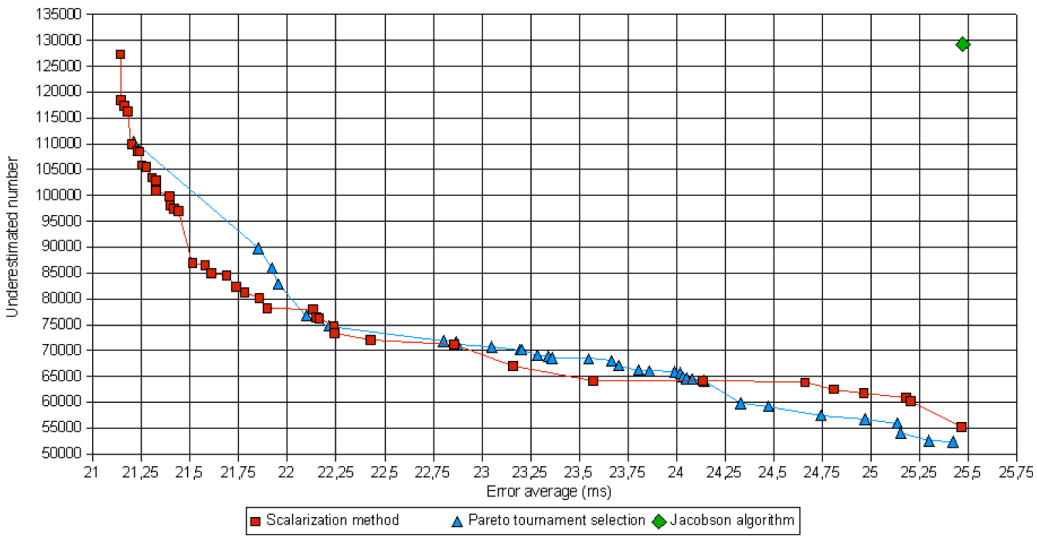


Figure 4.4 - Pareto front generated with the non-dominated solutions for each multi-objective approach

The most important result is that GP found 84 solutions that outperform the Jacobson algorithm, 40 have been found with the scalarization method and 34 with the Pareto based tournament selection. This result is particularly significant because it demonstrates the potential effectiveness of GP in an important application domain. Interestingly, the scalarization method generates solutions that dominate those found with the Pareto-based tournament for a large range of values of the error average (from 21.125 to 24.125) except for one solution. Pareto based tournament provides better solutions in terms of the number of underestimated RTTs.

4.1.5.2 Comparison of the RTT Predictors

To gain further insights into the quality of the solutions, in particular regarding the improvements that can be obtained with respect to the Jacobson algorithm, we analysed the performance on each single file of the cross validation dataset. We present the results for the Jacobson algorithm and for two solutions located at the two extremes of the Pareto front: the one giving the best results in terms of underestimated RTTs (located bottom-right in Figure 4.4) and the one giving the best results in terms of average error (located top-left).

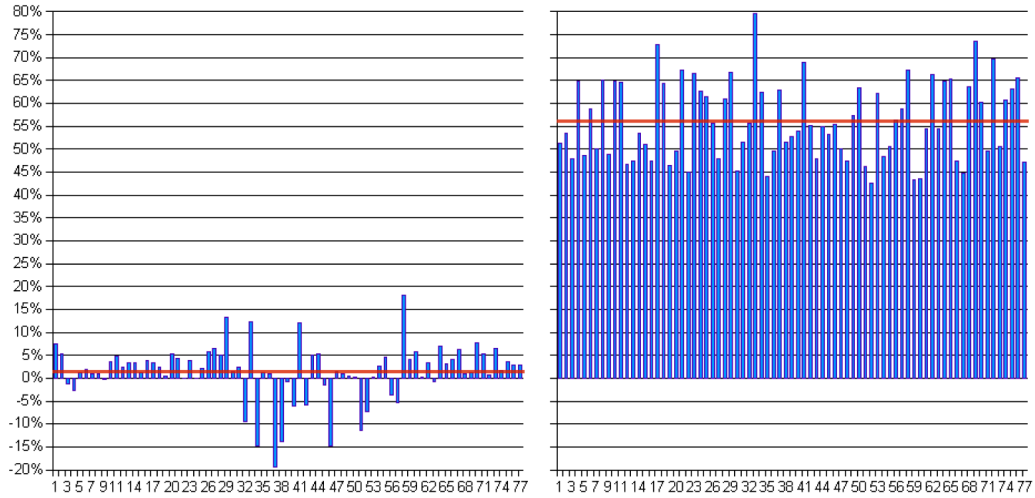


Figure 4.5 - Number of underestimated RTT for each trace file. Best formula found by GP in terms of average error (left) and in terms of number of underestimated RTTs (right)

Figure 4.5 describes the results in terms of underestimated RTTs. Each point in the X-axis corresponds to one file of the cross validation dataset, whereas the Y-axis is the improvement with respect to Jacobson, in percentage. The horizontal line represents the average improvement across the entire cross validation dataset.
Figure 4.5-left shows the result for the best formula found by GP in terms of average error. It can be seen that the average improvement over the Jacobson algorithm is small (approximately 2%) and that in some files the average error is worse.
Figure 4.5-right shows the result for the best formula found by GP in terms of number of underestimates. This case is much more interesting because the formula found by GP largely outperforms the Jacobson algorithm, with a 56% average improvement (34% of RTTs are underestimated by Jacobson and only 15% by the formula found by GP). Moreover, a remarkable improvement can be observed in every trace file.

Figure 4.6 describes the results in terms of average error, with the same notation as above. It can be seen the best formula in terms of average error (left figure) exhibit a 15% average improvement over Jacobson (corresponding to 4.7 ms) and that some improvement can be observed in every trace file. The best formula in terms of underestimated RTTs (right figure) exhibits instead essentially the same performance as that of Jacobson.

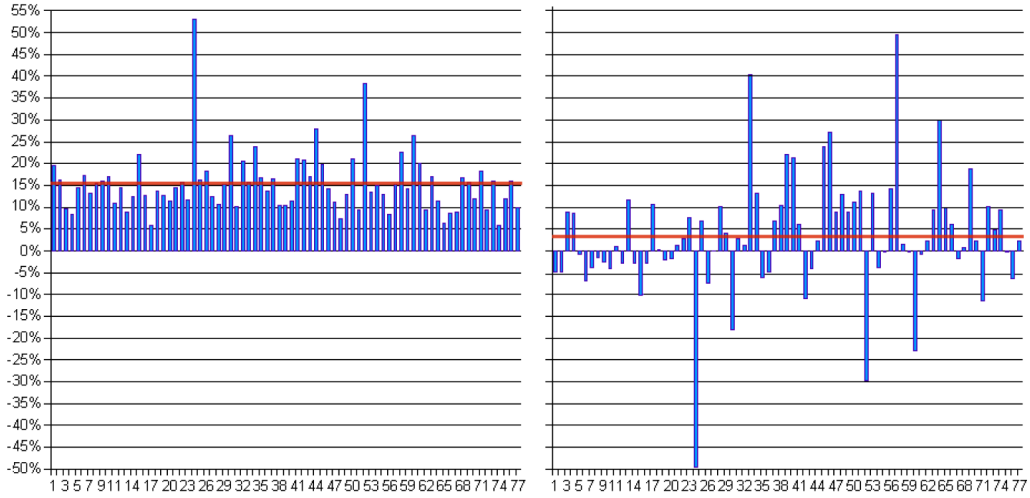


Figure 4.6 - Error average for each trace file. Best formula found by GP in terms of average error (left) and in terms of number of underestimated RTTs (right)

4.1.6 Summary

We applied two radically different multi-objective approaches on an important real-world problem. We used an *a priori* method which combines all the objectives into a single one by weighting each objective in advance, and an *a posteriori* approach based on a Pareto tournament selection. The quality of the solutions provided by the two approaches is similar, although solutions obtained with Pareto-based tournament tended to be more effective in terms of the number of underestimated RTTs (a particularly critical issue for TCP performance). The effectiveness of the simple scalarization method was rather surprising and is probably due to the small number of objectives: covering a sufficiently wide set of weights remains computationally acceptable.

While this is an interesting result itself, the most significant result consists in the performance of the formulas found with multi-objective GP: they are significantly better than those exhibited by the RTT predictor used in all TCP implementations.

This result could lead to several interesting applications of GP in the networking field – e.g., tailoring RTT predictors to individual hosts, rather using the same estimator for all hosts; differentiating the estimator based on the application using TCP, whether web navigation or transmission of email; and so on.

4.2 Detection of Web Defacements

Web site defacement is the process of introducing unauthorized modifications to a web site. Usually one or more web pages are replaced or modified, either completely or only in part. More than 490,000 web sites have been *defaced* last year and the trend has been constantly growing in the recent years: every day, about 1500 web pages are defaced [98]. A defaced site may contain disturbing images or texts, political messages and so on, as well as a few messages or images representing a sort of signature of the hacker that performed the defacement.
Fraudulent changes could also be aimed at remaining hidden to the user, focussing for example on links or forms.
This kind of attack creates obvious security problems for users and makes the need for automated defacement detection techniques evident. However, detecting web defacements automatically is very difficult because web pages are highly dynamic and their degree of dynamism may vary widely across different pages. The challenge is to deal with such highly dynamic content keeping false positives to a minimum, and, at the same time, generating meaningful alerts.

In this section we propose a novel detection approach of web site defacement based on genetic programming. What makes GP particularly attractive in this context is that it does not rely on any domain-specific knowledge, whose description and synthesis is invariably a hard job. In a preliminary learning phase, GP builds an algorithm based on a sequence of readings of the remote page to be monitored and on a sample set of attacks. Then, we monitor the remote page at regular intervals and apply that algorithm, which raises an alert when a suspect modification is found. We developed a prototype based on a broader web detection framework proposed earlier [7][8] and we tested our approach over a dataset of 15 dynamic web pages, observed for about a month, and a collection of real web defacements. We simulated attacks by means of a set of real defacements extracted from a public attacks archive (Zone-H digital attack archive, http://www.zone-h.org). We compared the results to those of a solution developed earlier [7] which embedded a substantial amount of domain specific knowledge, and the results clearly show that GP is remarkably effective for this task. GP generates automatically algorithms capable of detecting almost all unauthorized

modifications and coping with the highly dynamic nature of web pages while obtaining a false positive rate sufficiently low to be useful.

We evaluated the proposed approach on a dataset composed by 15 dynamic web pages that we observed for about a month. We simulated attacks by means of a set of real defacements extracted from a public attacks archive (Zone-H digital attack archive, http://www.zone-h.org). The effectiveness of GP on this task is remarkable: GP generates automatically algorithms capable of detecting almost all
unauthorized modifications and coping with the highly dynamic nature of web pages while obtaining a false positive rate sufficiently low to be useful.

4.2.1 Related Work

Several prior works have addressed the use of GP [95][65], as well as other evolutionary computation techniques [94][12], for network based or host based intrusion detection systems. What makes such approaches attractive is their ability to find automatically models capable of coping effectively with the huge amount of information to be handled [85]. We are not aware of any attempt of using such techniques for automatic detection of web defacements.
One of the key differences between our scenario and such prior studies concerns the nature of the dataset used for training: in our case, it includes relatively few readings (some tens), each one composed by many values, whereas in the network and host-based IDS fields, it usually includes much more readings (thousands and more), each one composed by few values.

Concerning web defacements detection, there are several automatic tools tailored at solving this problem and some of them are commercially available. All the tools that we are aware of must be installed *on the site* to be monitored and are based on essentially the same idea: a copy of the page to be monitored is kept in a "very safe" location; the page content is compared to the *trusted copy* and an alert is generated whenever there is a mismatch [82][30]. Such an approach has the potential to spot any unauthorized change, irrespective of how small and localized it is. On the other hand, it requires the site administrator to be able to provide a valid baseline for the comparison and keep it constantly updated. Yet, nowadays, most web resources are built on the fly dynamically, often by aggregating pieces of information from different sources, thus making this approach quite difficult to exploit in practice.

For our purpose we use GP within a broader framework proposing a significantly different approach to this problem, based on anomaly detection [7]. This specialized tool monitors a collection of web pages, that are typically remote, hosted by different organizations and whose content, appearance, degree of dynamism are not known a priori. For each page, we execute a *learning phase* for constructing a profile that will then be used in the *monitoring phase*. When a reading does not fit the profile, the tool raises an alert – which could trigger the sending of a notification to the administrator of the page. The tool is modular in the sense that it delegates the details of learning and monitoring to pluggable modules. In the context of this paper, GP is the technology that we have used for designing and implementing such modules. Full details about the tool can be found in the cited paper, we provide here only the context necessary for this work.

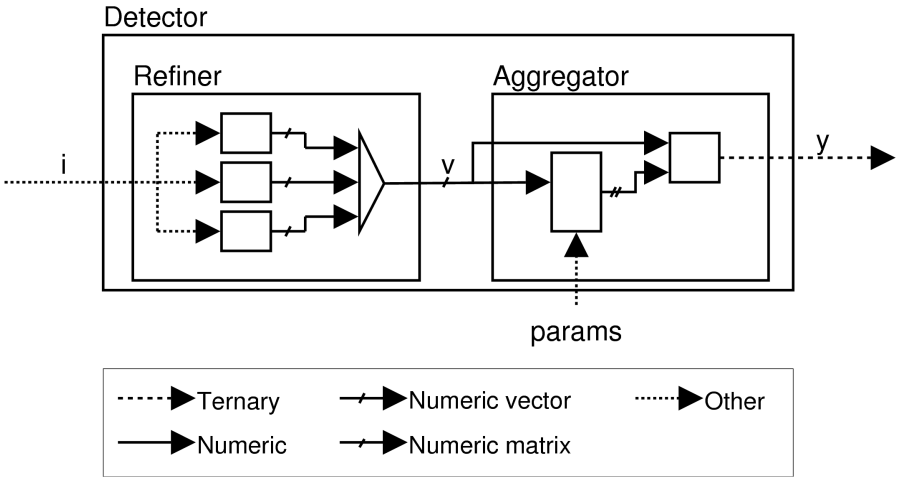


Figure 4.7 - Detector architecture.
Different arrows types correspond to different type of data.

4.2.2 Experimental Scenario

We consider a source of information producing a sequence of *readings* $\{i_1, i_2, \ldots\}$ which is input to a *detector* (Figure 4.7). The detector will classify each reading as being either normal (*negative*) or anomalous (*positive*). The detector consists internally of a *refiner* followed by an *aggregator*. In our scenario the source of information is a web page, univocally identified by an URL, and each reading consists of the document downloaded from that URL.

The refiner implements a function that takes a reading i and produces a fixed size numeric vector $v=R(i)$. In our case the transformation involves evaluating and quantifying many features of a web page related to both its content and appearance (e.g., number of external links, relative frequencies of HTML elements, and so on). The refiner is internally composed by one or more *sensors*. A sensor S is a component which receives as input the reading i and returns a fixed size numeric vector v_S. The output of the refiner is composed by concatenating the output of all sensors. Our refiner produces a vector $v=R(i)$ of 1466 elements, obtained by concatenating the outputs from 43 different sensors. Sensors are functional blocks and have no internal state, that is, $v=R(i)$ depends only on the current input i and does not depend on any prior reading.

The aggregator is the core component of the detector and it is the one that actually implements the GP approach. In a first phase, that we call the *learning phase*, the aggregator collects a sequence of readings in order to build the *learning sequence* S_{learning}.

The GP process is applied on the learning sequence, as described below, with the purpose of obtaining an individual suitable for the detection task; during this phase, the aggregator is not able to classify readings. In a second phase, the *monitoring phase*, the aggregator uses the individual obtained earlier to analyse the current reading. In the monitoring phase, for each reading i_k the aggregator may return either $y_k=$*negative* (meaning the reading is normal) or $y_k=$*positive* (meaning the reading is anomalous).

We actually applied the GP approach at the end of the learning phase. Each individual implements a function $F(v)$ of the output v of the refiner, for example (v^i denotes the i-th component of v):

$$F(v)=12-max(v^{57},v^{233})+2.7\mathrm{v}^{1104}-\frac{v^{766}}{v^{1378}} \tag{4.3}$$

The output of the aggregator for a given reading i_k is defined as follows (v_k denotes the refiner output for the reading i_k):

$$y_k=\begin{cases} negative & \text{if } F(v_k)<0 \\ positive & \text{if } F(v_k)\geq 0 \end{cases} \tag{4.4}$$

Individuals, i.e., formulas, of the population are selected basing on their ability to solve the detection problem, which is measured by the fitness function. In this case, we want to maximise the aggregator ability to detect attacks and, at the same time, minimise the number of wrong detections. For this purpose, we define a fitness function in terms of *false positive rate* (FPR) and *false negative rate* (FNR), as follows:

(i) we build a sequence S_{learning} of readings composed by readings of the observed page (S_l) and a sample set of attacks readings;

(ii) we count the number of false positives – i.e., genuine readings considered as attacks – and false negatives – i.e., attacks not detected – raised by F over S_{learning}, thus computing the respective FPR and FNR;

(iii) finally, we set the fitness value $f(F)$ of the individual F as follows:

$$f(F)=\text{FPR}(S_{\text{learning}})+\text{FNR}(S_{\text{learning}}) \tag{4.5}$$

This fitness function defined above is used to select the best individuals and proceed the evolution cycle, until to meet either of the following termination criteria:

(i) an ideal formula F with $f(F)=0$ is found.

(ii) or the evolutionary process reaches the predefined upper limit for the number of generation, i.e. more than $g_{\max}=100$.

In the latter case, the individual with the best (lowest) fitness value is selected.

The problem definition for the GP process relies on a terminals set T_S composed of a specified set of constants $C_S=\{0, 0.1, 1\}$, a specified set V_S of independent variables corresponding to the output vector v of the refiner. The functions set is based on a specified set of functions $F_S=\{+, -, \times, /, \leq, \geq, min, max, \text{-}\}$ where - represents the unary minus, and $\leq$ and $\geq$ returns 1 or 0 depending on whether the relation is true or not. All functions take two arguments, except for the unary minus.
We experimented with different subsets of V_S and F_S in order to gain insights into the actual applicability of GP to this task. For choosing the elements of V_S, we applied a *feature selection* algorithm for deciding which elements of v should be included in V_S. Note that elements in v not included in V_S will have no influence whatever on the decision taken by the aggregator, because no individual will ever include such elements. The feature selection algorithm is aimed at selecting those v elements which seem to indeed have significance in the decision and works as follows.

Let $S_{\text{learning}}=\{v_1, \ldots, v_n\}$ be the learning sequence, including all genuine readings and all simulated attacks.
Let X_i denote the random variable whose values are the values of the i-th element of v (i.e., v^i) across all readings of S_{learning}. Recall that there are 1466 such variables according to the size of v. Let Y be the random variable describing the desired values for the aggregator: $Y=0$, $\forall$ genuine reading $\in S_{\text{learning}}$; $Y=1$ otherwise, i.e., $\forall$ simulated attack reading $\in S_{\text{learning}}$. We computed the absolute correlation c_i of each X_i with Y and, for each pair $\langle X_i, X_j \rangle$, the absolute correlation $c_{i,j}$ between X_i and X_j. Finally, we executed the following iterative procedure, starting from a set of unselected indexes $I_U=\{1, \ldots, 1466\}$ and an empty set of selected indexes $I_S=\emptyset$:

(i) we selected the element $i \in I_U$ with the greatest c_i and moved it from I_U to I_S;

(ii) $\forall j \in I_U$, we set $c_j := c_j - c_{i,j}$. We repeated the two steps until a predefined size s for I_S is reached. Set V_S will include only those elements of vector v whose indexes are in I_S.

In other words, we take into account only those indexes with maximal correlation with the desired output (step (i)), attempting to filter out any redundant information (step (ii)).

4.2.3 Experiments and Results

4.2.3.1 Data-set

In order to perform our experiments, we built a dataset as follows. We observed 15 web pages for about one month, collecting a reading for each page every 6 hours, thus totalling 125 readings for each web page. These readings compose the *negatives sequences* — one negative sequence $S_{N,p}$ for each page p: we visually inspected them in order to confirm the assumption that they are all genuine. The observed pages differ in size, content and dynamism and include pages of e-commerce web sites, newspapers web sites, and alike. They are the same pages that we observed in [8]. Then we built a single *positives sequence* S_P composed by 75 readings extracted from a publicly available defacements archive (http://www.zone-h.org).

4.2.3.2 Methodology

In order to set a baseline for assessing the performance of GP, we injected the very same dataset to another aggregator that we developed earlier, not based on GP [8]. This aggregator implements a form of anomaly detection based on domain-specific knowledge. In short, it builds a profile of the observed page from a set of normal observations; then, it signals an anomaly whenever the actual observation of the page deviates from the profile, on the assumption that any anomaly represents evidence of an attack. We can point out that this notion of "profile" encompasses many different points of view. For example, mean and standard deviation of the number of lines; set of images or links contained in every reading; subtree of the HTML tree contained in every reading. This aggregator raises an alert depending on number and types of deviations from the profile (see the cited paper for more details). Note that this aggregator makes use of a learning sequence that does not include any attack, thus it does not exploit any information related to positive readings. The GP-based aggregator, in contrast, does use such information. Note also that our existing aggregator takes into account all the 1466 elements output by the refiner, whereas GP-based aggregator considers only those elements chosen by the feature selection algorithm.

We generated 25 different GP-based aggregators, by varying the number of selected features s in 10, 20, 50, 100, 1466 (thus including the case in which we did not discard any feature). The different functions sets used in the experiments are reported in Table 4.3.

F_{S1}	$+$, $-$
F_{S2}	$+$, $-$, $\times$, $/$, -
F_{S3}	$+$, $-$, $\times$, $/$, $\leq$, $\geq$, -
F_{S4}	$+$, $-$, $\times$, $/$, *min*, *max*, -
F_{S5}	$+$, $-$, $\times$, $/$, $\leq$, $\geq$, *min*, *max*, -

Table 4.3 - The five functions sets used in the experiments.

We used FPR and FNR as performance indexes, that we evaluated as follows. First, we built a sequence S'_P of positive readings composed by the first 20 readings of S_P. Then, for each page p, we built the learning sequence S_{learning} and a testing sequence S_{testing} as follows.

(i) We split $S_{N,p}$ in two portions S_l and S_t, composed by 50 and 75 readings respectively.

(ii) We built the learning sequence S_{learning} appending S'_P to S_l.

(iii) We built the testing sequence S_{testing} appending S_P to S_t.

(iv) Finally, we tuned the aggregator being tested on S_{learning} and we tested it on S_{testing}. To this end, we counted the number of false negatives — i.e., missed detections — and the number of false positives — i.e., legitimate readings being flagged as attacks.

As already pointed out, the anomaly-based aggregator executes the learning phase using only S_l and ignoring S'_P.

In the next sections we present FPR and FNR for each aggregator, averaged across the 15 pages of our dataset. GP-based aggregators will be denoted as GP-s-F_{Si}, where s is the number of selected features and F_{Si} is the specific set of functions being used; anomaly-based aggregator will be denoted as *Anomaly*.

4.2.3.3 Results

Table 4.4 summarizes our results. The table shows FPR, FNR and the fitness f exhibited by the individual selected to implement the GP-based aggregator (the meaning of the three other columns is discussed below). The aggregator with best performance, in terms of FPR + FNR, is highlighted. It can be seen that the GP process is quite robust with respect to variations in s and F_S.
Almost all GP-based aggregators exhibit a FPR lower than 0.36% and a FNR lower than 1.87%. The anomaly-based aggregator — i.e., the comparison baseline — exhibits a slightly higher FPR (1.42%) and a slightly lower FNR (0.09%). In general, the genetic programming approach seems to be quite effective for detecting web defacements.

Aggregator	**FPR**	**FNR**	f	g	t_s	t_h
Anomaly	1.42	0.09	-	-	-	-
GP-10-F_{S1}	0.00	0.71	0.0	1.0	17.0	2.7
GP-10-F_{S2}	0.09	0.98	0.0	1.0	23.3	3.7
GP-10-F_{S3}	0.09	0.62	0.0	1.1	20.4	3.5
GP-10-F_{S4}	4.53	0.44	0.0	1.0	20.7	3.7
GP-10-F_{S5}	0.09	0.89	0.0	1.0	27.9	3.9
GP-20-F_{S1}	0.09	1.16	0.0	1.0	18.2	2.6
GP-20-F_{S2}	0.18	1.33	0.0	1.0	12.8	2.4
GP-20-F_{S3}	0.36	0.80	0.0	1.0	20.1	3.1
GP-20-F_{S4}	0.09	0.89	0.0	1.0	36.5	4.2
GP-20-F_{S5}	0.00	0.89	0.0	1.0	39.5	3.9
GP-50-F_{S1}	0.00	1.24	0.0	1.0	5.1	1.6
GP-50-F_{S2}	0.09	0.98	0.0	1.0	20.4	2.9
GP-50-F_{S3}	0.36	0.98	0.0	1.0	19.3	2.9
GP-50-F_{S4}	0.18	0.89	0.0	1.0	15.4	3.1
GP-50-F_{S5}	**0.18**	**0.27**	**0.0**	**1.0**	**29.4**	**3.0**
GP-100-F_{S1}	0.09	1.16	0.0	1.0	15.5	2.1
GP-100-F_{S2}	0.09	1.33	0.0	1.1	11.4	2.2
GP-100-F_{S3}	0.00	1.87	0.0	1.3	14.1	3.1
GP-100-F_{S4}	0.09	0.27	0.0	1.1	18.7	3.1
GP-100-F_{S5}	0.09	1.33	0.0	1.2	15.4	2.6
GP-1466-F_{S1}	0.00	0.80	0.0	1.0	8.9	1.9
GP-1466-F_{S2}	0.18	0.44	0.0	1.0	6.1	2.3
GP-1466-F_{S3}	0.18	0.98	0.0	1.2	5.3	1.5
GP-1466-F_{S4}	0.18	1.87	0.0	1.2	11.2	1.8
GP-1466-F_{S5}	3.73	0.18	0.0	1.4	9.9	2.1

Table 4.4 - Performance indexes. FPR, FNR and f are expressed in percentage.

We analyzed GP-based aggregators also by looking at the number of generations g that have evolved for finding the best individual and the complexity of the corresponding abstract syntax tree, in terms of number of nodes t_s and height t_h (these data are shown in Table 4.4). We found that formulas tended to be quite simple, i.e., the corresponding trees exhibited low t_s and t_h. We also found, to our

surprise, that g=1 in most cases.
This means that generating some random formulas (500 in our experiments) from the available functions and terminals sets suffices to find a formula with perfect fitness – i.e., one that exhibits 0 false positives and 0 false negatives over the learning sequence. We believe this depends on the high correlation between some elements of the vector output by the refiner (i.e., some v^i) and the desired output. Since the feature selection chooses elements based on their correlation with the desired output, most of the variables available to GP will likely have an high correlation with output.

Our domain knowledge, however, suggests that the simple formulas found by the GP process could not be very effective in a real scenario. Since they take into account very few variables, an attack focussing on the many variables ignored by the corresponding GP aggregators would go undetected. This consideration convinced us to develop a more demanding testbed, as follows.

4.2.3.4 Results with "shuffled" Data-set

In this additional set of experiments, we augmented the set of positive readings for any given page p_i by including genuine readings of *other* pages. While the previous experiments evaluated the ability to detect manifest attacks (defacements extracted from Zone-H), here we also test the ability to detect innocent-looking pages that are different from the usual appearance of p_i. More in detail, for a given page p_i we defined a sequence S^o_{learning} composed by 14 genuine readings of the *other* pages (one for each other page) and a sequence S^o_{testing} composed by 70 readings of the other pages (5 readings for each other page, such that S^o_{learning} and S^o_{testing} have no common readings). Then, we included S^o_{learning} in S_{learning} and S^o_{testing} in S_{testing} (we omit the obvious details for brevity). Clearly, readings in S^o_{learning} are labelled as positives and readings in S^o_{testing} should raise an alarm.

Table 4.5 presents the results for this testbed. The anomaly-based aggregator now exhibits a slightly higher FNR; FPR remains unchanged, which is not surprising because this aggregator uses only the negative readings of the learning sequence and these are the same as before. We note that the anomaly-based aggregator is very effective also in this new setting, in that it still exhibits a very low FPR while being capable of flagging as positive most of the genuine readings of *other* pages.

Aggregator	**FPR**	**FNR**	f	g	t_s	t_h
Anomaly	1.42	2.39	-	-	-	-
GP-10-F_{S1}	9.87	2.48	0.4	35.5	83.8	7.2
GP-10-F_{S2}	9.24	1.84	0.0	14.3	69.1	7.5
GP-10-F_{S3}	9.24	1.29	0.1	35.7	66.4	8.0
GP-10-F_{S4}	11.38	1.98	0.1	24.1	93.1	7.9
GP-10-F_{S5}	7.73	1.52	0.1	30.5	66.1	6.9
GP-20-F_{S1}	23.38	2.30	0.1	16.9	54.2	5.3
GP-20-F_{S2}	16.44	1.61	0.0	10.8	68.3	6.2
GP-20-F_{S3}	13.51	1.33	0.0	16.4	52.1	6.2
GP-20-F_{S4}	17.87	1.38	0.0	14.6	56.3	6.3
GP-20-F_{S5}	17.87	0.55	0.0	19.5	70.4	6.5
GP-50-F_{S1}	14.76	1.56	0.1	23.7	43.1	4.4
GP-50-F_{S2}	13.16	1.61	0.0	4.7	45.0	5.4
GP-50-F_{S3}	18.58	0.83	0.0	11.7	32.4	5.4
GP-50-F_{S4}	7.56	1.52	0.0	12.5	41.1	6.0
GP-50-F_{S5}	11.47	1.66	0.0	16.1	71.2	6.8
GP-100-F_{S1}	0.62	2.30	0.0	29.4	51.5	4.5
GP-100-F_{S2}	5.51	1.38	0.0	10.5	25.6	4.1
GP-100-F_{S3}	6.93	0.55	0.0	16.4	33.9	4.8
GP-100-F_{S4}	5.87	1.61	0.0	10.2	40.3	5.1
GP-100-F_{S5}	12.09	1.52	0.0	17.7	31.9	5.2
GP-1466-F_{S1}	0.18	1.38	0.0	21.0	37.4	3.8
GP-1466-F_{S2}	0.44	1.10	0.0	18.5	24.8	4.1
GP-1466-F_{S3}	**0.71**	**0.64**	**0.0**	**15.1**	**30.1**	**4.7**
GP-1466-F_{S4}	5.69	1.06	0.0	19.7	27.9	4.7
GP-1466-F_{S5}	0.98	1.24	0.0	16.5	69.9	5.5

Table 4.5 - Performance indexes with the new testbed. FPR, FNR and f are expressed in percentage.

Concerning GP-based aggregators, Table 4.5 suggests several important considerations. In general the approach seems now to be influenced by the number s of variables selected: taking more variables into account lead to better performance, in terms of FPR + FNR. Interestingly, the best result is obtained with s=1466, i.e., with *all* variables available to the GP process. Moreover, there are situations in which the

fitness f of the selected formula is no longer perfect (i.e., equal to 0). This means that, in these situations, the GP process is no longer able to to find a formula that exhibits $f = \text{FPR} + \text{FNR} = 0$ on the learning sequence. Note that this phenomenon tends to occur with low values of s. Finally, values of t_s, t_h and, especially, g, are greater than in the previous testbed, which confirms that GP process is indeed engaged. Several generations are necessary to find a satisfactory formula and the formula is more complex than those previously found, in terms of size and height of the corresponding abstract tree.

Figure 4.8 shows results of Table 4.5 in a graphical way and visually confirms that the GP approach is quite robust in respect to the specific set of functions being used but is sensible to the number s of selected features.

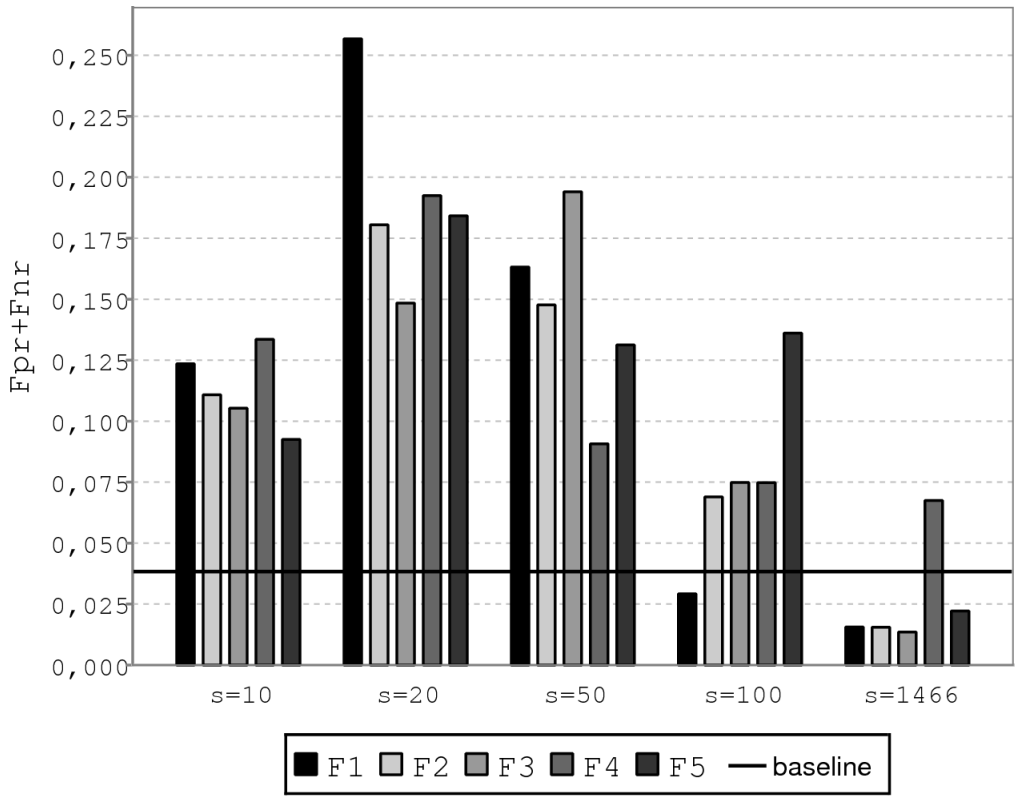


Figure 4.8 - Sum of FPR and FNR for different parameter combinations.

Finally, we compared the computation times for learning and monitoring phases obtained with GP-based approach against those of anomaly-based approach. The former takes about 100 secs for performing the tuning procedure (of which about 5 secs are used for the features selection) and about 500 µsecs for evaluating one single reading in the monitoring phase; the latter takes about 10 msecs for the tuning procedure and about 100 µsecs for evaluating one single reading in the monitoring phase. These numbers are obtained with a dual AMD Opteron 64 with 8GB RAM running a Sun JVM 1.5 on a Linux OS.

4.2.4 Summary

We have proposed and evaluated experimentally an approach based on genetic programming for detecting automatically defacements of web pages. What makes this problem difficult is that web pages are highly dynamic and the degree of dynamism change widely across pages. The main power of GP lies in its ability to construct automatically algorithms capable of describing the dynamic nature of a given web page without any domain-specific knowledge. We tested a prototype over a selection of 15 highly dynamic web pages that we observed for about a month and found that this approach is indeed practical: it is able to detect nearly all of the attacks that we simulated, while keeping the number of false positives sufficiently low to be practical. The approach exhibits performance close or better than other approaches that we pursued in the past, whose design required a considerable amount of domain-specific knowledge.

4.3 Stiffness Estimation

Stiffness is the property of an elastic body to resist to deformations when one or more forces are applied on this body. The stiffness problem consists in the approximation of the stiffness matrix values as accurately as possible. We use genetic programming to symbolically regress each element on the diagonal of the stiffness matrix. We compare the real stiffness values with those estimated with the formulas found by GP.
Our results show genetic programming is an effective approach for this task and could be part of the toolbox of many engineers. Moreover the formulas obtained with the GP process could give better insights on the stiffness phenomenon.

First we give a brief definition of the stiffness concept. In Section 4.3.2 we describe the experimental procedure used to discover new formulas which estimate the stiffness values. Section 4.3.3 discusses the performances of the formula found by genetic programming. Finally, we provide conclusions, and possible future directions of improvements related to this application.

4.3.1 Stiffness Estimation Problem

Stiffness is the resistance of an elastic body to deflection or deformation by an applied force. This property is dependent on the material and its shape, but also to the boundary conditions applied to this body.

The stiffness k of a body which is deformed on a distance δ under an applied force F is calculated as follows:

$$k = \frac{F}{\delta} \tag{4.6}$$

Stiffness is usually measured in newtons per metre.

As both the applied forces and deformations are vectors (respectively $\vec{F}$ and $\vec{\delta}$), then their relationship is characterized by a stiffness matrix K where:

$$\vec{F} = K \cdot \vec{\delta} \tag{4.7}$$

In a complex structure, deformations will generally not occur to the same point where the force is applied and will not follow the same direction as the applied force. A stiffness matrix enables to characterize such complex systems straightforwardly.

A body may also have a rotational stiffness, where the stiffness k for a rotation θ under an applied moment M is given by:

$$k = \frac{M}{\theta} \tag{4.8}$$

In the International System of Units, rotational stiffness is measured in newton-metres per radian.

We will estimate the stiffness values associated with a simple bearing as illustrated in Figure 4.9.

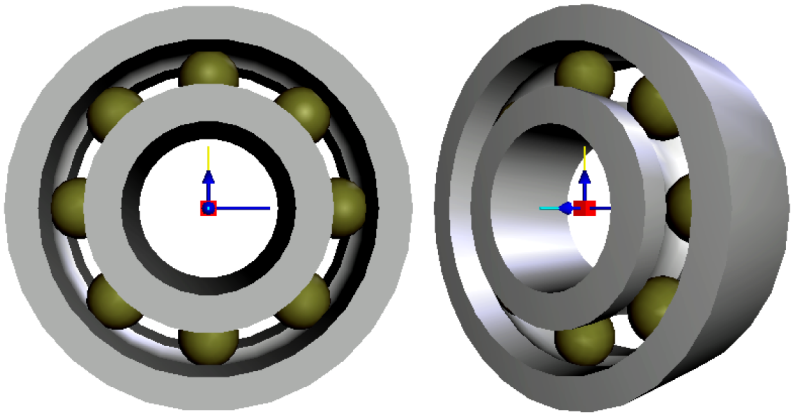


Figure 4.9 - Deep Groove Ball Bearing

We apply a combination of two loads on this bearing, the first one is a radial force F_r (vertical axis on the Figure 4.9), and the second one is an axial force F_z corresponding to the horizontal axis orthogonal to the bearing. Rotation speed is constant.
The vector $\vec{F}$ is composed of three forces, one for each axis and two moments. We obtain a 5 by 5 stiffness matrix. Since the purpose of this chapter is to investigate genetic programming capabilities we will consider only the three first elements on the diagonal of the matrix, those corresponding to the force component. However, it is clear that the methodology can be applied unchanged to all the other elements of the matrix.
Finding a mathematical model able to extrapolate new data points outside a discrete set of known data points, when the number of experimental results is limited, is particularly important in an industrial context.

4.3.2 Experimental Procedure

We calculate the stiffness values for our training set with the SKF bearing beacon software. Bearing beacon allows the modeling in a 3D graphic environment of generic mechanical systems like gear boxes with a precise bearing model for an in-depth analysis of the system behavior in a virtual environment. More details on this product can be found on the web at http://www.skf.com/portal/skf/home/products.

We transform the raw stiffness matrices provided by the software in three training sets. Each of them is composed of 1201 fitness cases corresponding to all combinations of the integer values taken in the interval [0, 1000] by step of 20 assigned to each combination of input's variables F_r, F_z and the output associated with one element of the stiffness matrix diagonal. The fitness function computes the error standard deviation on the training set i.e. the root of the mean of the squared distances between the desired values and those obtained with the program considered (cf. Equation 2.3).

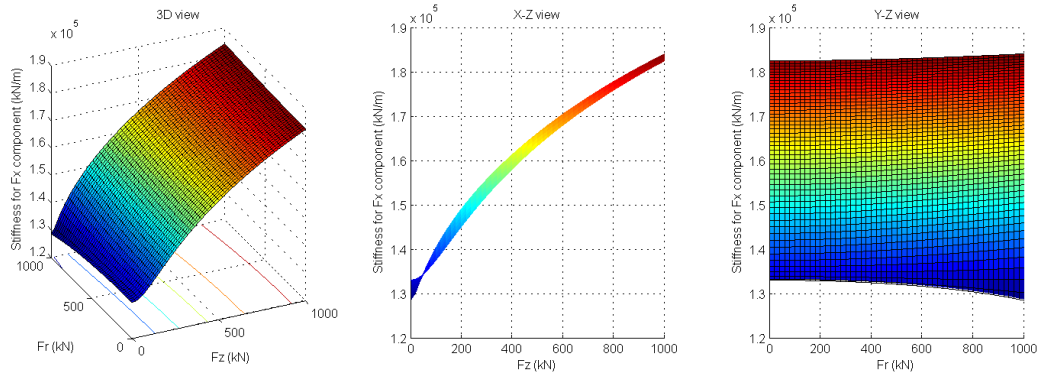


Figure 4.10 - Stiffness for the F_x component

Figure 4.10 shows the surface to approximate for the first element in the stiffness matrix diagonal (force component on axis x) in a three-dimensional view and in a X-Z and Y-Z plane view. Figures related to the two others element to approximate can be found in Appendix A.

We indicate functions and terminals set for each problem in Table 4.6. Ephemeral Random Constants (ERC) are integer values randomly chosen in the interval [0, 10].

Terminal set	F_r, F_z, 0, 0.1, 1, ERC
Function set	+, -, ×, /

Table 4.6 - Terminals and functions sets.

The parameters common to all GP runs are summarized in the Table 4.7.

Parameter	Setting
Population size	2000
Selection	Tournament of size 7
Initialization method	Ramped Half-and-Half
Initialization depths	2-5 levels
Maximum depth	7
Internal node bias	90% internals, 10% terminals
Elitism	2
Duplication rate	5%
Crossover rate	85%
Mutation rate	10%

Table 4.7 - Parameters settings.

Each test is the result of 50 independent executions. Each execution started with a different seed for the random number generator.
We ran all simulations using a machine architecture based on a processor Intel Centrino Duo T5600 1.83 GHz with 1 GB of RAM. Each run takes approximately 20 minutes to be completed.

4.3.3 Results

To assess the results obtained by the formulas found by GP, we focus on the following metrics:

- ***Standard deviation error or RMSE***, it is also the fitness function which evaluates the candidate solutions, it quantifies the difference between the expected value and the response predicted by the model and is used to determine whether the model fit or not the data.
- ***Correlation coefficient*** indicates the strength and direction of a linear relationship between two random variables.

We report in Table 4.8 the results for the stiffness approximation of each element of the stiffness matrix diagonal. To gain further insights into the quality of the solutions, in particular regarding the dispersion along the regression line, we plot the expected stiffness values versus the estimated ones in the scatter charts reported in Figure 4.11.

	RMSE	**Correlation Coefficient**
Stiffness for the F_x component	442	0.9995
Stiffness for the F_r component	749	0.9989
Stiffness for the F_z component	202	0.9997

Table 4.8 - RMSE and correlation coefficients for stiffness approximation

We note that GP exhibits a very good estimation for each stiffness surface since the correlation coefficient is always greater than 0.99. Another important result is that GP found stable solutions without outliers points along the regression line as it can be seen on the scatter charts. This result is significant because it suggests that GP provides robust solutions for this problem.
We can also note that when low radial and axial forces are applied on the bearing, the formulas found by GP are less accurate (cf. Figure 4.11). It could mean that when low forces are applied we are in a transient state and approximating the stiffness until a certain threshold for low forces requires a distinct formula.

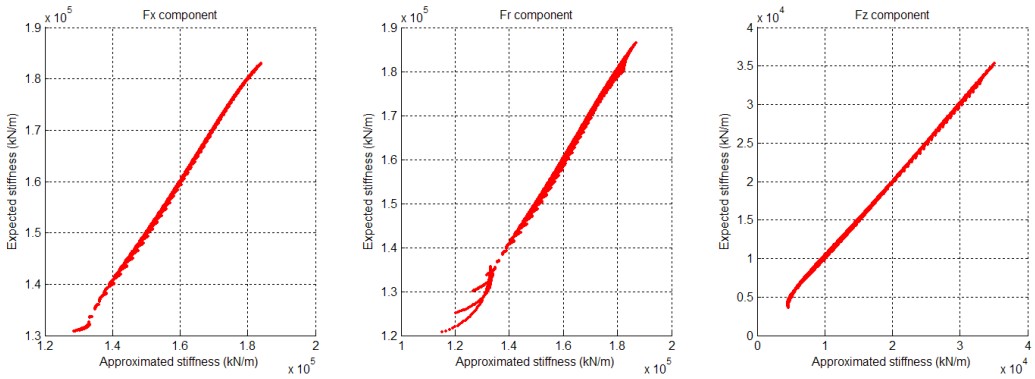


Figure 4.11 - Expected versus estimated stiffness values for each force component.

4.3.4 Summary

The solutions provided by the GP are able to approximate very closely the stiffness surface without producing outliers points. However GP is not already able to make the distinction between the transient state and the normal state, works should be done to improve efficiency of GP for this kind of phenomenon.

4.4 Concluding Remarks

Several lessons can be drawn from the experience gained with these three applications. First, the GP approach can produce competitive results when compared with the formulas found by humans. However the GP process requires a large amount of computational resources to obtain results (Section 4.3.2) and it is difficult to determine a good subset of functions and variables to insert in the functions and terminals set (Section 4.2.3). Finally the GP approach is not able to make the distinction between the transient state and the normal state. In the following chapters we address these issues by proposing new strategies for improving the GP practicability on a wide range of problems.

Chapter 5

New Strategies for Improving Scalability

"All truths are easy to understand once they are discovered; the point is to discover them."

Galileo Galilei – Physicist, astronomer and philosopher

A complex issue in genetic programming is the ability to scale toward harder and larger problems. Several factors may affect the scalability of the algorithm. In the first part of this chapter we present these issues and the current solutions developed in the literature. In the second part we propose alternative strategies able to improve the GP approach on some of these issues.

5.1 Issues in Scaling Genetic Programming

5.1.1 Effectiveness and Computational Effort

First of all, evaluating a candidate solution is computationally expensive for two main reasons:

- A single candidate solution may be evaluated on a large number of fitness cases. (typical sizes of fitness cases sets range from 50 to 5000 fitness cases according to [6]).
- To produce acceptable results, the GP process requires large populations which evolves for a large number of generations. However, more recent works advocate for different approaches, using either populations of moderate or variable size (e.g. [32][58]), or using multiple independent short runs (e.g. [57][69]) in order to outperform a long run.

Indeed, almost all of the computational resources are consumed in the evaluation of the individuals. Moreover population size and fitness cases number influence not only the CPU time but also the memory footprint. To speed up the evaluation step, several parallel models have been proposed to exploit the intrinsic parallelism of the evolutionary algorithms: master-slave with one population, fine-grained or cellular model [29] and island-model [2][66][90]. The island model is the most popular approach. It works by dividing a population into smaller subpopulations called islands. Each island evolves on a different computer node. At fixed interval of time, copies of the best individuals found so far migrate to other islands (according to a predefined set of neighbors) where they replace the worst individuals. This model requires new parameters like the number of individuals which migrate, the migration rate, topology of the grid and so on...

Given the variability of any independent GP run and the need to perform many runs to ensure an acceptable solution is found for some problems, another approach is to address more computational resources on the populations which are most likely to yield competitive solutions. In [13] and [55], the authors propose two strategies, called beam search and pyramid search, where a GP run is terminated when it is proving to be unfruitful, and the computational resources are redistributed to runs which are performing well. These approaches are based on the assumption that it is generally possible to identify at an early stage of the evolution whether a run is predisposed to fail.
The population beam search approach initializes several populations which evolve for a predefined short number of generations. Then the populations which appeared to be predisposed to failure are either terminated or replaced by copies of populations that appear to perform well. Then, these populations are evolved again for a short number of generations. This cycle is repeated until to consume the global amount of computational resources allocated to the process. In doing so this approach attempts to maximise the chance that a successful path of evolution will be followed in at least one of the population copies.
This approach requires two additional parameters: the number of populations which are retained for duplication and the replacement interval which determines how regularly populations are terminated and replaced with copies of better performing populations.

With the pyramid search method a large number of initial populations is generated. Then, as the search progresses, more and more populations are terminated, based upon the current best fitness of individuals within the population. As a result of the

termination of populations performing poorly, the number of evolving populations is reduced from a large number of populations at the start, or base, of a pyramid search, to a small number of populations at the end point. It is hoped that the computational resources invested in evolving a larger number of populations in early stages of the pyramid search will result in more populations with a better potential to effectively solve the problem at hand. As for the beam search, the pyramid search requires two additional parameters: the number of initial populations and the pruning ratio, i.e., the fraction of remaining populations to terminate.

One could view these strategies as giving a better return on the number of evaluations performed than the standard strategy.

5.1.2 Premature Convergence

A second major problem which may occur in genetic programming is premature convergence towards a local optima. If the selection pressure is too high, assigning only to few good quality individuals high probabilities of surviving, then a population tends to contain similar individuals and the genetic diversity rapidly decreases. The suboptimal genetic material (subtrees) which might help in finding the global optimum is deleted too rapidly. On the other hand, the selection pressure cannot be chosen arbitrarily low if we want the GP algorithm to be effective. In difficult optimisation problems, suitable population size, mutation and recombination rates, and selection parameters, which influence the selection intensity, are usually not known beforehand. In [37], the authors propose an alternative selection algorithm called Fitness Uniform Selection Scheme (FUSS). The algorithm selects a fitness value uniformly in the interval $[f_{\min}, f_{\max}]$ where $f_{\min}$, $f_{\max}$ are the lowest/highest fitness values respectively in the current population. Then, the individual with fitness nearest to f is selected and a copy is added to the population, possibly after mutation and recombination. The scheme automatically creates a suitable selection pressure and preserves genetic diversity. Premature convergence is avoided in FUSS by abandoning convergence at all. Nevertheless there is a selection pressure in FUSS towards higher fitness. The probability of selecting a specific individual is proportional to the distance to its nearest fitness neighbor. In a population with a high density of unfit and low density of fit individuals, the more fit ones are effectively favored.

5.1.3 Code Growth or Code Bloat

Code growth in the individuals over time is expected when this growth is part of the process of solving a problem. For example, the GP process typically starts from populations composed of small trees, and it may be necessary for the programs to grow in complexity to solve the problem in exam. But, often code growth is not correlated with any increase in fitness, in this case we will speak about "code bloat". Thus bloat consists essentially in pieces of code (subtrees) that does not change the semantics of the program. These "useless" pieces of code are often called *introns*.
Because of its practical effects (large programs are difficult to interpret, lead to poor generalisation and are computationally expensive to evaluate), bloat has been extensively studied in the last decade. Therefore, several theories have been proposed to explain bloat, we present briefly the most recent of them below:

- **Nature of program search spaces theory** [52]: above a certain size, the distribution of fitness values does not vary with size. Since there are more long programs, the number of long programs for a given fitness is greater than the number of short programs of the same fitness. Over time GP samples longer and longer programs simply because there are more of them.
- **Crossover bias theory** [72][20]: on average, each application of subtree crossover removes as much genetic material as it inserts. So, crossover in itself does not produce growth or shrinkage. However, crossover pushes the population towards a particular distribution of program sizes (a Lagrange distribution of the second kind), where small programs have a much higher frequency than longer ones. For example, crossover generates a very high proportion of single-node individuals. In virtually all problems of practical interest, very small programs have no chance of solving the problem. As a result, programs of above average length have a selective advantage over programs of below average length. Consequently, the mean program size increases.

Several techniques to control bloat have been proposed [52][86]. For instance modifications of the variation operators with *size fair crossover* or *size fair mutation* [49], Tarpeian bloat control [70] and parsimony pressure [96].

5.1.4 Problem Decomposition

Problem decomposition aims to exploit modularities, symmetries, and regularities inherent to the problem to solve. This task is routinely done by human programmers when they organise sequences of repeated steps into reusable modules such as

subroutines, functions, and classes. Then these modules are organised in a hierarchy in which top level modules may invoke lower level ones with different parameters inputs. Problem decomposition through hierarchical programming allows a better control of complexity and facilitates code reuse. Code reuse is particularly useful for the GP process because it avoids rediscovering useful blocks of code at different places within the program where it is required. The original GP algorithm has no explicit support for module creation and code reuse. In the following sections we look at a number of techniques that have been used to enhance GP's ability to scale up to larger and more complex problems. However, these techniques have many strong limitations and decomposing a problem in an automatic way remains the "Holy Grail" of the genetic programming field.

5.1.4.1 Automatically Defined Functions

Automatically Defined Functions (ADFs) constitutes the most popular technique to evolve reusable modules [47]. With this technique the structure of an individual is composed of one or more parametrised function subroutines (i.e. ADFs) and a main program that may invoke those functions and produces the result. Typically each ADF is a separate tree; consisting of its arguments (which are the terminals) and the same functions as the main program tree (eventually plus calls to other ADFs). ADFs are evolved simultaneously and in association with the main result-producing tree.

Automatically defined functions provide a mechanism to encapsulate and reuse blocks of code which may lead to faster convergence towards the solution [78].

However use ADFs implies several limitations and drawbacks. First, for using ADFs, it is necessary to determine the number of functions that will be evolved along with the main program. Then for each ADF a user must specify the signature of the function i.e., the number of parameters this function takes, the data type for each argument and also the return type of the function. The functions set of the main program should be extended to include the automatically defined functions too.

Moreover it is necessary to modify the recombination operator, e.g. crossover to make sure that the points of crossover are chosen in such a way that, if the first crossover point, which is randomly selected, lies in an ADF then the second point crossover is chosen from the same ADF. If it is not done, a call to an ADF may become a part of the same ADF, causing an infinite loop or an infinite recursion.

5.1.4.2 Module Acquisition

An alternative to the use of ADFs is the technique called Module Acquisition (MA) proposed by Angeline and Pollack [4]. This approach introduces two additional operators in the evolutionary process:

- **compress** which selects *randomly* subtrees from the population and protects them from any further structural modification by compressing it into a module. Parts of the subtree not included in the function become its arguments. Modules are stored in a library, their value being determined by how often they are used by evolving individuals.
- **expand** which decompresses the modules and make them available for any future structural modifications.

In one comparison [43] using even-k-parity problems, it was found the ADF approach was superior to the standard non-ADF approach whereas no improvement was seen with MA.

5.1.4.3 Adaptive Representation

The Adaptive Representation through Learning (ARL) algorithm [79] addresses the random selection of the subtree of a tree in MA. ARL identifies and extracts subroutines from offspring which exhibit the best improvements over the fitness of their parents. Then the function set is extended with the selected subtrees (modules). The current population is then enriched by replacing individuals by new randomly created individuals based on the extended functions set. In their experiments they compared their approach with standard GP and GP with ADFs on even-k-parity problems of increasing difficulty (from k=3 to k=8). They report results indicating a superior performance compared to both of the other methods in terms of number of generations required to find a correct solution, and size of the solution found.

However in [19], the authors show that random selection of subtrees for reuse is more effective than other heuristics across a larger range of problems. Furthermore, it was also shown that ARL does not produce highly modular solutions and once the contents of modules are allowed to evolve they become a form of ADF.

5.1.4.4 Layered Learning

The Layered Learning (LL) approach was introduced by Stone and Veloso in [87] and applied to the specific domain of simulated robotic soccer. The LL approach consists in breaking a problem into a bottom-up hierarchy of subproblems. Each subproblem is associated with a layer in the problem-solving process. The idea is that the solutions found at lower layers when solving the simpler subproblems directly facilitates the learning required in higher layers. In layered learning GP (LLGP), evolution begins in a first layer that proceeds towards the solution of a subproblem of the overall problem. The evolution in a layer stops when either a solution for the subproblem is found, or when a predefined number of generations has elapsed. The evolutionary process is pursued into the next layer, using the genetic material of the previous layer as the initial population to evolve towards the solution of the original high-level problem.
There are various ways in which genetic material may be propagated from one layer to the next. The population in the lower layer may simply be re-cast as initial population for the next layer; or the best individuals from the lower layer may be used exclusively to seed the higher layer.

The preliminary step for using layered learning is to decompose a problem into tasks to be associated with the lower layers. Usually this decomposition is performed by identifying a lower-order form of the same problem [38] or a component sub-task. However, decomposing a problem in this way requires an extensive knowledge of the problem at hand, and also some insights into the components that would be useful in forming a solution.

5.2 Our Proposal: Reduction & Differentiation Strategy

In the previous section we presented some common methods for improving a genetic programming search on difficult problems. These methods attempt either to exploit parallelization techniques, or to prevent premature convergence with new selection operators, or to decompose the original problem in sub-problems.

5.2.1 Underlying Ideas

But recent research [16] suggests that the composition of an initial population has also a crucial influence on the probability to obtain an acceptable solution. It is pointed out that the initial population must contain a sufficient quantity of useful building blocks (i.e. subtrees) and that these blocks must be part of the fittest individuals. Moreover, it was shown in [17] that the building blocks used during the GP process are not dispersed throughout in the initial population, but are instead concentrated in a subset of individuals. Thus the effectiveness of GP to solve a problem is conditioned by its ability to create potential good individuals in the initial population and identify the individuals which are most likely to provide building blocks useful to find the solution. In the further sections we build on these existing results and on an interpretation of several concepts presented below to propose four new models focusing on the composition of the initial population.

The strategy presented in this section is based on the concept of species and speciation. A species corresponds to a population which is associated with a pool of genes. A species is polymorph, with the capacity to adapt itself to the constraints of the environment and to occupy an ecological niche. Competition between species may occur when there are neighbors and their territories overlap. The consequences are variable, either a species eliminates the other, or each species dominates a more restricted domain and their expansions are inhibited by the competition or the ecological impossibility to occupy another domain.

The emergence of a new species requires that such species is isolated from the other species as in the aphorism of Moritz Wagner (1813-1887) “Without isolation, no species”. The speciation process (or formation of species) induces a radial extension of the species in the environment and a diversification when the species meet a new ecological niche.
As will be clearer in the following sections, the selection process acts on several small populations isolated from each other, and causes a rapid transformation from an unbalanced population towards a population acquiring a new balance with its environment. The evolution progresses through three states: first the species is unadapted to its environment, then the species comes in a pre-adaptive state where the selection pressure leads the population towards a new balance, and finally the species is integrated and knows only small fluctuations in its environment.

Several factors may break this balance:

- Mutations inside an individual which are abrupt changes in the genetic information.
- Migration between the species. The populations composition is affected by the arrival or the departure of migrants which have a frequency of genes different from those of the populations they integrate or leave. They introduce a modification of the genetic pool.

If the population is small, a problem of sampling may occur. If a random genetic mutation gives an advantage to an individual, this individual will more likely spread its genetic information through its descendants. After a small number of generation the descendants will dominate the population leading to a loss of diversity in the population. This phenomenon, known as "genetic drift" should be avoided as much as possible in the evolutionary algorithms. Indeed it induces a premature convergence in a small region of the search species. For this reason, the GP practitioners use large populations to counteract the "genetic drift" but this approach involve obvious problems in terms of computational cost: more individuals to evaluate means a lot of time to wait for.

In this section we propose a model based on these observations where the speciation process is artificially recreated. In our context a species encapsulates a population and includes all individuals that share the same functions and terminals set. A species marks the boundary between populations rather than between individuals. Our approach attempts to provide a better sampling of the search space, where several differentiated populations (i.e., species) evolve in order to maximise the coverage of the search space. In a second stage we exploit the partial solutions found in the most promising regions of the search space to speed-up the convergence towards a solution. We call this strategy *Reduction & Differentiation* since each species is assigned to a reduced and distinct search space from each other (Figure 5.1). Each search space may overlap partially the surrounding search spaces but they are essentially different. The species evolve in their assigned search space for a certain number of generations. Then, in the second phase, we collect the most competitive individuals found in each region and explore the resulting search space. In other words we are able to introduce a clear separation between the exploration phase and the exploitation phase. The evolutionary process is decomposed in two distinct levels of research where the higher level combines the best partial solutions found by the lower level.

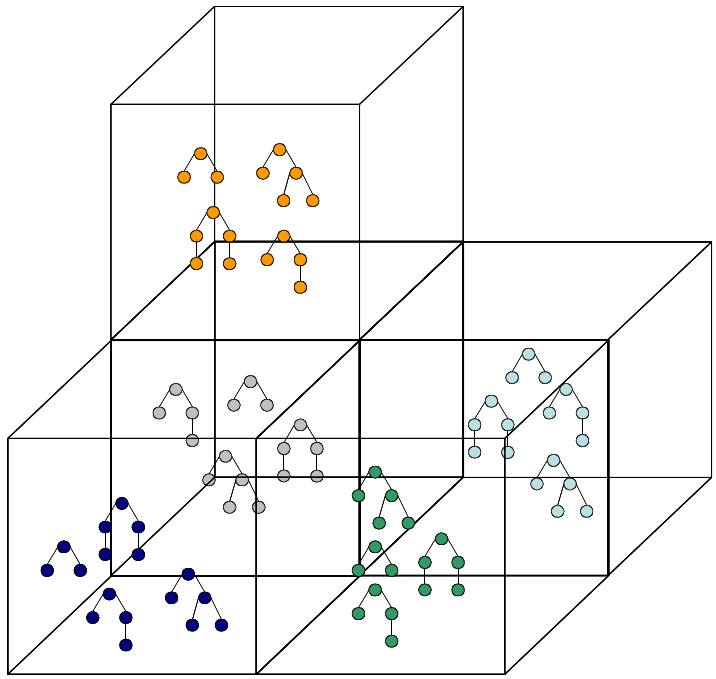


Figure 5.1 - Species assigned to different regions of the search space

5.2.2 Artificial Speciation: a First Attempt

This first model works in two successive stages. During the first stage we partition the search space in smaller regions that are explored independently of each other. We do so by generating initial populations carefully tailored to maximise the coverage of each region. Then, our algorithm collects the most competitive individuals found in each partition and exploits them in order to get a solution.

5.2.2.1 Model Description

In this section we describe step by step the working principles of our approach. Clearly, if the search finds an individual that solves the problem, the search stops immediately. For ease of description, we omit this action from the description below.

(i) A first species is associated with the original relations and variables set $R_S \cup V_S$ provided by the user. Its initial population is constructed by allowing an individual to contain elements from $R_S \cup V_S$. We denote this species as "reference species".

(ii) We apply a *reduction and differentiation* function on the relations set R_S, as follows. Let n denote the cardinality of R_S. We build all possible subsets of R_S composed of exactly p elements, where p is a parameter of the algorithm such that $p < n$. We may apply the same procedure also on the variables set V_S, or on the union of both $R_S \cup V_S$. If the reduction and differentiation is applied on R_S then the elements of V_S are added to the new subsets, if it is applied on V_S we add the elements of R_S. We denote by f_{RD} the reduction and differentiation function and by RD_{SS}^{i} $(i \in [1,n])$ the subsets generated.

(iii) Subsets obtained are checked in such a way that the type of each argument for a function from RD_{SS} may be associated with a variable or a function returning a value of same type. The reverse is also required: each value of a certain type returned by a function or a variable must find a corresponding function which accept the value as argument Moreover RD_{SS} must contain at least one element, otherwise the use of an evolutionary approach would be meaningless (the individuals would be composed of a single node, a variable).

(iv) We assign a number of species, say *speciesNumber*, equal to the number of subsets generated by f_{RD} plus one (i.e., the "reference species"), that is $speciesNumber = C_n^p + 1$. The algorithm analyzed in this section uses $p = n-1$. It follows that, in our case it will be $speciesNumber = C_n^{n-1} + 1$, hence $speciesNumber = n+1$. A species is associated with each subset generated by f_{RD}. It follows that each species operates on a relations and variables set different from the set of any other species.

(v) We construct the initial population of each species according to the relations and variables set associated with that species. The process progresses by turn. During a turn each species evolves independently of each other species for one generation. At regular interval of turns the worst performing species are stopped.

(vi) When there is no more active species, then we merge all the species and keep only the best individuals based on a tournament selection procedure. This new population then evolves as usual, i.e., until the remaining number of generations is reached.

5.2.2.2 Species Manager

The purpose of the species manager is twofold: first, monitor the performances of each species currently evolving, second, manage the global amount of resources allocated to the species in such a way that the species performing worst are stopped first.

Indexes

In order to monitor the performances of each species, we use three different indexes which are combined to rank the species from each other. These indexes attempt to capture several aspects of the fitness distribution extracted from a species population. In the following all calculations are based on the adjusted fitness values (Equation (2.5)) which lie between 0 and 1, with 1 as the best value. In the following m will denote the number of individuals in the species population.

The first index compares directly the fitness distributions of two species A and B, where the fitness values associated with the individuals are sorted from the lower fitness value to the higher.

$$I_1(f^A, f^B) = \sum_{i=1}^{m} \begin{cases} -1 & f_i^A < f_i^B \\ 1 & f_i^A > f_i^B \\ 0 & f_i^A = f_i^B \end{cases} \tag{5.1}$$

If $I_1 > 0$ then the species A is considered as better than B. If I_1 is less than 0 then the species B is better than A, otherwise they are considered as equals.

The second index measures how many distinct fitness values are present in the population. This index is similar to the information entropy defined for a discrete random variable X which can take values in $\{x_1, \ldots, x_m\}$:

$$E(X) = -\sum_{i=1}^{m} p(x_i) \log_2(p(x_i)) \tag{5.2}$$

In information theory, entropy measures the uncertainty associated with a random variable. Entropy may be also considered as the absolute limit of the best possible lossless compression for a communication channel. If we consider a message as a series of symbols, the shortest possible representation to transmit the message is the Shannon entropy in bits/symbol multiplied by the number of symbols in the original message. Entropy is zero when the random variable is "certain" to be predicted and takes a positive value otherwise. In our case we are looking for a maximum of uncertainty in the population since it means a better diversity of the population in exam.

Our index is computed in a different way but still provides the same information and with less rounding errors. First we discretize the fitness distribution as follows:

$$\forall i \in [1, m-1] \quad fd_i = \begin{cases} 1 & f_i < f_{i+1} \\ 0 & f_i = f_{i+1} \end{cases} \tag{5.3}$$

To compare the entropy between two species A and B, it is sufficient to compute the sum of the difference:

$$I_2(f^A, f^B) = \sum_{i=1}^{m} (fd_i^A - fd_i^B) \tag{5.4}$$

The comparison value given by I_2 is interpreted as for I_1.

The previous indexes are based on the characteristics of the fitness distribution related to the species at the current generation g. On the contrary the last index quantifies the improvement of a species by keeping a trace of the best fitness value found for each generation. From these historical fitness values we compute a best-fitness cumulative distribution as follows:

$$\forall\ t \in [1, g] \quad fbc(t) = fb(t+1) - fb(t) \tag{5.5}$$

Please note that the computation of this distribution begins only at the second generation, otherwise there is no difference to compute.
A comparison between two best-fitness cumulative distributions is performed by summing the difference between each element of the distributions:

$$I_3(fbc^A, fbc^B) = \sum_{t=1}^{g} (fbc_t^A - fbc_t^B) \tag{5.6}$$

The resulting value is interpreted as for I_1 and I_2.

The last operation consist in combining the three indexes using the Pareto Dominance relationship defined in Section 1.3: if a species dominates another species, its rank is decremented, in the opposite case its rank is incremented. At the end we obtain a rank for each species based on an accurate analysis of the fitness distribution and the evolution of best fitness values in the species.

Each species is compared with each other over the three indexes. For each index k we fulfill a dominance matrix which reports the result of the comparison. The value -1 indicates that a species dominates another, 1 the species is dominated and 0 indicates a non-dominance relationship. In the matrix instance reported in (5.7) the species A is dominated by the species D but dominates B.

$$DM_k = \begin{matrix} Species \\ \\ \\ \\ \end{matrix} \begin{matrix} A & B & C & D \\ \end{matrix} \begin{pmatrix} 0 & -1 & 0 & 1 \\ 1 & 0 & 1 & -1 \\ 0 & -1 & 0 & 1 \\ -1 & 1 & -1 & 0 \end{pmatrix} \begin{matrix} A \\ B \\ C \\ D \end{matrix} \tag{5.7}$$

Then we sum each row of the matrix in order to obtain the rank of the species for the index k in exam. Thus the dominance matrix becomes:

$$R_k = \begin{matrix} A & B & C & D \\ 0 & -1 & 0 & 1 \end{matrix} \tag{5.8}$$

Stopping policy

The species with the lowest rank is stopped by the species manager according to a semi-sigmoid function taking in parameters the global amount of generations G, the number of generations already done x and the total number of species *speciesNumber*. It may occur that two or more species have the same rank, in this case we randomly choose one of the species. The function returns the number of species which should be stopped after x generations as follows:

$$semiSig(x, speciesNumber, G) = \begin{cases} \dfrac{speciesNumber}{1 + e^{\left(4 - \frac{8 \cdot x}{G}\right)}} & \dfrac{x}{G} \leq 2 \\ \dfrac{x \cdot speciesNumber}{G} & \dfrac{x}{G} > 2 \end{cases} \tag{5.9}$$

The number of species to stop is determined with the difference between the value returned by *semiSig* and the number of species already stopped. Figure 5.2 shows the stopping distribution for 10 species sharing 100 generations.

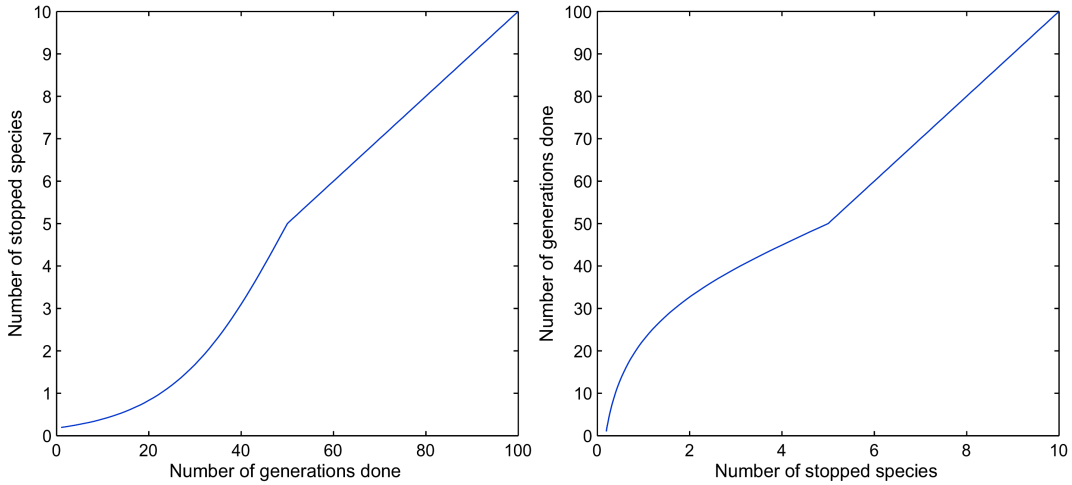


Figure 5.2 - Figure on the left shows the number of stopped species as a function of the number of generations. Figure on the right presents the inverse function.

We use a semi-sigmoid function in order to let the species evolve for a sufficient number of generations before to be stopped. With a linear function, and for a small amount of generations, some species would be stopped too early.

The species manager pursues objectives similar to the pyramid search presented in the Section 5.1.1. However it is worth to note several important differences:

- The species manager determines automatically the number of species according to the number of elements in the set on which is applied f_{RD}.
- The species are stopped, not eliminated, i.e. they participate to the selection step.
- We used three different indexes whereas the pyramid approach determines the performance of a population only with the fitness value of the best individual, which is only a partial view of the characteristics related to a population.

5.2.2.3 Model's Dynamics

An example of the algorithm's dynamics is shown in Figure 5.3 where all populations are composed of 500 individuals and the allocated number of generations for the process is $25 \cdot (speciesNumber + 1)$. We add one to the *speciesNumber* because we must also consider the final species. For this example we have chosen to reduce only the relations set. The algorithm initiates by creating 4 species which evolve independently until the species manager progressively stop them. After each turn, the

species manager stocks one generation for the evolution of the final species. When all species have been stopped, then, the algorithm selects the best individuals in the species according to their fitness values. The best 500 individuals are used to build up a new population. This population evolves either until the problem is solved, or until the upper bound of the allocated number of generations is reached.

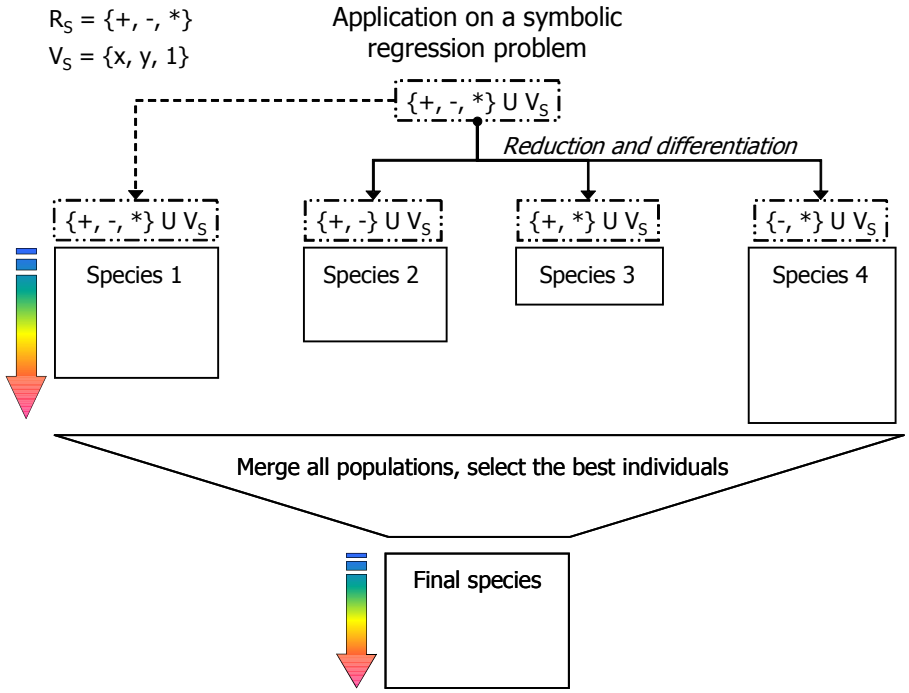


Figure 5.3 - Reduction & Differentiation strategy.

5.2.3 Experimental Design

We benchmarked our proposal on the standard test problems presented in Section 2.4. For each one we executed the following algorithms:

- *Classical GP* (CGP) For comparison purpose, we used the standard GP algorithm as described in Chapter 2.
- *Reduction & Differentiation* (R&D) strategy. For each problem we apply the Reduction & Differentiation function by reducing either the variables set, or the relations set, or the union of both. We refer to these as follows:

  (i) *R&D* $f_{RD}(R_S)$ when the reduction and differentiation function f_{RD} is applied only on R_S,

(ii) *R&D* $f_{RD}(V_S)$ when f_{RD} is applied only on V_S,

(iii) *R&D* $f_{RD}(R_S \cup V_S)$ when f_{RD} is applied on $R_S \cup V_S$.

– *Extinction & Fusion* (E&F) strategy. In order to evaluate the effectiveness of our reduction and differentiation procedure, we repeated the very same tests as those of the previous suite, but without applying f_{RD}. For example, we repeated the test *R&D* $f_{RD}(R_S)$ with the same number of species but without applying f_{RD} on R_S. The species manager works exactly as for the *R&D* strategy.

We allocated 300 generations for each run of the classical GP. That is, we stopped the evolution when the predefined amount of generations is used up. For the *E&F* and *R&D* strategy we allocate 25 generations per species to give a global amount of generations of $25 \cdot (speciesNumber + 1)$ generations. The number of generations is only an upper bound to stop the runs. Indeed, the computational cost of a generation may differ from an approach to another as explained in the following page.

Each test is the result of 100 independent executions. Each execution started with a different seed for the random number generator. Moreover, we used the same seeds for each test. We ran all simulations using a machine architecture based on a processor Xeon 3.2 GHz with 2 GB of RAM.
The parameters common to all tests are summarized in the Table 5.1.

Parameter	**Setting**
Population size	500
Selection	Tournament of size 7
Initialization method	Ramped Half-and-Half
Initialization depths	2-6 levels
Maximum depth	15
Internal node bias	90% internals, 10% terminals
Elitism	1
Duplication rate	5%
Crossover rate	80%
Mutation rate	15%

Table 5.1 - Parameters settings.

In order to compare the strategies we used three different metrics:

- *Percentage of success*, a run is considered successful if the algorithm finds an optimal solution within the amount of generations defined by the user.
- However, for some benchmarks, no optimal solutions are found. To overcome this problem we propose to consider the *average* and *standard deviation* of the best fitness values for each run.
- We measure the computational cost by logging the *time* spent for each run. This index captures the fact that each evaluation has its own cost, depending for instance on the number of nodes or the complexity of each node in that evaluation. In our context we cannot process all evaluations as having the same cost for two reasons. First, the individuals in each species are built on a different set of functions and terminals from each other. Second, our strategy perform a large number of evaluations in the first generations when the individuals are relatively simple. Obviously, the absolute value of the "time" performance index is not very meaningful: while probability of success and number of evaluations describe properties that are intrinsic to the genetic programming process, time is related to the specific hardware and software platform used. However, as we shall see, the value of the "time" performance index does provide important insights into the behavior of the algorithms when compared to each other.

5.2.4 Results and Discussion

Our strategy aims to address two issues. First, for a given quality of solution we are looking for an approach able to get an acceptable solution with the smallest amount of computational resources as possible. Second, for a given computational effort, we seek the best solution possible. In other words, on the one hand we would like to minimise the number of evaluations necessary to get a solution, and on the other hand we would like to maximise the accuracy of the solution.

5.2.4.1 Numerical Functions

Quintic function

Table 5.2 shows the average and standard deviation for the best fitness values obtained over the 100 runs for the quintic function. We note that *R&D* and *E&F* applied on $R_S \cup V_S$ are the most accurate methods.

	Best adjusted fitness values		**Elapsed time (min.)**
	Average	Standard deviation	
Classical GP	0.9938	0.0177	333
E&F $f_{RD}(R_S)$	0.9984	0.0018	134
R&D $f_{RD}(R_S)$	0.9987	0.0010	106
E&F $f_{RD}(V_S)$	0.9982	0.0018	89
R&D $f_{RD}(V_S)$	0.9983	0.0017	**63**
E&F $f_{RD}(R_S \cup V_S)$	0.9993	0.0006	180
R&D $f_{RD}(R_S \cup V_S)$	**0.9993**	0.0007	149

Table 5.2 - Results for the quintic function.

However the *R&D* strategy achieves this result within a minor amount of time. In order to get better insights on the algorithms, we compute the average of the best fitness values as a function of the computational cost (measured by the average elapsed time). For the approaches *E&F* and *R&D* we consider the best fitness values only for the "reference species", since the other species are stopped before the end of the run. For the sake of readability, we plotted only the tests that give the best probability of success for each model (*Classical GP*, *R&D*, *E&F*).

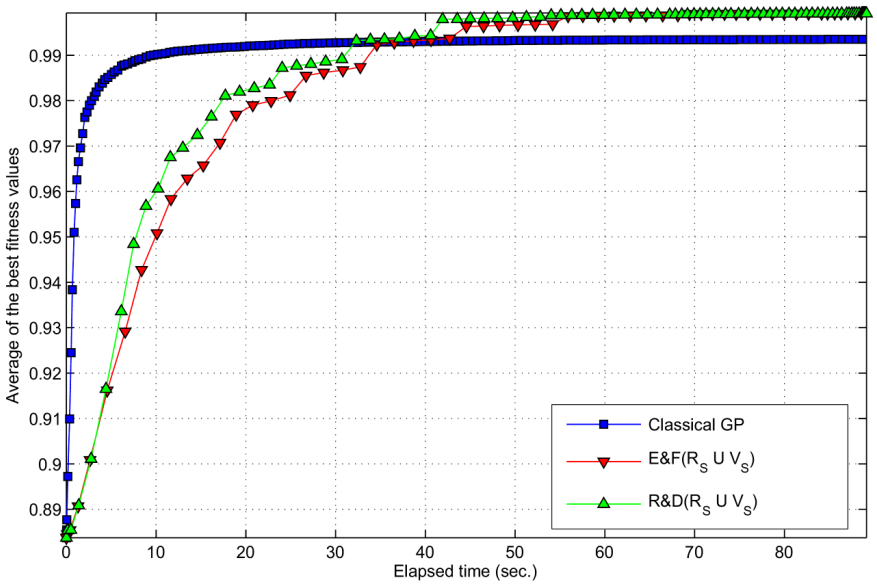


Figure 5.4 - Average of the best fitness values versus time for the quintic function.

It is worth to note that the classical algorithm reaches an upper bound earlier than the other methods, but it remains in an idle state whereas our approach continues to improve the accuracy of the candidate solutions. The same behavior can be noticed for the other functions. This phenomenon is characteristic of the premature convergence problem presented in Section 5.1.2. Both *E&F* and *R&D* strategies are able to overcome this difficulty. Moreover our strategy is able to obtain the best accuracy more quickly. A similar behavior has been found for the other benchmark functions. Results for these functions are reported in the Appendix B.

5.2.4.2 Santa Fe Ant

Table 5.3 shows the percentage of success achieved by the three algorithms for the Santa Fe ant problem. We note that our proposal improves the percentage of success of 26% when compared to the classical GP. The *E&F* strategy outperforms the *R&D* strategy but required more time to complete all runs.

	Percentage of success	**Elapsed time (min.)**
Classical GP	9	861
E&F $f_{RD}(R_S)$	**41**	524
R&D $f_{RD}(R_S)$	35	**427**

Table 5.3 - Percentage of success and elapsed time for the Santa Fe ant problem.

In order to get better insights on the algorithms, we compute the cumulative percentage of success as a function of the computational cost (measured by the average elapsed time). For example in the Figure 5.5, the classical algorithm applied to the ant problem gives a percentage of success of 5% for a computing time around 130 seconds. It appears clearly that *R&D* improves the percentage of success from a computational cost of 135 seconds.

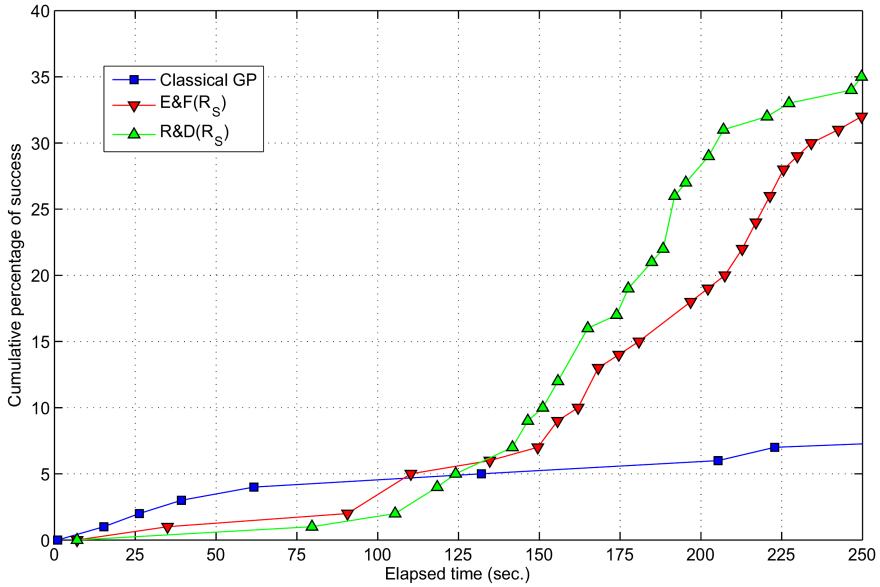


Figure 5.5 - Cumulative percentage of success versus time for the Santa Fe ant problem.

It is interesting to note that, in the Santa Fe ant problem, the classical algorithm reaches its upper bound very quickly whereas with our method the percentage of success increases regularly.

5.2.4.3 Boolean Domain

k-majority on problem

For the 5-majority problems the *R&D* approach is able to obtain 100% of success reducing dramatically the time required to obtain this result. Indeed *R&D* is 13 times faster that the best alternative technique.

	Percentage of success		**Elapsed time (min.)**	
	5 bits	7 bits	5 bits	7 bits
Classical GP	97	17	24	649
E&F $f_{RD}(R_S)$	**100**	27	39	124
R&D $f_{RD}(R_S)$	**100**	47	**3**	**104**
E&F $f_{RD}(V_S)$	**100**	45	58	155
R&D $f_{RD}(V_S)$	93	19	19	170
E&F $f_{RD}(R_S \cup V_S)$	**100**	46	91	203
R&D $f_{RD}(R_S \cup V_S)$	**100**	**57**	4	194

Table 5.4 - Percentage of success and elapsed time for the k-majority problems with k=5 and k=7.

This result is confirmed by the curves of cumulative percentage of success reported in Figure 5.6.

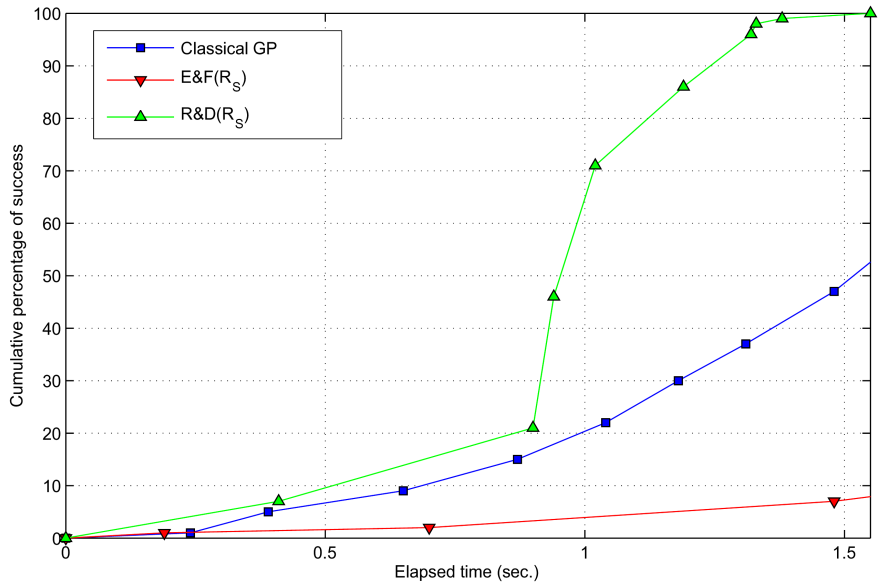


Figure 5.6 - Cumulative percentage of success versus time for 5-majority-on problem.

The same behavior is observed for the 7-majority problem, where the *R&D* strategy applied on $R_S \cup V_S$ gives the best results.

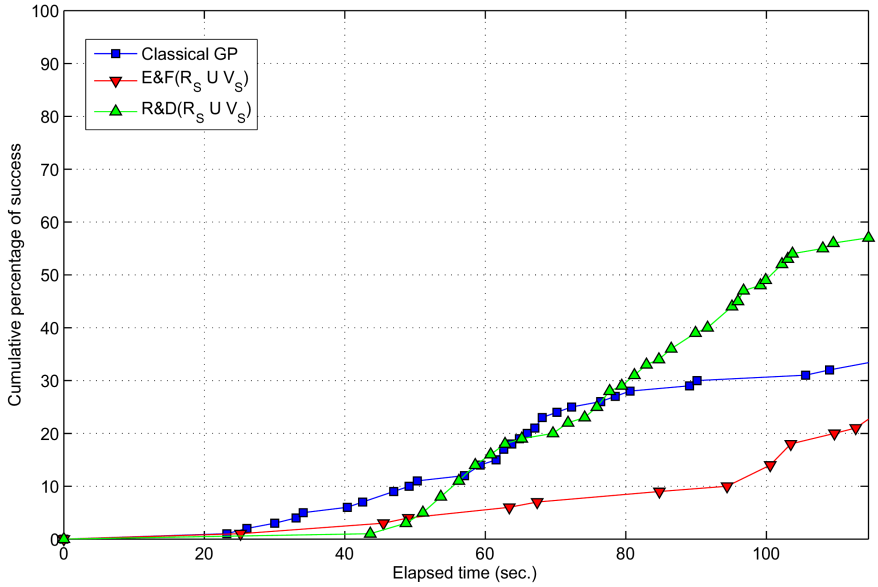


Figure 5.7 - Cumulative percentage of success versus time for 7-majority-on problem.

k-Bits Multiplexer problem

	Percentage of success		**Elapsed time (min.)**	
	6 bits	11 bits	6 bits	11 bits
Classical GP	91	14	33	2005
E&F $f_{RD}(R_S)$	**100**	20	36	427
R&D $f_{RD}(R_S)$	**100**	7	**1**	**245**
E&F $f_{RD}(V_S)$	**100**	18	63	517
R&D $f_{RD}(V_S)$	96	18	8	517
E&F $f_{RD}(R_S \cup V_S)$	**100**	**40**	107	898
R&D $f_{RD}(R_S \cup V_S)$	**100**	26	**1**	780

Table 5.5 - Percentage of success and elapsed time for the k-Bit multiplexer problems with k=6 and k=11.

For the 6-bits multiplexer, the *R&D* strategy clearly dominates the other approaches. Indeed it reaches 100% of success 36 times faster than the best alternative.
On the contrary, the best strategy for the 11-bits multiplexer is *E&F*. This fact is confirmed by the curves of cumulative percentage on the Figure 5.8.

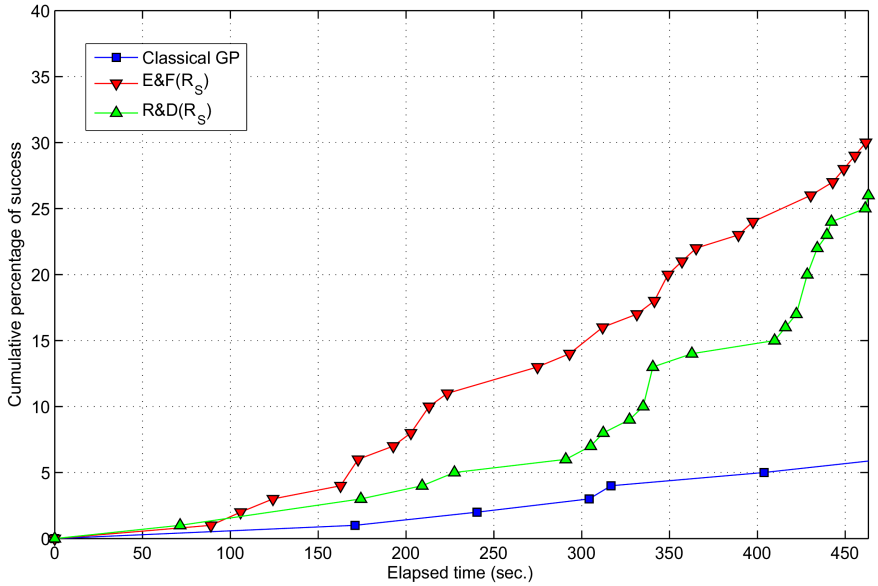


Figure 5.8 - Cumulative percentage of success versus time for 11-Bits multiplexer problem.

Even-k-parity problem

	Percentage of success		**Elapsed time (min.)**	
	4 bits	5 bits	4 bits	5 bits
Classical GP	60	**9**	253	591
E&F $f_{RD}(R_S)$	99	1	113	164
R&D $f_{RD}(R_S)$	66	1	65	118
E&F $f_{RD}(V_S)$	99	5	113	158
R&D $f_{RD}(V_S)$	35	0	**63**	**88**
E&F $f_{RD}(R_S \cup V_S)$	**100**	5	175	274
R&D $f_{RD}(R_S \cup V_S)$	85	2	72	182

Table 5.6 - Percentage of success and elapsed time for the even-k-parity problem

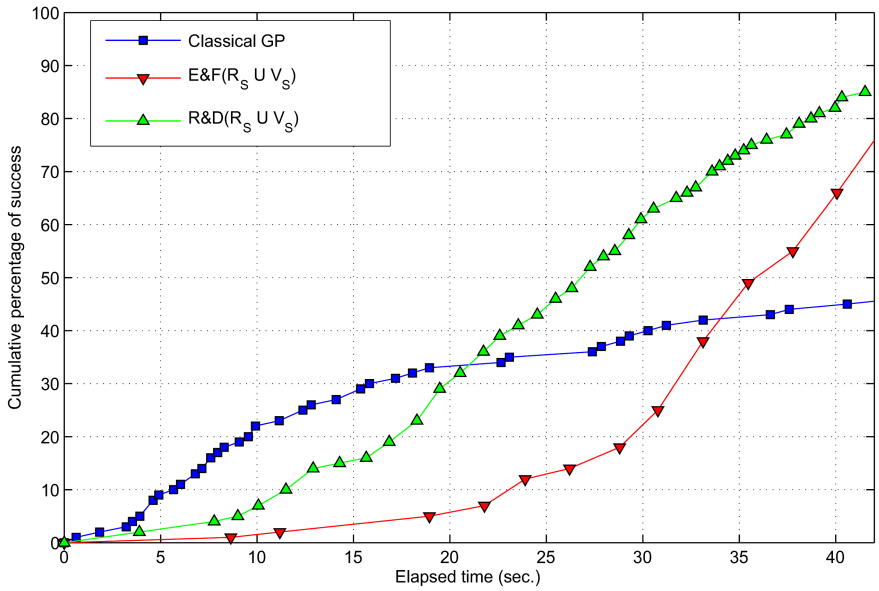


Figure 5.9 - Cumulative percentage of success for even-4-parity

For the even-4-parity problem we retrieve the same behavior as described for the ant problem. However, for the even-5-parity and the 11-bits multiplexer our approach fails to improve the percentage of success. It is worth to note that when f_{RD} is applied on V_S, we always got the worst results. To overcome this problem we propose a modification of the f_{RD} function presented in the following section.

5.3 Disambiguate the Search Space

In order to understand why the *R&D Strategy* does not perform well with the boolean problems we analyse the behavior of the algorithm on an instance of the even-3-parity.

5.3.1 Search Space Analysis

When f_{RD} is applied on V_S a variable is removed from V_S, let say D_0, but this variable directly determines the output value. Indeed several ambiguities appear in the resulting data-set reported in the Figure 5.3. For example the first row gives the opposite result of the fifth row for the same input values. Therefore the GP process is faced to a situation in which the context is insufficient to determine the output.

D_0	D_1	D_2	Even-3-parity
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Table 5.7 - Truth table for even-3-parity

In order to avoid this problem we propose to split the reduced data-set according to an heuristic able to detect the ambiguities. The heuristic has the following properties:

(i) If there are ambiguites (that is when the two vectors of independent variables have exactly the same values but associated with a different output), it guarantees that they are indeed detected.

(ii) If there are no ambiguites then it tries to found the two vectors which maximises the following ratio:

$$H_{ij} = \frac{|y_i - y_j|}{1 + \delta(\vec{v}_i, \vec{v}_j)} \tag{5.10}$$

where y_i, y_j are the expected values for fitness case i and j and δ denotes the Euclidean distance between the values of the independent variables v which have not been removed by f_{RD}.

In order to find the ambiguous fitness cases, we process the original data-set as follows:

(i) We sort the data-set according to the values of the variable to remove.

(ii) We look for a first true ambiguity: a true ambiguity is found when $H_{ij} = |y_i - y_j|$

(iii) We split the data-set according to the index j of the ambiguity found in (ii), we reapply the process (ii) on the second part of the data-set.

(iv) If no ambiguity has been found, then the data-set is divided at the index j if the fitness cases i and j maximise H_{ij}.

Moreover we consider only the data points which have:

- an absolute difference between their expected values greater than the standard deviation of y;
- and an Euclidean distance less than the average of the Euclidean distance for any couple of points.

From now, several species may share the same variables set V_S, but are associated with a different data-set. In the following we will consider these species as siblings.

To measure the influence of our disambiguation heuristic we rerun the *R&D strategy* with this modification on two instances of the even-parity problem. Of course this modification is used only when f_{RD} is applied on $R_S \cup V_S$ or on V_S, otherwise it has no influence. Moreover, at least two independent variables must be present in V_S, otherwise the procedure can not be applied.

5.3.2 Results and Discussion

In the previous tests we have seen that the best results were obtained by applying f_{RD} on $R_S \cup V_S$. It means that this method is able to provide good results in the worst case when the domain is not well understood. For all the following tests, we always put our algorithm in this situtation.

	Percentage of success		**Elapsed time (min.)**	
	4 bits	5 bits	4 bits	5 bits
Classical GP	60	**9**	253	591
E&F $f_{RD}(R_S \cup V_S)$	**100**	5	175	274
R&D $f_{RD}(R_S \cup V_S)$ (version 1)	85	2	**72**	**182**
R&D $f_{RD}(R_S \cup V_S)$	90	0	107	300

Table 5.8 - Percentage of success and elapsed time for the even-k-parity problems with k=4 and k=5

The results reported in Table 5.8 show that our disambiguation procedure does not achieve any improvements (or only few percents for the even-4-parity). This result suggests another step is missing in our strategy.
In [38] the authors show that even-parity problems are solved more easily if the same problem at a lower order of difficulty is solved beforehand. Therefore, our assumption is that the f_{RD} should be apply recursively on $R_S \cup V_S$ in order to solve first the even-parity problem for a smaller number of input variables.

5.4 R&D at N Levels: a Third Attempt

In this section we propose an extended version of the *R&D strategy* where the *reduction & differentiation* mechanism is replicated over N levels. Indeed we apply the partitioning of the search space recursively on the most promising region based on the species rank. If this new division does not generate species which outperforms the species at the higher level, then the recurrence is stopped. Along the recurrence path our algorithm collects the most competitive individuals found in each partition and evolves them in order to provide the best candidates for the higher level of division. Therefore, the tradeoff between exploration/exploitation is determined automatically as the result of the competition between species.

5.4.1 Model Description

In this new model we apply recursively the basic principle of the *R&D strategy* in order to decompose the evolutionary process in N distinct levels of search.

Using recurrence means to define two criteria common to all recursive algorithms:

- a termination criterion,
- a branch and bound operation based on some criterion.

We decided to base these criteria on the rank of the species as defined in the Section 5.3.1.2. In order to incorporate recurrence in the *R&D strategy* we need to insert two new steps, one after the step (v) and another after the step (vi):

(v') If the reference species has been stopped whereas some species are still active then we select the best active species according to their rank. This species will play the role of the "reference species", then we reapply steps (i) to (v). Otherwise, it means that no species outperforms the "reference species", and the recurrence stops. Then we replace the individuals of this "reference species"

by the best individuals found in the species created from a subset of the relations and variables set of the "reference species" and the "reference species" itself. We select these individuals with a tournament selection procedure of size 7.

(vii) After merging the populations in a single one, this new population re-evolves until the problem is solved or until all generations allocated to this level by the species manager are used.

5.4.2 Model's Dynamics

Figure 5.10 shows the successive stages of the recursive *R&D* process applied on the relations set.

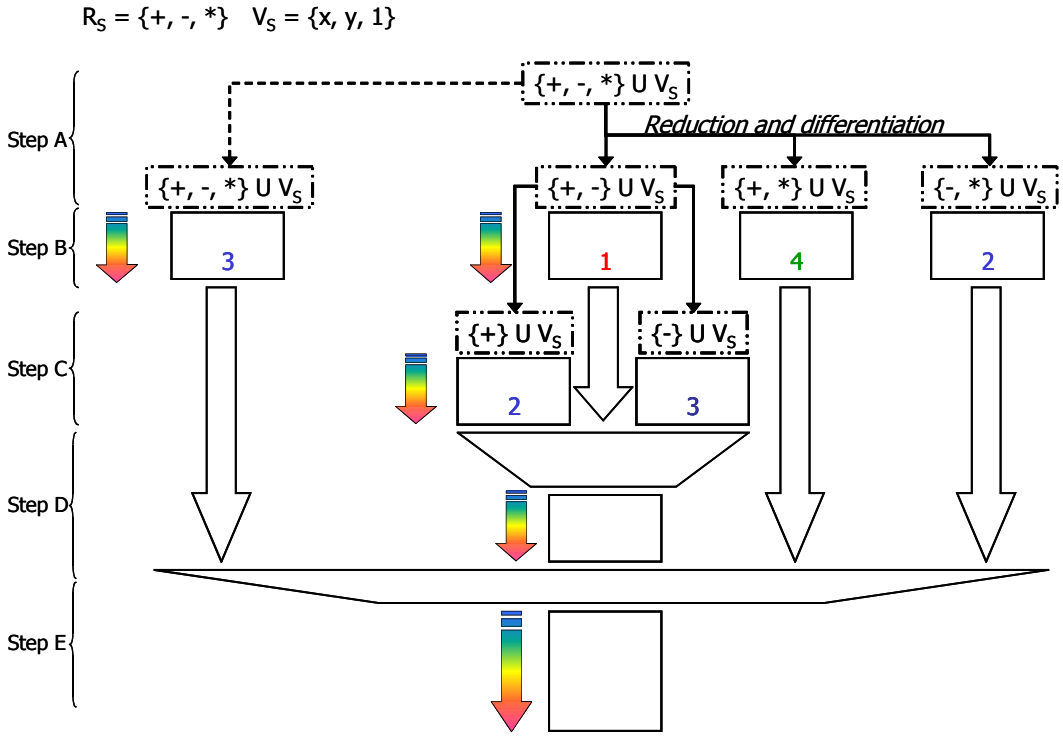


Figure 5.10 - Example of the recursive R&D strategy.

The algorithm starts by generating all possible subsets from the relations set (step A).

Then a species is assigned to the original relations and variables set and each other species is assigned to each subset and populated with individuals randomly generated

with the elements of the associated relations and variables set. Each population evolves independently until they are stopped by the species manager. On the figure, rank values are indicated in the squares representing the species.

When the reference species is stopped (step B), we reapply recursively the *R&D* strategy on the relations set associated with the best ranked species. A new species manager is created for this new level. The species manager is initialised in the same way that for the upper level. At the end of the evolution all newest species have been stopped before the reference species, thus the recurrence stops (step C).

In the step D, the species at the second level of recurrence are merging with the best species at the higher level. This species re-evolves for the remaining number of generations of this level.

Finally, the species at the first level are merging together and the best individuals are used to build up a new population. This population evolves until the problem is solved or until all generations have been used (step E).

5.4.3 Results and Discussion

In this section we focus on a comparison between the previous version of the *R&D* and the new one. However we keep the performance of *Classical GP* and *E&F* as baselines for our comparisons.

	Percentage of success		**Elapsed time (min.)**	
	4 bits	5 bits	4 bits	5 bits
Classical GP	60	9	253	591
E&F $f_{RD}(R_S \cup V_S)$	**100**	5	175	**274**
R&D $f_{RD}(R_S \cup V_S)$ (version 2)	90	0	**107**	300
R&D $f_{RD}(R_S \cup V_S)$	95	**25**	187	744

Table 5.9 - Percentage of success and elapsed time for the even-k-parity problems with k=4 and k=5.

This new version performs slightly better for the even-4-parity but requires more time as expected since a larger number of species is evolved.

For the even-5-parity, an improvement of 16% has been achieved but it requires a larger amount of time.

A careful analysis of the log records showed us that the recursive version of R&D is able to find solutions for lower order of difficulty. But the species resulting of the merging step are not able to recombine the solutions found at the lower level. Our assumption is that partial solutions are randomly cut out by the crossover operator in the merged species. Therefore, a new mechanism should be added to our model in order to protect the solutions found at the lower level of search.

5.5 On Introducing Modularity in R&D

In order to protect partial solutions from the deleterious effects of the crossover operator, we propose to encapsulate the best solutions found in the lower levels of the recursive *R&D* strategy into modules. In addition, modules provides a way to decompose the problem exploiting symmetries, and regularities inherent to the problem in exam.

5.5.1 Creation & Representation of Modules

Modules are created when the recursion stops, i.e., when the division of the search space stops. Two cases are considered:

(i) One or more species contain an optimal solution for the search space in which they evolve. These optimal solutions are transformed in modules.

(ii) No optimal solutions have been found, then the individual with the best fitness value amongst the species is selected to become a module.

In our context a module embeds a complete tree in a simple node. The leaves of the tree become parameters of the module. The creation of a new module is shown in Figure 5.11.

In addition we also generate new terminal nodes from the modules definition. Typically, these new terminals are modules which have their parameters setted up as in the original tree used to generate the module.

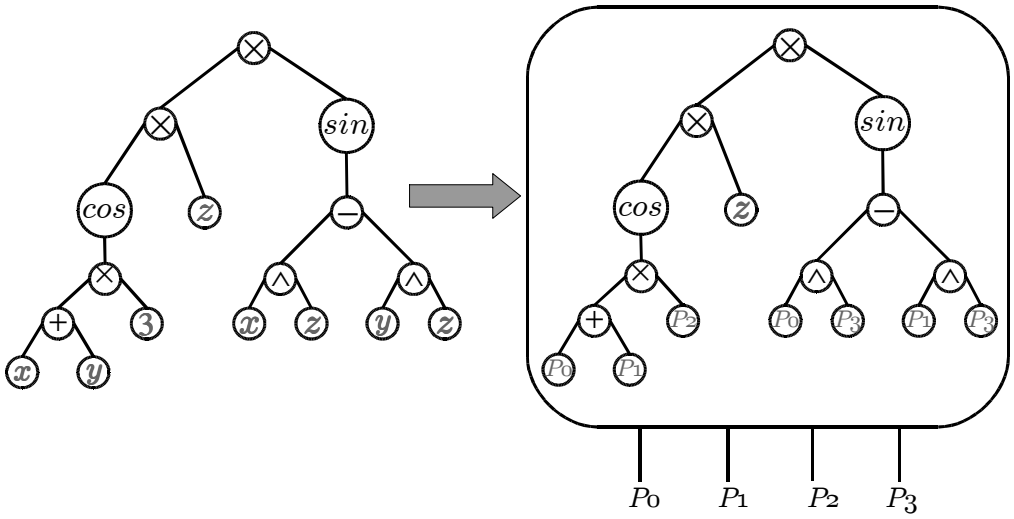


Figure 5.11 - Creation of a new module from a tree.

5.5.2 R&D Model's Modifications

In order to use modules in our strategy, we introduce several modifications in the previous model. In the following we consider that the recursion stopped at level l. Modifications are as follows:

(i) We generate a new species based on an extension of the relations and variables set of the reference species at level l-1 with modules and terminals created from the solutions found in the species evolved at level l.

(ii) The new species become the new "reference species" and evolves until to reach the same number of generations than the previous "reference species".

(iii) When the "reference species" stops then all species at level l, the previous and the current "reference species" at level l-1 are merged together.

(iv) From this point the strategy proceeds as usual.

The procedure described above is replicated for each level of division. As a consequence modules defined at a level may include modules generated in the lower level.

5.5.3 Results and Discussion

5.5.3.1 Boolean Domain

Even-k-parity problem

	Percentage of success		**Elapsed time (min.)**	
	4 bits	5 bits	4 bits	5 bits
Classical GP	60	9	253	591
E&F $f_{RD}(R_S \cup V_S)$	**100**	5	175	**274**
R&D $f_{RD}(R_S \cup V_S)$ (version 3)	95	25	187	744
R&D $f_{RD}(R_S \cup V_S)$	**100**	**100**	**101**	355

Table 5.10 - Percentage of success and elapsed time for the even-k-parity problems with k=4 and k=5.

In this section we report only the results for the even-parity problem. The results for the other problems are reported in Appendix C.
With the introduction of modules in our model we obtain better results in terms of percentage of success. But this improvement is achieved for a higher computational cost (except for the even-parity problem). Several factors may explain this increase in computational time. First, the trees which contain modules require more time to be evaluated since each module node contain another tree which must be evaluated. Moreover, for a same upper bound depth, trees with modules are more complex than trees without modules. Indeed, if we replace each module node by the tree they embed, then the trees are much deeper.

5.5.3.2 Numerical Functions

We retrieve also this growth in computational cost for the numerical problems. Indeed, for all observed functions we got a better accuracy but at the price of an increase in computational time. The results for the other functions are reported in Appendix C.
It is worth to note that our strategy found the exact polynomial solution 15 times for the quintic function. It proves that our algorithm is able to disambiguate the search space in such a way that only useful elements from the relations and variables set are

kept for a further evolution.

Quintic function

	Best adjusted fitness values		**Elapsed time (min.)**
	Average	Standard deviation	
Classical GP	0.9938	0.0177	333
E&F $f_{RD}(R_S \cup V_S)$	0.9993	0.0006	180
R&D $f_{RD}(R_S \cup V_S)$ (version 1)	**0.9993**	0.0007	149
R&D $f_{RD}(R_S \cup V_S)$	**0.9996**	0.0008	3868

Table 5.11 - Results for the quintic function.

5.5.3.3 Santa Fe Ant

For the Santa Fe ant, our strategy follows the same trend as for the boolean problems.

	Percentage of success	**Elapsed time (min.)**
Classical GP	9	861
E&F $f_{RD}(R_S)$	41	524
R&D $f_{RD}(R_S)$ (version 1)	35	**427**
R&D $f_{RD}(R_S)$	**60**	2170

Table 5.12 - Percentage of success and elapsed time for the Santa Fe ant problem.

5.6 Summary

In this chapter we introduced new search strategies with the intent to improve efficiency and accuracy of the classical algorithm. The results confirm the ability of our algorithm to greatly increase either the probability of success or the accuracy of the solutions found for the problems used. However if modules are used, our strategy requires more time to obtain these improvements.

But beyond the respective performances of each strategy presented in this chapter, our study permits to identify the key mechanisms which can make enhancements of the original algorithm successful:

(i) The Reduction & Differentiation function associated with a careful management of the allocated computational resources carries out to maximise the coverage of the search space. Although the *E&F* strategy (similar to the pyramid and beam search strategies) also extends the search space, the individuals in each species are distributed on the overall search space. On the contrary the *R&D* strategy divides the search space in small regions which are explored by different species. By doing so, each species evolves on a *reduced* and *distinct* search space from its neighbors. The results presented in section 5.2.4 show that the exploration realised in this way is more efficient than the other strategies.

(ii) Then our disambiguation procedure coupled with a recursive application of the Reduction & Differentiation function is able to decompose the original problem in sub-problems which may be solved more easily.

(iii) Finally partial solutions found in different regions of the search space are encapsulated into modules in order to protect them from the deleterious effects of the recombination operator.

All these steps enable our strategy to provide a tradeoff between exploration and exploitation which is dynamically adapted to the problem in exam.

In addition, our study gives some insights for understanding why other techniques based on modules can not work for particular problems. Actually, in the ADFs, modules are evolved from the beginning of the evolution. Therefore the separation between exploration and exploitation is not clearly defined. Moreover ADFs require that the user defines the number of functions and specify the signature of each one beforehand. In the same way, other approaches based on modules require an important knowledge on the problem in exam. On the contrary, our strategy is able to discover automatically the correct form of the modules without any intervention from the user.

These results suggest that our model could be a general choice for genetic programming, in particular, whenever the domain is not well understood and involves a wide number of functions and terminals.

Chapter 6

Hyper-Volume Error Separation Strategy

Symbolic regression is aimed at discovering mathematical expressions, in symbolic form, that fit a given sample of data points. It is usually assumed that the data points in the sample are related to a unique function. There are many applications, however, in which this assumption may constitute an oversimplification, such as signal processing, time series prediction, pattern recognition and so on. In such cases the data-set could span across portions of the input space that are to be modeled differently, which means that the symbolic regression should produce a discontinuous function.

Attacking this problem involves facing several challenging issues:

(i) Localizing discontinuity boundaries in the data sample without preliminary knowledge about their number and location.

(ii) Partitioning the data-set according to such boundaries, in order to improve the fit on each partition.

(iii) Assembling the formulas found in each partition into a consistent hierarchy in order to provide a single discontinuous function.

Obviously, performing the above steps in a multidimensional space adds substantial further complexity.

6.1 Related Work

Symbolic regression problems are usually faced with a functions set including basic arithmetic operators (e.g., +, −, *, /) and other elementary functions (e.g., exp, log, cos, sin). As these basic elements are used by the GP process to evolve more elaborate programs – i.e., formulas – most of any resulting formula will be continuous

and smooth. To improve accuracy when the underlying model is discontinuous, one can introduce in the functions set conditional operators and relational operators (*if*, $\leq$, $\geq$, $=$). Another approach is to change the representation of the solution. In [83], the authors propose to introduce a new hybrid structure to deal with discontinuity for univariate functions. This structure called point-tree representation contains the discontinuity fields and the associated sub-functions in the same individual. Authors report significant improvements of success rate and better solutions than the classical GP approach. This new representation implies a modification of the genetic operators and preliminary knowledge on the number of discontinuity points. Most importantly, it can be applied only to univariate data-sets.

In this section we propose a novel approach, suitable for multivariate data-sets and that does not require any prior knowledge about the number of discontinuities. We are not aware of any other proposal with similar features.
We generate an initial population from scratch and let this population evolve for a small number of generations. We select the best individual and evaluate the error for each fitness case. This error is used by an algorithm developed by us, that we call *Hyper-Volume Error Separation* (HVES) and implements an heuristic for identifying the portions of the input space requiring different approximating functions. Next we reflect such partition of the input space on the data-set and run several preliminary evolutions, one for each partition. The populations resulting from such independent evolutions are finally merged and evolved again.

We compared our approach to the three approaches mentioned above on 8 distinct benchmarks, including all those considered in [83]. The results show that our approach is very effective and largely outperforms the existing alternatives, because it provides significant improvements of the accuracy of the solutions with either the same or lower computational resources.

The following sections are organized as follows. In Section 6.2 we give an overview of the Genetic Programming strategy used to discover discontinuous functions. Then, we describe the underlying mechanisms of the Hyper-Volume Error Separation algorithm. Section 6.4 describes the experimental procedure used to benchmark our approach. Section 6.5 discusses the results and the behavior of the new algorithm.

6.2 Coupling GP with HVES: an Overview

In this section we describe step by step the working principles of our approach. In Section 6.3 we will enter more in detail in the HVES algorithm.
Clearly, if the search finds an individual that solves the problem for a given data sample, then the search stops immediately. For ease of description, we omit this action from the description below.

6.2.1 Model Description

We generate an initial population P_I from scratch and let this population evolve for a predefined number of generations. We select the best individual and evaluate the error for each fitness case. A fitness case is an input(s)/output(s) pair, which allows measuring how well an evolved individual estimates the output(s) from the input(s). The resulting errors and the entire dataset D are given as parameters to our HVES algorithm. This algorithm partitions D in two subsets D_H and D_R according to an heuristic described later. Then we generate two further populations from scratch, say P_H and P_R, and let them evolve for a small and predefined number of generations on only part of the dataset: P_H is given only D_H whereas P_R is given only P_R (Figure 6.1).

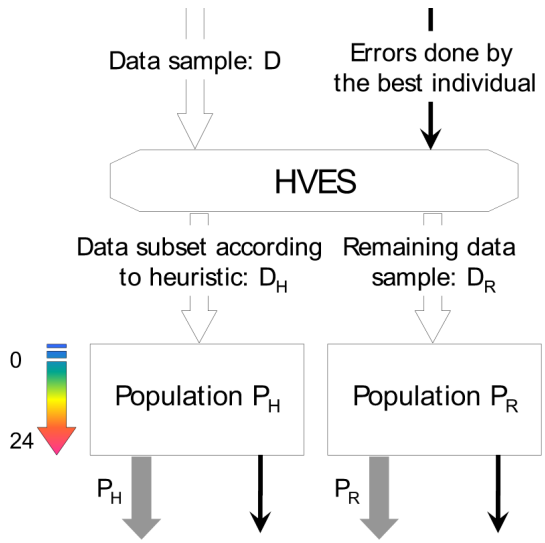


Figure 6.1 - GP with HVES in the division phase (thick gray arrows represent populations, thick empty arrows represent data-sets).

Finally, we merge the evolved P_I, P_H, P_R and let the resulting final population P_F evolve for a predefined number of generations on the entire dataset D (Figure 6.2). We discovered in our early experiments that this merging step is very helpful. Each evolution phase consists of the same number of generations and involves a population of the same size.

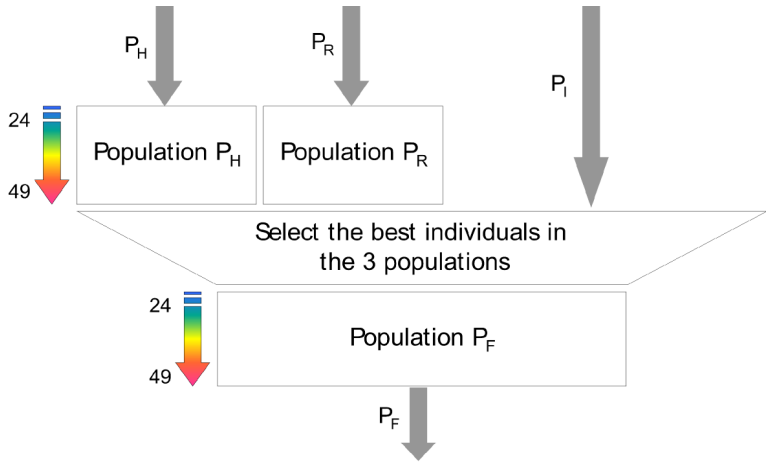


Figure 6.2 - GP with HVES in the merging phase.

Our HVES algorithm works as follows. It partitions the input space in several hyper-volumes whose boundaries are determined by discontinuities in the error function (i.e., the function associating the error of the best individual with each fitness case). Then, it selects the "most difficult" hyper-volume (see below) and partitions the dataset D in two regions: one D_H including all the fitness cases within this hyper-volume, and one D_R including all the remaining fitness cases. Recall that after HVES we focus the evolution of a population on D_H and of *another* population on D_R. The choice of the "most difficult" hyper-volume is made through an index describing a trade-off between number of fitness cases and resulting error – either few points with a large error, or many points with a small error. The rationale is that such hyper-volume should not contain any discontinuity.

The algorithm is also applied *recursively* between HVES and the merging phase, as follows (recursion is not shown in the figures, for clarity). The pair $\langle P_H, D_H \rangle$ produced by HVES plays the role of $\langle P_I, D \rangle$. The population P_H actually used for the merging phase is the one produced by this recursion. Recursion stops when one of the following conditions is satisfied.

- A maximum decomposition depth defined by the user is reached.
- The HVES algorithm does not find any discontinuity boundaries.

The same applies to $\langle P_R, D_R \rangle$. Recursion turns out to be helpful for finding all discontinuity boundaries and for improving the accuracy on difficult regions of the data sample.

6.3 HVES

In this section we provide the details of our HVES algorithm, which has already been illustrated above. Recall that HVES attempts to identify discontinuities by analysing the error rather than the dataset. The reason why this is possible is as follows. The problem consists in finding a function $G(\vec{x})$ that approximates an unknown function $F(\vec{x})$ minimizing the error function $\varepsilon(\vec{x})$.

$$F(\vec{x}) = G(\vec{x}) + \varepsilon(\vec{x}) \tag{6.1}$$

where $\vec{x} = (x_1, x_2, \ldots, x_n)$ denotes a point in $\Re^n$ where $\Re$ is the set of real numbers.

Note that if $F(\vec{x})$ is a discontinuous function and $G(\vec{x})$ is a continuous function then discontinuities in $F(\vec{x})$ are reflected in $\varepsilon(\vec{x})$. We rely on this property to infer the discontinuity boundaries in the data sample from the error vector of the best individual.

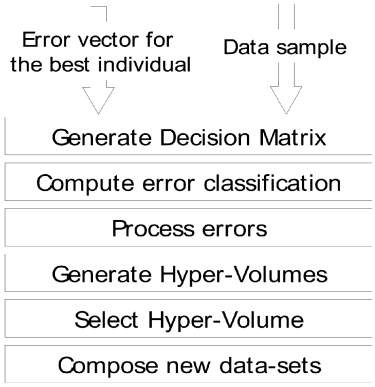


Figure 6.3 - HVES steps.

Rather than describing hyper-volumes explicitly, our HVES algorithm groups fitness cases depending on the hyper-volume they belong to. We represent each such group by means of a tree in which each node corresponds to a fitness case. What makes the algorithm rather complex is that all its steps – e.g., measuring errors, finding discontinuities, grouping points – is done in a multidimensional space. The algorithm

can be decomposed in a succession of six steps as illustrated in Figure 6.3. We will describe these steps one by one in the following subsections.

6.3.1 Decision Matrix

We define a decision matrix DM as a convenient support for further processing. The decision matrix contains a row for each fitness case, as follows. The first n columns describes the n input variables; column $n+1$ contains the desired output; column $n+2$ contains the errors produced by the best individual. The last column contains the error category according to a classification following the rules defined in the next subsection. m denotes the number of fitness cases.

$$DM=\begin{pmatrix} x_{11} & \cdots & x_{1n} & f_1 & e_1 & c_1 \\ x_{21} & \cdots & x_{2n} & f_2 & e_2 & c_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{m1} & \cdots & x_{mn} & f_m & e_m & c_m \end{pmatrix} \tag{6.2}$$

We also generate a map M for accessing DM conveniently, as follows. For each dimension of the input space $\Re^n$, say i, we determine the array L_i containing all distinct values for the i-th input variable, across all fitness cases. Then we sort L_i in ascending order. The map M is composed by the n resulting arrays. We will move on an axis in a discrete way by incrementing or decrementing an index to access to each cell of the array.

6.3.2 Error Classification

We perform a coarse grain classification of the errors in three categories, as follows. We denote the error on fitness case i as e_i. First, we compute the error standard deviation (or root mean square error) across all fitness cases:

$$\sigma e=\sqrt{\frac{1}{m}\sum_{i=1}^{m}(e_i)^2} \tag{6.3}$$

where $e_i=f_i-g_i$, g_i is the output computed by the candidate solution, and m is the number of fitness cases. This index captures the dispersion of the errors. Then, we compute the error kurtosis:

$$ke = \sqrt{\frac{1}{m(\sigma e)^4} \sum_{i=1}^{m} (e_i)^4} \tag{6.4}$$

Kurtosis is the degree of peakedness of a distribution, defined as a normalized form of the fourth central moment of a distribution. This index gives insights on the shape of the error distribution.
Using these indexes we classify errors according to the rules in Table 6.1.

Error category	**Rule**
No error (denoted *NoError* in DM)	When error $e_i = 0$
Low error level (denoted *LowError* in DM)	When $\lvert e_i \rvert < \frac{\sigma e}{1+ke}$
High error level (denoted *HighError* in DM)	When $\lvert e_i \rvert \geq \frac{\sigma e}{1+ke}$

Table 6.1 - Rules for error classification.

6.3.3 Error Processing

In order to detect discontinuities in the error, we use a simple heuristic based on angles. Each fitness case is considered as a point of a space with $n+1$ dimensions, where the last dimension is the error associated with that point. For deciding whether there is a discontinuity at point v, we determine the two points closest to v, say u and w, and construct two vectors $\vec{uv}$, $\vec{vw}$. Then we evaluate the angles between these vectors and each axis of the input space $\Re^n$. If, for each axis, the absolute value of the difference between these angles is greater than a certain axis-specific threshold, then we assume there is a discontinuity boundary separating $\vec{uv}$ and $\vec{vw}$. An example is given in Figure 6.4 for a one-dimensional space and for a threshold $\Phi = \frac{\pi}{2}$.

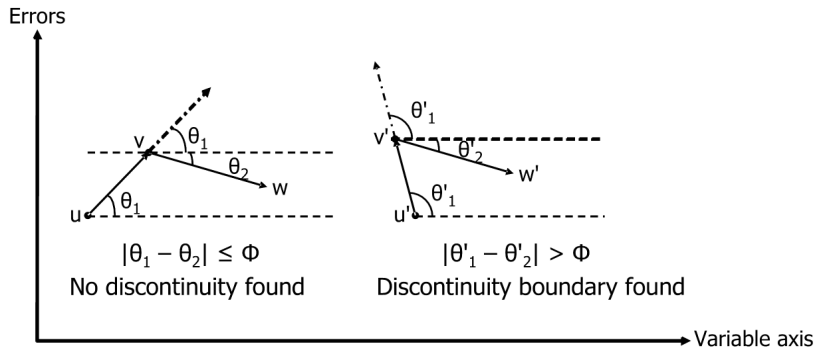


Figure 6.4 - Discontinuity detection.

The evaluation of the axis-specific threshold Φ_i for each axis i is as follows. For each fitness case v, we determine the closest neighbors for each axis, as follows. Let $M(v)_i$ denote the index in map M (Section 6.3.1) of v along axis i. For each axis i we collect all fitness cases whose index in M is $M(v)_i$+1. Then we keep the closest fitness case in terms of Euclidean distance.

Next, we determine the point of $\Re^n$, say v_c, whose coordinates are those of the closest points previously found, e.g., the i-th element of v_c is the i-th element of the fitness case closest to v along axis i. We calculate the gradient $\nabla \varepsilon$ of the ε function with respect to vector $v\vec{v}_c$:

$$\nabla \varepsilon = \left(\frac{\partial \varepsilon}{\partial x_1}, \frac{\partial \varepsilon}{\partial x_2}, \ldots, \frac{\partial \varepsilon}{\partial x_n} \right) \tag{6.5}$$

We convert each component of the gradient to an angle by means of the tan^{-1} function (error gradients may also interpreted like the slopes of the error for each axis), thereby obtaining the following tuple:

$$\Theta = (\theta_1, \theta_2, \ldots, \theta_n) \tag{6.6}$$

Finally, for each axis i, we calculate the standard deviation of the angles across all the m fitness cases and determine the angle thresholds Φ_i accordingly:

$$\Phi = (\sigma(\theta_1), \sigma(\theta_1), \ldots, \sigma(\theta_n)) \tag{6.7}$$

6.3.4 Hyper-Volume Generation

As observed at the beginning of this section, we group fitness cases based on the hyper-volume they belong to, without describing such hyper-volumes explicitly. In order to describe such grouping in a convenient way we developed a tree structure (Figure 6.5) in which each node corresponds to a fitness case and stores: (i) the corresponding *DM* row; and (ii) the corresponding indexes in map *M*.

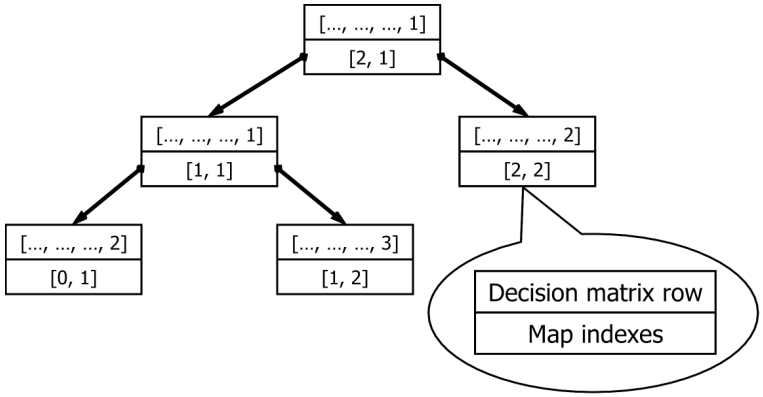


Figure 6.5 - Hyper-Volume tree representation

The algorithm for building such trees is given in Figure 6.6. The entry point is *buildHVTreeSet()*. Essentially, a child node N_c is inserted according to a system of rules using information contained in the root node N_r, the parent node N_p and in some cases the grand parent node N_{gp} (at the beginning root node and parent node are the same). A new tree is created when there is no more fitness case which satisfies any of the the rules embedded in procedure *insertNode()* in Figure 6.6.

Global variables

DM	Decision matrix
M	Generated map
setNodesAnyTree	Set of the nodes belonging to one of the trees
setNodesThisTree	Set of the nodes belonging to the tree under construction

buildHVTreeSet()
1) **Foreach** row in *DM*,
  a) Generate a root node N_r
    i) **if** $N_r \notin$ *setNodesAnyTree* **then**
      A) Clear *setNodesThisTree*
        buildHVTree(N_r)

buildHVTree(N_p)
N_p Parent node

1) **Foreach** $i \in [1,n]$, // all the axis
   a) **While** there is a node to insert in this tree,
      i) N_c := point closest to N_p along current axis and not yet inserted in this tree
      ii) **insertNode**(N_p, N_c)

insertNode(N_p, N_c)
   1) N_r := root node of the tree
   2) C_r := error category for the root node N_r
   3) C_p := error category for the parent node N_p
   4) C_c := error category for the node in exam N_c
   5) **if** C_r=*NoError* $\wedge$ C_p=*NoError* $\wedge$ C_c=*NoError* $\wedge$ $N_c \notin$ *setModesAnyTree* **then**
      Insert N_c as a child of N_p, put in *setNodesAnyTree*
   6) **if** C_r=*LowError* $\wedge$ C_p=*NoError* $\vee$ C_p=*LowError* **then**
      a) **if** C_c=*NoError* $\wedge$ $N_c \notin$ *setNodesThisTree* **then**
         Insert N_c as a child of N_p, put in *setNodesThisTree*
      b) **elseif** C_c=*LowError* $\wedge$ $N_c \notin$ *setNodesAnyTree* **then**
         Insert N_c as a child of N_p, put in *setNodesAnyTree*
   7) **if** C_r=*HighError* $\wedge$ C_p=*LowError* $\vee$ C_p=*HighError* **then**
      a) **if** C_c=*NoError* $\wedge$ $N_c \notin$ *setNodesThisTree* **then**
         Insert N_c as a child of N_p, put in *setNodesThisTree*
      b) **else**
         i) N_{gp} := the parent of the parent node N_p
         ii) α := angle vector computed from the nodes N_{gp} and N_p
         iii) β := angle vector computed from the nodes N_p and N_c
         iv) **if** C_c=*LowError* $\wedge$ $N_c \notin$ *setNodesThisTree* **then**
            A) **if** $|\alpha_i - \beta_i| < \Phi_i \ \ \forall i \in [1,n]$ **then**
               Insert N_c as a child of N_p, put in *setNodesThisTree*
         v) **elseif** C_c=*HighError* $\wedge$ $N_c \notin$ *setNodesThisTree* **then**
            A) **if** $|\alpha_i - \beta_i| < \Phi_i \ \ \forall i \in [1,n]$ **then**
               Insert N_c as a child of N_p, put in *setNodesAnyTree*
   8) **if** N_c has been inserted **then**
      buildHVTree(N_c)

Figure 6.6 - The Hyper-Volume building algorithm.

6.3.5 Hyper-Volume Selection

We defined an index capturing the size of an hyper-volume and the amount of errors associated with the fitness cases in it. We called this index Weighted Euclidean Distance (WED):

$$WED = \sum_{k}^{s} \sum_{l}^{c} \left(\delta(node_k, childNode_l) \frac{e_k + e_l}{2} \right) \tag{6.8}$$

where: $\delta(node_k, childNode_l)$ denotes the Euclidean distance between the points of the input space associated with nodes $node_k$, $childNode_l$; s is the number of nodes in the Hyper-Volume tree; c is the number of children of node $node_k$.

6.3.6 Data-sets Composition

We define an hyper-parallelepiped enclosing the hyper-volume found at the previous step, as follows. We determine, for each axis, the minimum and maximum values across all fitness cases stored in the tree. Such values define the boundaries of the hyper-parallelepiped. Finally, we partition the data-set D so that fitness cases within the hyper-parallelepiped belong to set D_H whereas the remaining fitness cases belong to D_R. Using an hyper-parallelepiped simplifies the interpretation of the final formulas, since discontinuity boundaries may be delimited using the conditional operator *if* associated with a predicate based on the intersection of the intervals found.

6.4 Experimental Setup

We compared the following approaches:

(i) *Classical GP*;

(ii) *Conditional GP*, i.e., GP with conditional and relational operators in the functions set in order to make capturing of discontinuities easier (as outlined in the Section 6.1);

(iii) *Hybrid Structure GP*: the approach proposed in [83];

(iv) *GP-HVES*, our proposal.

We used a set of eight functions already presented in Chapter 2 as benchmark. Functions DF_1 to DF_6 have one input variable, functions DMF_1, DMF_2 have two input variables. Functions DF_1, DF_2, DF_3, DF_4 are those used in [84]. We defined the other functions so that they indeed include discontinuities. We remark that we defined the univariate functions DF_5, DF_6 so that their discontinuities are not very pronounced. Our method is suitable for this kind of problems, unlike the proposal in [84] that has been explicitly designed for non-smooth functions. We executed all the approaches on all benchmarks, except that we did not apply Hybrid Structure GP to DF_5, DF_6 (because it would have been unfair) nor to DMF_1, DMF_2 (because these functions are multivariate).
The functions and terminals set used in each case are shown in Table 6.2.

	GP, GP-HVES	
Benchmark functions	**Terminal set**	**Function set**
DF_1, DF_2, DF_3, DF_4	x, 1,	+, -, ×, /
DF_5, DF_6	x, 1	+, -, ×, /
DMF_1, DMF_2	x, y, 1	+, -, ×, /, *cos*, *sin*
	Extra terminals and functions for GP conditional	
DF_1, . . ., DF_6 and DMF_1, DMF_2	*true*	If, ≤, ≥, *or*, *and*, *not*

Table 6.2 - Terminals and functions sets.

For DF_1, . . ., DF_4 we used the same fitness as in [84], i.e., the sum of absolute errors:

$$SAE = \sum_{i=1}^{m} |f_i - g_i| \tag{6.9}$$

For all other functions we used as fitness the mean of the squared distances between the expected values f_i and the values g_i obtained by the individual:

$$MSE = \frac{1}{m} \sum_{i=1}^{m} (f_i - g_i)^2 \tag{6.10}$$

We executed 3 tests for each test case. Each test is the result of 100 independent executions. Each execution starts with a different seed for the random number generator but we used the same seeds for each test. We ran all simulations on a PC based on a processor Intel Xeon 3.20 GHz with 2 GB of RAM.
All the parameters are summarized in Table 6.3. Whenever GP-HVES runs an evolution, the maximum number of generations is set to 25.

Parameter	**Setting (DF1 to DF4)**	**Setting (DF5 to DF6 and MDF1 to MDF2)**
Population size	500	1000
Max generation number	600	200
Selection	Tournament of size 7	
Initialization method	Ramped Half-and-Half	
Initialization depths	2 levels	2-4 levels
Maximum depth	10	8
Internal node bias	90% internals, 10% terminals	
Elitism	0	1
Duplication rate	10%	5%
Crossover rate	70%	85%
Mutation rate	20%	15%
Recursion depth (HVES only)	11	3

Table 6.3 - Parameters settings

6.5 Results

To assess the accuracy of the solutions, we considered the fitness of the best individual found in each run. We computed average and standard deviation of this fitness value across all runs. Figure 6.7 shows the results for each approach.

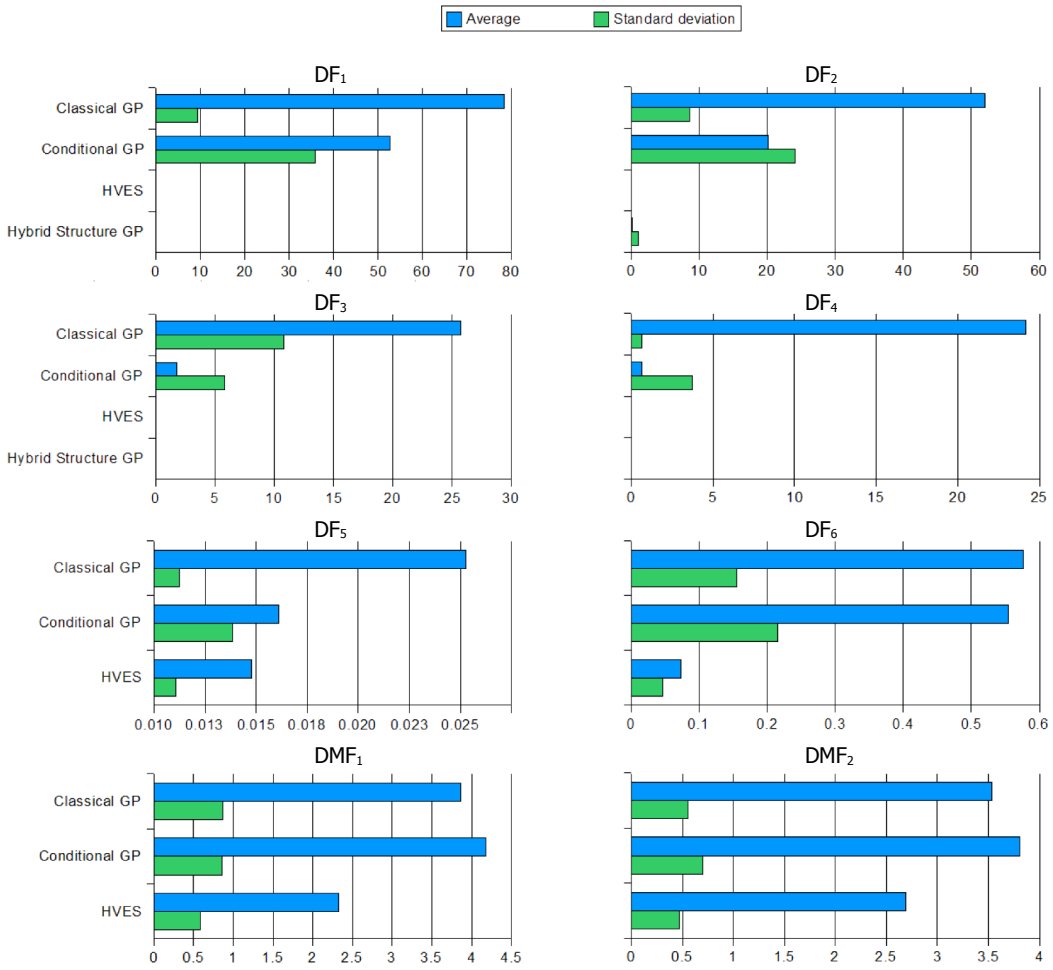


Figure 6.7 - Average and standard deviation of the fitness values for each tested function

Our approach is able to find the exact discontinuous functions with a success rate of 100% for the four first functions. It is worth to note that our proposal does at least as well as the GP based on hybrid structure. For the functions DF_1, DF_2, DF_3, DF_4 we never reach the maximum recurrence depth defined in Table 6.3. The algorithm stops after two or three recursions only since it finds the exact discontinuity boundaries and the correct discontinuous functions.

We note that our proposal exhibits the best accuracy for DF_5, DF_6, the improvement with respect to the best result with the GP algorithm using conditional and relational operators is 109% and 765%, respectively.
Another important result is that our approach is able to improve the accuracy on the

multivariate functions DMF_1 and DMF_2. Improvements for these functions respect to the best result with the classical GP algorithm is 166% and 132% respectively.
It is interesting to note that in the multivariate test cases, the solutions found by the classical GP algorithm are more accurate than those found with a GP approach using the relational and conditional operators.

Concerning computational cost, we decided not to measure the number of evaluations because this index is not very meaningful in this context: in GP-HVES the data-set size is not the same across all evolutions. Consequently, we decided to measure the computational cost by logging the time spent on all runs. This index has also the advantage of capturing the fact that, in practice, different evaluations do not have the same cost, depending for instance on the number of nodes composing an individual or the complexity of each node.

We report in the Table 6.4 the total time spent in hours for each tested function and each approach. It appears clearly that the HVES outclasses the others strategies whatever the function involved.

	Functions							
Total time	DF_1	DF_2	DF_3	DF_4	DF_5	DF_6	DMF_7	DMF_8
Classical GP	0.7	0.6	4.8	1.8	7.1	14.6	36.5	39.4
Conditional GP	6.1	11.9	3.1	0.7	9.5	17.2	31.8	37.2
HVES	**0.1**	**0.3**	**1.8**	**0.2**	**7.1**	**13.0**	**25.0**	**28.4**

Table 6.4 - Total time spent in hours.

6.6 Summary

In our approach we execute a preliminary evolution and use the error exhibited by the best individual in order to infer discontinuity boundaries in the data sample. Then we apply an algorithm designed by us for selecting an Hyper-Volume in the input space whose boundaries approximately follow the discontinuities. This Hyper-Volume partitions the fitness cases in two sets that are used for driving further preliminary evolutions. The best individuals found by these independent evolutions are then merged and evolved again. The process is also applied recursively until either no further discontinuities are found or a predefined recursion depth is reached.

The results confirm the ability of our algorithm to greatly increase the accuracy of the solutions with respect to existing GP-based approaches. In some benchmarks we have also observed a significant improvement in computational cost and we have never observed an increase in computational cost.

Chapter 7

Conclusions and Research Perspectives

This thesis introduced genetic programming as a promising technique for coping with problems in which finding a solution and its representation is difficult but evaluating the performance of a candidate solution is reasonably simple. We illustrated the GP approach with three real-world applications related to different fields in science and engineering:

- The first application was dedicated to the automatic synthesis of network delay predictors for the protocol TCP. We apply a multi-objective genetic programming approach for constructing a round-trip time (RTT) predictor, i.e., the formula that predicts dynamically the delay experienced by packets along a network connection. The solutions that we found outperform the RTT predictor currently used by all TCP implementations. This result could lead to several applications of genetic programming in the networking field.
- In the second application we used GP for detecting web site defacements. Web site defacement, is the introduction of unauthorized modifications to a web site. Detecting such events automatically is very difficult because web pages are highly dynamic and their degree of dynamism may vary widely across different pages. What makes GP particularly attractive in this context is that it does not rely on any domain-specific knowledge, whose description and synthesis is invariably a hard job. We compared the results to those of a solution we developed earlier, whose design embedded a substantial amount of domain specific knowledge, and the results clearly show that GP may be an effective approach for this job.
- Finally we applied GP on the stiffness problem. The stiffness problem consists in the approximation of the stiffness matrix values as accurately as possible. In this thesis we used Genetic Programming to symbolically regress each element on the diagonal of the stiffness matrix. We compare the real stiffness values

with those estimated with the formulas found by GP. Our results show genetic programming is an effective approach for this task and could be part of the toolbox of many engineers.

In a second part we addressed current issues in scaling genetic programming by creating two new strategies. Purposefully we do not try to improve each step of the evolutionary process separately, a large work have been already done in the past in this direction and may be used as well inside the proposed strategies. On the contrary we encapsulate the GP process for improving efficiency and accuracy of the standard approach.
Our first strategy called Reduction & Differentiation strategy offers a new way to manage the evolutionary process through four keys ideas:

(i) We use the concept of species built on subsets of functions and variables in order to maximize the coverage of the search space. In this way, each species works on a distinct region of the original search space.

(ii) We disambiguate the search space by splitting the original data-set according to the ambiguities found after the removal of a variable.

(iii) We decompose the evolutionary process in N distinct levels of research.

(iv) We preserve the partial solutions found in the lower level by transforming these solutions in modules. With this approach, the higher level can easily recombine the best partial solutions found by the lower level.

The results obtained show that our strategy can improve either the percentage of success to get an acceptable solution or the accuracy of the solutions when compared with the standard algorithm. This comparison covers three problem domains and corresponds to nearly 1 billion individual evaluations, or several months of calculations.
This strategy could be improved in several ways. May be the most obvious is to compile the modules found in order to reduce the computational cost. Another approach could be to restrict the maximum depth for the individuals built with modules.

The second strategy called Hyper-Volume Error Separation aims to perform symbolic regression of multivariate data-sets where the underlying phenomenon is best characterized by a discontinuous function. We are not aware of any other GP-based approach with similar features, in particular, regarding its ability of handling multivariate data-sets.

In the future, we plan to investigate other heuristics for discontinuity detection in noisy data-sets. We are currently studying the work done in computer vision, signal processing and statistics fields in [53][54][10][73]. The challenge will be to adapt these techniques for multidimensional spaces since these approaches usually work in one or two dimensions only. We also envisage to apply our approach for time series prediction and classification tasks. However for these applications we should develop an appropriate policy to use cross validation inside our algorithm in order to avoid over fitting problems.

Bibliography

[1] J. Aikat and J. Kaur and F. D. Smith and K. Jeffay, "Variability in TCP round-trip times", IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement, 2003, pp. 279-284.

[2] E. Alba and M. Tomassini, "Parallelism and Evolutionary Algorithms", IEEE Transactions on Evolutionary Computation, volume 6, issue 5, 2002, pp. 443-462.

[3] M. Allman and V. Paxson, "On estimating end-to-end network path properties", ACM SIGCOMM Comput. Commun. Rev., 1999, pp. 263-274.

[4] P. J. Angeline and J. Pollack, "Evolutionary Module Acquisition", Proceedings of the Second Annual Conference on Evolutionary Programming, 1993, pp. 154-163.

[5] T. Back and U. Hammel and H.-P. Schwefel, "Evolutionary computation: comments on the history and current state", IEEE Transactions on Evolutionary Computation, volume 1, issue 1, 1997, pp. 3-17.

[6] W. Banzhaf and P. Nordin and R. E. Keller and F. D. Francone, "Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications", dpunkt Verlag fur digitale Technologie GbmH and Morgan Kaufmann Publishers, Inc., 1998.

[7] A. Bartoli and E Medvet, "Automatic Integrity Checks for Remote Web Resources", IEEE Internet Computing, volume 10, issue 6, 2006, pp. 56-62.

[8] A. Bartoli and E. Medvet, "A Framework for Large-Scale Detection of Web Site Defacements", In submission, 2007.

[9] P. A. N. Bosman and D. Thierens, "The balance between proximity and diversity in multiobjective evolutionary algorithms", IEEE Transactions on Evolutionary Computation, volume 7, 2003, pp. 174-188.

[10] A. W. Bowman and A. Pope and B. Ismail, "Detecting discontinuities in nonparametric regression curves and surfaces", Statistics and Computing, volume 16, issue 4, 2006, pp. 377-390.

[11] K. Chellapilla, "Evolving Computer Programs without Subtree Crossover", IEEE Transactions on Evolutionary Computation, volume 1, issue 3, 1997, pp. 209-216.

[12] Y. Chen and A. Abraham and B. Yang, "Hybrid flexible neural-tree-based intrusion detection systems", International Journal of Intelligent Systems, volume 22, issue 4, 2007, pp. 337-352.

[13] V. Ciesielski and X. Li, "Pyramid search: Finding solutions for deceptive problems quickly in genetic programming", Proceedings of the 2003 Congress on Evolutionary Computation CEC2003, 2003, pp. 936-943.

[14] C. A. Coello Coello, "An Updated Survey of GA-Based Multiobjective Optimization Techniques", ACM Computing Surveys, volume 32, issue 2, 2000, pp. 109-143.
[15] A. Colorni and M. Dorigo and V. Maniezzo, "Distributed Optimization by Ant Colonies", First European Conference on Artificial Life, 1992, pp. 134-142.
[16] J. M. Daida, "Towards identifying populations that increase the likelihood of success in genetic programming", Proceedings of the 2005 conference on Genetic and evolutionary computation (GECCO 2005), 2005, pp. 1627-1634.
[17] J. M. Daida and M. E. Samples and M. J. Byom, "Probing for limits to building block mixing with a tunably-difficult problem for genetic programming", Proceedings of the 2005 conference on Genetic and evolutionary computation (GECCO 2005), 2005, pp. 1713-1720.
[18] C. Darwin, "On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life", John Murray, London, 1859.
[19] A. Dessi and A. Giani and A. Starita, "An Analysis of Automatic Subroutine Discovery in Genetic Programming", Proceedings of the Genetic and Evolutionary Computation Conference, 1999, pp. 996-1001.
[20] S. Dignum and R. Poli, "Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat", GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, 2007, pp. 1588-1595.
[21] R.C. Dimitriu and H.K.D.H. Bhadeshia and C. Fillon and C. Poloni, "Understanding the Hot–Strength of Ferritic Steels using Neural Networks and Genetic Programming", International Conference on Neural Network and Genetic Algorithm in Materials Science and Engineering (NGMS 2008), 2008.
[22] A. Ekart and S. Z. Nemeth, "Selection Based on the Pareto Nondomination Criterion for Controlling Code Growth in Genetic Programming", Genetic Programming and Evolvable Machines, volume 2, issue 1, 2001, pp. 61-73.
[23] C. Fillon and A. Bartoli, "A Divide and Conquer strategy for improving efficiency and probability of success in Genetic Programming", Proceedings of the 9th European Conference on Genetic Programming, 2006, pp. 13-23.
[24] C. Fillon and A. Bartoli, "Multi-objective Genetic Programming for Improving the Performance of TCP", Proceedings of the 10th European Conference on Genetic Programming, 2007, pp. 170-180.
[25] C. Fillon and A. Bartoli, "Symbolic Regression of Discontinuous and Multivariate Functions by Hyper-Volume Error Separation (HVES)", 2007 IEEE Congress on Evolutionary Computation, 2007, pp. 23-30.
[26] C. Fillon and C. Poloni and A. Bartoli, "A Genetic Programming approach for Stiffness Estimation", Artificial Intelligence for Industrial Applications, AI4IA

2007, 2007, pp. 55-61.
[27] D. B. Fogel, "Evolutionary Computation: Toward a New Philosophy of Machine Intelligence", , volume , issue , 1995.
[28] L. J. Fogel and A. J. Owens and M. J.Walsh, "Artificial Intelligence through Simulated Evolution", , John Wiley & Sons, New York, 1966.
[29] G. Folino and C. Pizzuti and G. Spezzano, "A Scalable Cellular Implementation of Parallel Genetic Programming", IEEE Transactions on Evolutionary Computation, volume 7, issue 1, 2003, pp. 37-53.
[30] W. Fone and P. Gregory, "Web Page Defacement Countermeasures", Proceedings of the 3rd International Symposium on Communication Systems Networks and Digital Signal Processing, 2002, pp. 26-29.
[31] C. Gagné and M. Parizeau, "Open BEAGLE: A New C++ Evolutionary Computation Framework", GECCO, 2002, pp. 888.
[32] C. Gathercole and P. Ross, "Small Populations over Many Generations can beat Large Populations over Few Generations in Genetic Programming", Genetic Programming 1997: Proceedings of the Second Annual Conference, 1997, pp. 111-118.
[33] J. J. Grefenstette and J. E. Baker, "How genetic algorithms work: a critical look at implicit parallelism", Proceedings of the Third International Conference on Genetic Algorithms, 1989, pp. 20–27.
[34] S. Handley, "On the use of a directed acyclic graph to represent a population of computer programs", Proceedings of the 1994 IEEE World Congress on Computational Intelligence, 1994, pp. 154-159.
[35] K. Harries and P. Smith, "Exploring Alternative Operators and Search Strategies in Genetic Programming", Genetic Programming 1997: Proceedings of the Second Annual Conference, 1997, pp. 147-155.
[36] J. M. Holland, "Adaptation in Natural and Artificial Systems", University of Michigan Press, Ann Arbor, MI, 1975.
[37] M. Hutter and S. Legg, "Fitness Uniform Optimization", IEEE Transactions on Evolutionary Computation, volume 10, issue 5, 2006, pp. 568-589.
[38] D. Jackson and A. P. Gibbons, "Layered Learning in Boolean GP Problems", Proceedings of the 10th European Conference on Genetic Programming, 2007, pp. 148-159.
[39] V. Jacobson, "Congestion avoidance and control", Proceedings ACM SIGCOMM, 1988, pp. 314-329.
[40] S. Kamio and H. Iba, "Adaptation technique for integrating genetic programming and reinforcement learning for real robots", IEEE Transactions on Evolutionary Computation, volume 9, issue 3, 2005, pp. 318-333.
[41] M. Keith and M. C. Martin, "Genetic Programming in C++: Implementation issues", Advances in Genetic Programming, MIT Press, 1994.

[42] J. Kennedy and R. C. Eberhart, "Particle Swarm Optimization", Int'l. Conference on Neural Networks, 1995, pp. 1942-1948.
[43] E. K. Kinnear, "Alternatives in Automatic Function Definition: A Comparison Of Performance", 1994
[44] S. Kirkpatrick and C. Gelatt and M. Vecchi, "Optimisation by simulated annealing", Science, volume 220, issue 4598, 1983, pp. 671-680.
[45] T. Kolodziejczyk and R. Toscano and C. Fillon and S. Fouvry, C. Poloni, "Ball bearing damage prediction under fretting wear conditions", In submisson, 2008.
[46] John R. Koza, "Genetic Programming: On the Programming of Computers by Means of Natural Selection", MIT Press, 1992.
[47] John R. Koza, "Genetic Programming II: Automatic Discovery of Reusable Programs", MIT Press, 1994.
[48] J. R. Koza and M. A. Keane and M. J. Streeter, "What's AI Done for Me Lately? Genetic Programming's Human-Competitive Results", IEEE Intelligent Systems, volume 18, issue 3, 2003, pp. 25-31.
[49] W. B. Langdon, "Size Fair and Homologous Tree Genetic Programming Crossovers", Genetic Programming and Evolvable Machines, volume 1, issue 1/2, 2000, pp. 95-119.
[50] W. B. Langdon and R. Poli, "Why Ants are Hard", Genetic Programming 1998: Proceedings of the Third Annual Conference, 1998, pp. 193-201.
[51] W. B. Langdon and R. Poli, "Why Building Blocks Don't Work on Parity Problems", 1998
[52] W. B. Langdon and T. Soule, R. Poli and J. A. Foster, "The Evolution of Size and Shape", 1999
[53] D. Lee, "Coping with discontinuities in computer vision: their detection, classification, and measurement", IEEE Transactions on Pattern Analysis and Machine Intelligence, volume 12, issue 4, 1990, pp. 321-344.
[54] D. Lee and G.W. Wasilkowski, "Discontinuity detection and thresholding-a stochastic approach", Proceedings of the 1991 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 1991, pp. 208-214.
[55] T. Loveard, "Genetic Programming With Meta-Search: Searching For a Successful Population Within The Classification Domain", Genetic Programming, Proceedings of EuroGP'2003, 2003, pp. 119-129.
[56] S. Luke, "Two Fast Tree-Creation Algorithms for Genetic Programming", IEEE Transactions on Evolutionary Computation, volume 4, issue 3, 2000, pp. 274-283.
[57] S. Luke, "When Short Runs Beat Long Runs", Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), 2001, pp. 74-80.
[58] S. Luke and G. C. Balan and Liviu Panait, "Population Implosion in Genetic Programming.", Genetic and Evolutionary Computation -- GECCO-2003, 2003, pp. 1729-1739.

[59] S. Luke and L. Panait, "A Survey and Comparison of Tree Generation Algorithms", Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), 2001, pp. 81-88.
[60] L. Ma and K. E. Barner and G. R. Arce, "Statistical Analysis of TCP's Retransmission Timeout Algorithm", IEEE/ACM Transactions on Networking, volume 14, issue 2, 2006.
[61] H. Majeed and C. Ryan, "A Less Destructive, Context-aware Crossover Operator for GP", Proceedings of the 9th European Conference on Genetic Programming, 2006, pp. 36-48.
[62] E. Medvet and C. Fillon and A. Bartoli, "Detection of Web Defacements by means of Genetic Programming", Third International Symposium on Information Assurance and Security, IAS 2007, 2007, pp. 227-234.
[63] D. J. Montana, "Strongly Typed Genetic Programming", Evolutionary Computation, volume 3, issue 2, 1995, pp. 199-230.
[64] H. Muhlenbein and G. Paas, "From recombination of genes to the estimation of distributions I. Binary parameters", Lecture Notes in Computer Science: Parallel Problem Solving from Nature, volume IV, issue 1411, 1996, pp. 178-187.
[65] S. Mukkamala and A. H. Sung and A. Abraham, "Modeling intrusion detection systems using linear genetic programming approach", IEA/AIE'2004: Proceedings of the 17th international conference on Innovations in applied artificial intelligence, 2004, pp. 633-642.
[66] P. Nordin and F. Hoffmann and F. D. Francone and M. Brameier and W. Banzhaf, "AIM-GP and Parallelism", Proceedings of the Congress on Evolutionary Computation, 1999, pp. 1059-1066.
[67] M. O'Neill and C. Ryan, "Grammatical Evolution", IEEE Transactions on Evolutionary Computation, volume 5, issue 4, 2001, pp. 349-358.
[68] D. Parrott and X. Li and V. Ciesielski, "Multi-objective Techniques in Genetic Programming for Evolving Classifiers", Proceedings of the 2005 IEEE Congress on Evolutionary Computation, 2005, pp. 1141-1148.
[69] E. Cant´u-Paz and D.E. Goldberg, "Are Multiple Runs of Genetic Algorithms Better than One?", Proceedings of the Genetic and Evolutionary Computation Conference GECCO'03, 2003, pp. 801-812.
[70] R. Poli, "A Simple but Theoretically-motivated Method to Control Bloat in Genetic Programming", Genetic Programming, Proceedings of EuroGP'2003, 2003, pp. 204-217.
[71] R. Poli and W. B. Langdon, "An Experimental Analysis of Schema Creation, Propagation and Disruption in Genetic Programming", Genetic Algorithms: Proceedings of the Seventh International Conference, 1997, pp. 18-25.
[72] R. Poli and W. B. Langdon and S. Dignum, "On the Limiting Distribution of Program Sizes in Tree-based Genetic Programming", Proceedings of the 10th

European Conference on Genetic Programming, 2007, pp. 193-204.
[73] P. Qiu, "Discontinuous regression surfaces fitting", Annals of Statistics, volume 26, issue 6, 1998.
[74] I. Rechenberg, "Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution", 1973.
[75] "RFC 1122, Requirements for Internet Hosts - Communication Layers", 1989.
[76] "RFC 793, Transmission Control Protocol", 1981.
[77] M. E. Roberts, "The Effectiveness of Cost Based Subtree Caching Mechanisms in Typed Genetic Programming for Image Segmentation", Applications of Evolutionary Computing, EvoWorkshops2003: EvoBIO, EvoCOP, EvoIASP, EvoMUSART, EvoROB, EvoSTIM", 2003, pp. 444-454.
[78] J. P. Rosca and D. H. Ballard, "Hierarchical Self-Organization in Genetic Programming", Proceedings of the Eleventh International Conference on Machine Learning, 1994.
[79] J. P. Rosca and D. H. Ballard, "Discovery of Subroutines in Genetic Programming", 1996.
[80] H.P. Schwefel, "Evolutionsstrategie und numerische optimierung.: Toward a New Philosophy of Machine Intelligence", 1975.
[81] H.P. Schwefel and T. Back, "Artificial Evolution: How and Why?", Genetic Algorithms and Evolution Strategies in Engineering and Computer Sciences, 1997, pp. 327-340.
[82] S. Sedaghat and J. Pieprzyk and E. Vossough, "On-the-fly web content integrity check boosts users' confidence", Commun. ACM, volume 45, issue 11, 2002, pp. 33-37.
[83] X. Shengwu and W. Weiwu, "A new hybrid structure genetic programming in symbolic regression", Proceedings of the 2003 Congress on Evolutionary Computation CEC2003, 2003, pp. 1500-1506.
[84] S. Xiong and W. Wang and F. Li, "A new genetic programming approach in symbolic regression", Proceedings 15th IEEE International Conference on Tools with Artificial Intelligence, 2003, pp. 161-165.
[85] D. Song and M. I. Heywood and A. Nur Zincir-Heywood, "Training genetic programming on half a million patterns: an example from anomaly detection", IEEE Transactions on Evolutionary Computation, volume 9, issue 3, 2005, pp. 225-239.
[86] T. Soule and J. A. Foster, "Effects of Code Growth and Parsimony Pressure on Populations in Genetic Programming", IEEE Transactions on Evolutionary Computation, volume 6, issue 4, 1998, pp. 293-309.
[87] P. Stone and M. Veloso, "Layered Learning", Proceedings of the 17th International Conference on Machine Learning, 2000, pp. 369-381.
[88] H. Tillema and O. van Verseveld, "Report on Evolutionary Design Code",

2007.
[89] V. Varadan and H. Leung, "Reconstruction of polynomial systems from noisy time-series measurements using genetic programming", IEEE Transactions on Industrial Electronics, volume 48, issue 4, 2001, pp. 742-748.
[90] D.A. Van Veldhuizen and J.B. Zydallis and G.B. Lamont, "Considerations in Engineering Parallel Multiobjective Evolutionary Algorithms", IEEE Transactions on Evolutionary Computation, volume 7, issue 2, 2003, pp. 144-173.
[91] D. Whitley, "The GENITOR algorithm and selection pressure: why ranked-based allocation of reproductive trials is best", Proceedings of the Third International Conference on Genetic Algorithms, 1989, pp. 239–255.
[92] G. C. Wilson and A. McIntyre and M. I. Heywood, "Resource Review: Three Open Source Systems for Evolving Programs--Lilgp, ECJ and Grammatical Evolution", Genetic Programming and Evolvable Machines, volume 5, issue 1, 2004, pp. 103-105.
[93] J. Woodward, "GA or GP? That is not the question", Proceedings of the 2003 Congress on Evolutionary Computation CEC2003, 2003, pp. 1056-1063.
[94] T. Xia and G. Qu and S. Hariri and M. Yousif, "An efficient network intrusion detection method based on information theory and genetic algorithm", Performance, Computing, and Communications Conference, 2005. IPCCC 2005. 24th IEEE International, 2005, pp. 11-17.
[95] C. Yin and S. Tian and H. Huang and J. He, "Applying Genetic Programming to Evolve Learned Rules for Network Anomaly Detection", Advances in Natural Computation, First International Conference, ICNC 2005, Proceedings, Part III, 2005, pp. 323-331.
[96] B.-T. Zhang and H. Muhlenbein, "Balancing Accuracy and Parsimony in Genetic Programming", IEEE Transactions on Evolutionary Computation, volume 3, issue 1, 1995, pp. 17-38.
[97] E. Zitzler and L. Thiele and M. Laumanns and C. M. Fonseca and V. G. da Fonseca, "Performance Assessment of Multiobjective Optimizers: An Analysis and Review", IEEE Transactions on Evolutionary Computation, volume 7, issue 2, 2003, pp. 117-132.
[98] Zone-H.org, "Statistics on Web Server Attacks for 2005", 2006.

Appendix A

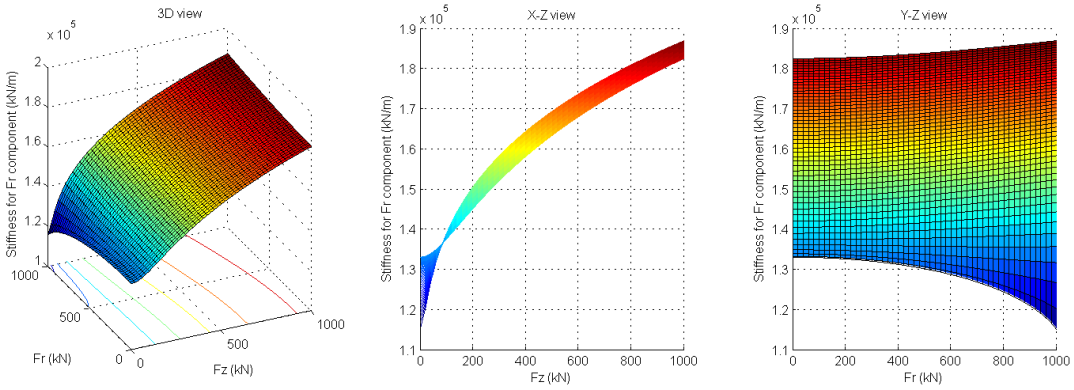


Figure 7.1 - Stiffness for the F_r component.

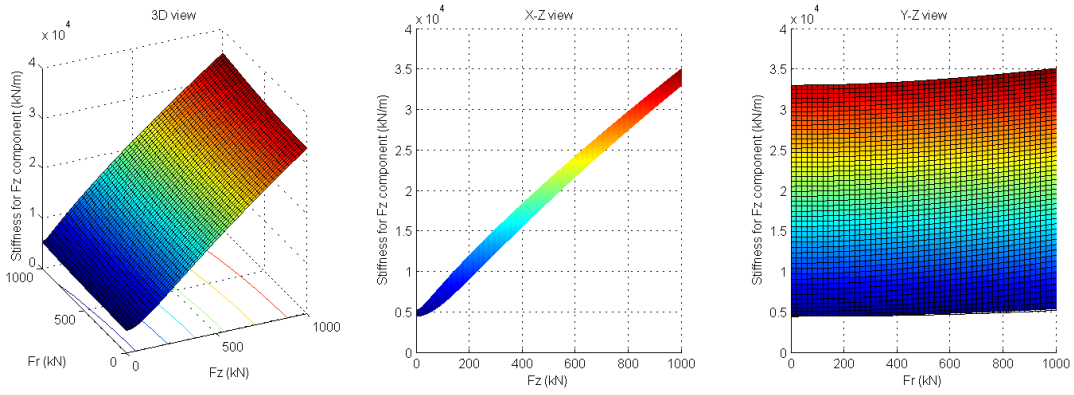


Figure 7.2 - Stiffness for the F_z component.

Appendix B

In this appendix we reported the results obtain for the first version of the *R&D* strategy on the numerical functions.
All of these are described in Chapter 2, Section 2.4.2 and Section 2.4.1.

Function F_1

	Best adjusted fitness values		**Elapsed time (min.)**
	Average	Standard deviation	
Classical GP	0.0814	0.0526	300
E&F $f_{RD}(R_S)$	0.1047	0.470	156
R&D $f_{RD}(R_S)$	0.1066	0.0682	134
E&F $f_{RD}(V_S)$	0.0910	0.0427	89
R&D $f_{RD}(V_S)$	0.0794	0.0327	**71**
E&F $f_{RD}(R_S \cup V_S)$	0.1053	0.0552	230
R&D $f_{RD}(R_S \cup V_S)$	**0.1133**	0.0884	187

Table 7.1 - Results for the function F_1.

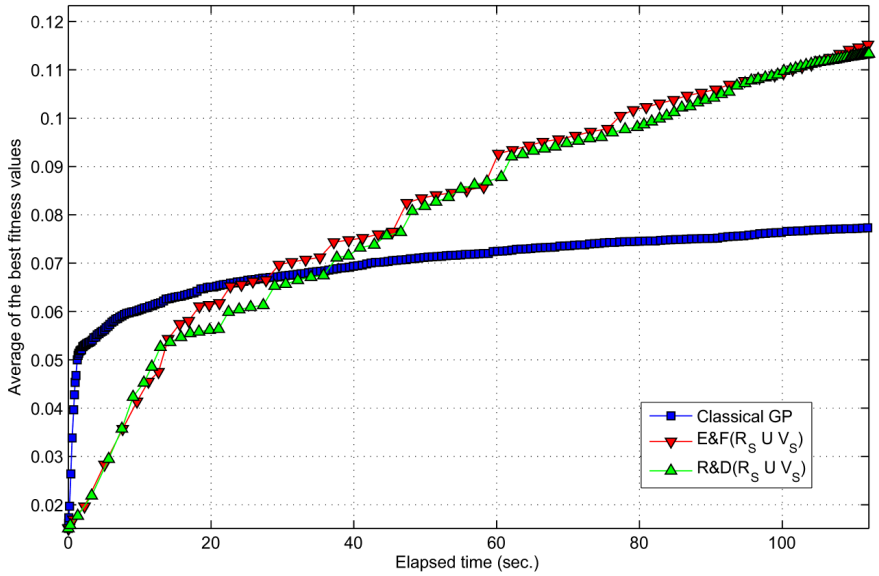


Figure 7.3 - Average of the best fitness values versus time for the F_1 function.

Function MF_1

	Best adjusted fitness values		**Elapsed time (min.)**
	Average	Standard deviation	
Classical GP	0.7983	0.0789	700
E&F $f_{RD}(R_S)$	0.8812	0.0315	424
R&D $f_{RD}(R_S)$	0.8749	0.0361	328
E&F $f_{RD}(V_S)$	0.8659	0.0419	310
R&D $f_{RD}(V_S)$	0.8473	0.0499	**239**
E&F $f_{RD}(R_S \cup V_S)$	**0.8983**	0.0238	607
R&D $f_{RD}(R_S \cup V_S)$	0.8874	0.0348	462

Table 7.2 - Results for the function MF_1.

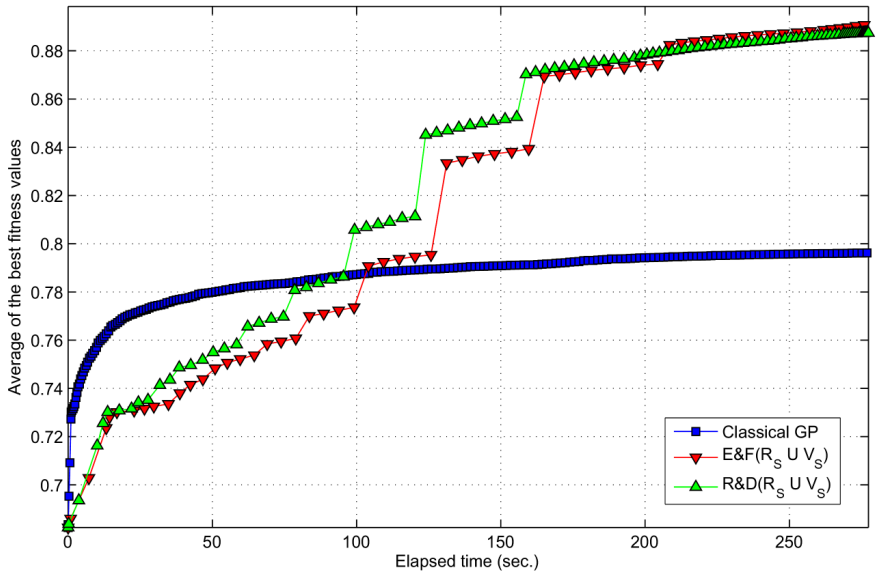


Figure 7.4 - Average of the best fitness values versus time for the MF_1 function.

Function MF_2

	Best adjusted fitness values		Elapsed time (min.)
	Average	Standard deviation	
Classical GP	0.7343	0.0641	1487
E&F $f_{RD}(R_S)$	0.7743	0.0595	548
R&D $f_{RD}(R_S)$	0.7616	0.0582	301
E&F $f_{RD}(V_S)$	0.7466	0.0583	422
R&D $f_{RD}(V_S)$	0.7393	0.0543	**230**
E&F $f_{RD}(R_S \cup V_S)$	0.7908	0.0534	566
R&D $f_{RD}(R_S \cup V_S)$	**0.7983**	0.0607	486

Table 7.3 - Results for the function MF_2.

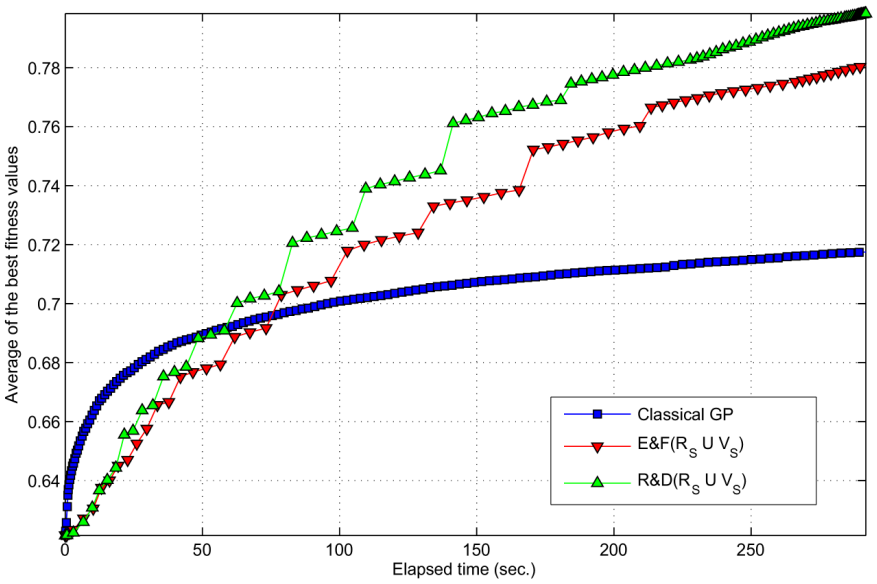


Figure 7.5 - Average of the best fitness values versus time for the MF_2 function.

Appendix C

In this appendix we reported the results obtain for the last version of the *R&D* strategy on the numerical functions and benchmarks in the boolean domain.

Function F_1

	Best adjusted fitness values		**Elapsed time (min.)**
	Average	Standard deviation	
Classical GP	0.0814	0.0526	300
E&F $f_{RD}(R_S \cup V_S)$	0.1053	0.0552	230
R&D $f_{RD}(R_S \cup V_S)$ (version 1)	**0.1133**	0.0884	**187**
R&D $f_{RD}(R_S \cup V_S)$	**0.1257**	0.0793	8262

Table 7.4 - Results for the function F_1.

Function MF_1

	Best adjusted fitness values		**Elapsed time (min.)**
	Average	Standard deviation	
Classical GP	0.7983	0.0789	700
E&F $f_{RD}(R_S \cup V_S)$	**0.8983**	0.0238	607
R&D $f_{RD}(R_S \cup V_S)$ (version 1)	0.8874	0.0348	**462**
R&D $f_{RD}(R_S \cup V_S)$	0.8996	0.0452	10873

Table 7.5 - Results for the function MF_1.

Function MF_2

	Best adjusted fitness values		**Elapsed time (min.)**
	Average	Standard deviation	
Classical GP	0.7343	0.0641	1487
E&F $f_{RD}(R_S \cup V_S)$	0.7908	0.0534	566
R&D $f_{RD}(R_S \cup V_S)$ (version 1)	0.7983	0.0607	**486**
R&D $f_{RD}(R_S \cup V_S)$	**0.8161**	0.0689	13460

Table 7.6 - Results for the function MF_2.

k-majority on problem

	Percentage of success		**Elapsed time (min.)**	
	5 bits	7 bits	5 bits	7 bits
Classical GP	97	17	24	649
E&F $f_{RD}(R_S \cup V_S)$	**100**	46	91	203
R&D $f_{RD}(R_S \cup V_S)$ (version 1)	**100**	**57**	4	**194**
R&D $f_{RD}(R_S \cup V_S)$	**100**	**69**	**2**	1017

Table 7.7 - Percentage of success and elapsed time for the k-majority problems with k=5 and k=7.

k-Bits Multiplexer problem

	Percentage of success		**Elapsed time (min.)**	
	6 bits	11 bits	6 bits	11 bits
Classical GP	91	14	33	2005
E&F $f_{RD}(R_S \cup V_S)$	**100**	40	107	898
R&D $f_{RD}(R_S \cup V_S)$ (version 1)	**100**	26	**1**	**780**
R&D $f_{RD}(R_S \cup V_S)$	**100**	**63**	2	2006

Table 7.8 - Percentage of success and elapsed time for the k-Bit multiplexer problems with k=6 and k=11.