UNIVERSITÀ DEGLI STUDI DI MILANO - BICOCCA

Dipartimento di Informatica Sistemistica e Comunicazione

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Magistrale in Informatica

Variable Size Populations for Dynamic Optimization with Genetic Programming

Relatore: Dr. L. Vanneschi
Correlatore: Prof. G. Mauri
Controrelatore: Dr. G. Di Caro

Candidato: *Giuseppe Cuccu*
Matricola: 055065

Anno Accademico 2008-2009

Contents

I Introduction **7**

1 Directives **9**

1.1 Abstract 9
1.2 Introduction 9
1.3 Hypothesis 10
1.4 Thesis' structure 10

II Phase 1 - Study **11**

2 What are Genetic Algorithms **13**

2.1 Imitation and evolution 13
2.2 Darwin's theory 13
2.3 General Characteristics 14
2.3.1 Search 14
2.3.2 Individual 14
2.3.3 Population 15
2.3.4 Fitness 15
2.3.5 Generations 15
2.4 Genetic Operators 15
2.4.1 Elitism 16
2.4.2 Selection 16
2.4.3 Crossover / Reproduction 17
2.4.4 Mutation 17
2.5 Application 17
2.6 Algorithm 18

3 From GAs to GP **19**

3.1 Generalized GA 19
3.2 Search space redefinition 19
3.3 Evaluation and Fitness 19
3.4 Individual redefinition 20

3.5 Note: on the bloat problem 20
3.6 Crossover redefinition 21
3.7 Mutations 21
3.7.1 Subtree mutation 21
3.7.2 Point mutation 22
3.7.3 Shrink mutation 22
3.7.4 Swap mutation 22

4 Previous applications of GP to DOPs 23
4.1 The road so far 23
4.2 The main difference 24

III Phase 2 - Implementation 25

5 ANSI Common Lisp 27
5.1 “Lisp?” 27
5.2 Speed in LISt Processing 28
5.2.1 Programmer time 28
5.2.2 Necessity vs. Possibility 28
5.2.3 Fast prototyping 29
5.2.4 Dynamic typing 29
5.2.5 Data structures 30
5.2.6 Macros: programs that write programs 30
5.3 Individual interpretation 31
5.4 Modularity 31
5.5 Different languages integration 32

6 Building a GP framework 33
6.1 A programmer’s job 33
6.2 Tools, loader and test 33
6.3 Tree management, individuals and population 34
6.4 Individual evaluation and fitness 34
6.5 GP operators 35
6.6 DOP simulator: MFFL 35
6.7 Flow control and report 35
6.8 Problem definition 36

7 Testing the framework 37
7.1 Function regression 37
7.2 On function regression and Lisp evaluation 37
7.3 Again on Lisp speed: optimizations 37
7.3.1 Why still lists 37

7.3.2 Why not an ad hoc evaluator 38
7.3.3 Type declarations 38
7.3.4 Reload and rebuild 38

IV Phase 3- Innovation 41

8 GP, DOP and DSP 43
8.1 DOP - Dynamic Optimization Problems 43
8.2 DSP - Dynamic Size Populations 43
8.3 Applying DSP on DOPs 44
8.4 Expected results 44
8.5 DIV and SUP algorithms 45
8.6 Why not DIV and SUP 45
8.7 Necessity and birth of a new function 46

9 Growing a new control 47
9.1 Note: on orthogonality 47
9.2 The pivot 47
9.3 The fitness function 48
9.4 The population function 48
9.5 Strength 49
9.6 Blend together 50

10 Limitations 51
10.1 Again on bloat: focus on population 51
10.2 Upper limit: Flush operator 52
10.3 Lower limit: Stand-by 52

11 On modularity and parallelism 53
11.1 Real time and testing 53
11.2 The SCILX cluster 53
11.3 Breaking the jobs 54
11.4 Rebuilding the output 54

12 Back to implementation 55
12.1 A closed road - the first DOP-sym 55
12.2 A new DOP simulator: Multi Function Fitness Landscape 56
12.3 Controls 57
12.4 Stats 58
12.4.1 GNUplot vs MatLab 59
12.4.2 On MatLab reports 59
12.5 Beyond mere Lisp: integration of three languages 59

V Make a move 61

13 Test session 63
13.1 Test definition 63
13.2 Generating the test cases 63
13.3 Experimental results 65
13.4 Conclusions 67
13.5 Contributions 68
13.6 Note: on GECCO 2009 69

14 Walking towards the future 71
14.1 Implementation status 71
14.2 On computational effort and best fitness 71
14.3 Multipopulation 71

Part I

Introduction

Chapter 1

Directives

1.1 Abstract

A new model of Genetic Programming with variable size population is presented in this thesis and applied to the reconstruction of target functions in dynamic environments (i.e. problems where target functions change with time). The suitability of this model is tested on a set of benchmarks based on some well known symbolic regression problems. Experimental results confirm that our variable size population model finds solutions of the same quality as the ones found by standard Genetic Programming, but with a smaller amount of computational effort.

1.2 Introduction

Many real-world problems are anchored in dynamic environments, where some element of the problem domain, typically the target, changes with time. For this reason, developing solid evolutionary algorithms (EAs) to reliably solve these problems is an important task. The application of evolutionary computation (EC) to dynamic environments creates challenges different from those encountered in static environments. Foremost among these, are issues of premature convergence, and the evolution of overfit solutions.

In the last few years, many contributions have appeared which studied dynamic optimization environments and developped new evolutionary frameworks for solving them. Nonetheless, the majority of those approaches are based on Genetic Algorithms (GAs) [18, 15] or Particle Swarm Optimization (PSO) [21, 28, 8] and the problem objective is finding the extrema (maxima or minima) of a target function that changes with time. On the other hand, very few contributions have appeared to date studying the ability of Genetic Programming (GP) [22] to recon-

struct target functions on dynamic optimization environments.

1.3 Hypothesis

In this thesis we hypothesize that variable size population GP outperforms standard (fixed size) GP on dynamic optimization problems. This idea is not new in evolutionary computation, for instance, it has been applied to PSO in [13]. however, it has never been applied to GP before. We propose a variable size population GP model called *DynPopGP*. It is inspired by the one presented in [29]. Simply speaking, it works by shrinking the population when fitness is improving and increasing its size, by adding new genetic material, when the evolution stagnates.

Our hypothesis is that when the target function changes, evolution of the current population should stagnate (because it has been evolved to approximate the previous target function). Thus the evolution should benefit from the creation of new genetic material, that should give the necessary amount of diversity to start the optimization of a new target function.

1.4 Thesis' structure

This work is structured as follows.

1. The first part is this introduction.
2. The second will cover the first part of the work. I first had to update myself to the state of the art of the GP techniques. There are been too searches for GP application on dynamic optimization problems, with small results.
3. The third part will explain our choice of implement a new framework on our own, instead of using one of the present. Will also describe the effort made and results obtained by the implementation phase.
4. The fourth part will be about innovation. We'll describe the core problem of this thesis, the solutions adopted, and the particular implementations.
5. The fifth and last part will connect present and future. We'll give the test results of our framework, demonstrate the usefullness of the chosen techniques, and trace a guideline for further studies and expansions.

Part II

Phase 1 - Study

Chapter 2

What are Genetic Algorithms

2.1 Imitation and evolution

When it comes to greatness, nobody beats good old Mother Nature.

Engineers know this for true since long, and are used to copy Nature's pattern and behaviors. Think about how many great "human inventions" are directly derived from Nature's design: bridges, boat shapes, photo-camera's sensors, sonar. Nature's design simply works: that's because it had long, long time to be tested - and refined.

2.2 Darwin's theory

Charles Robert Darwin. Medic, voyager, geologist, naturalist. A great life of science, scoping almost all 19Th century. He was there when science was defined, he defined a good slice of science.

It's masterpiece, "The origin of species" [10] (1859), states the bases of Natural Selection, on which in turn are based the evolutionary algorithms:

- Reproduction - "Species have great fertility. They have more offspring than ever grow to adulthood."
- Variation - "In sexually reproducing species, generally no two individuals are identical."

  "This slowly effected process results in populations that adapt to the environment over time, and ultimately, after interminable generations, these variations accumulate to form new varieties, and ultimately, new species."
- Adaptation - "Some of this variations directly affect the ability of an individual to survive in a given environment."

- Ereditariety - “Much of this variation is inheritable.”
- Competition - “Individuals less suited to the environment are less likely to survive and less likely to reproduce, while individuals more suited to the environment are more likely to survive and more likely to reproduce.”

  “An implicit struggle for survival ensues.”

  “The individuals that survive are most likely to leave their inheritable traits to future generations.”

Evolutionary algorithms take those keypoint, and convert them in operators that work on data.

Other two point from the original Darwin’s writings are worth mentioning.

- Populations remain roughly the same size, with small changes.
- Food resources are limited, but are relatively stable over time.

Our game will be played on the meaning of “roughly the same size”. It doesn’t mean “static”, he better intended that the population size will adapt to the environment conditions. That’s exacly what we’re about to do here.

The conditions, in nature, are relatively stable over time. Our thesis is that this is not a limitation to the model, and we’ll try to prove it useful on dynamic changing environments.

2.3 General Characteristics

We’ll here discuss the general characteristics of EAs.

2.3.1 Search

Evolutionary algorithms are fundamentally search algorithms. They search in the space of solutions, for a given problem. It’s the problem that defines the space. Because of this, the search space is usually too wide to be deterministically explored, and the elements are very complex.

An element of the search space, is called an individual.

2.3.2 Individual

An individual is something that carries determined characteristics into its genes, that is its configuration. In EAs, individuals are striped

off of everything else: they're mere genes. Particularly, in GAs they're coded as binary digits sequences. Every bit contributes to the individual behavior, and all individual share the same structure characteristics (e.g., in GAs, the string length).

2.3.3 Population

A population is a set of individuals. GA works on population to gain from evolutionary theory. Is on population that Darwin's key point are applied. What we want to do with individuals, is to sort them in order to select the best characteristics between all of them, to build a single optimum individual. A population is so a simply pool of genetic "research material", that we are constantly improving, moving toward our aim. It's our set of hooks to the search space.

2.3.4 Fitness

The fitness function gives, for each individual, a measure of how it is "fitted" for the current system. Of course, this is a very general definition: the fitness function can be anything. It just has to take the individual as an input, and output a member of a fully ordered set. The most common implementations are functions whose unique input is an individual, and whose output is a real number.

The interpretation of this output is correlated to the class of problem - minimization or maximization.

2.3.5 Generations

The concept of time in GAs is enclosed in the discrete concept of generation. Generation is one complete application of the main loop of the algorithm (see next).

It encloses the concept of time because any temporal transaction, or change in the external system, can only be recognized after the core loop is completed. A generation is composed by sequential application of all the genetic operators, to build a new population.

2.4 Genetic Operators

The genetic operators are functions that implement the Darwin's directives, and act as natural selection on the population.

2.4.1 Elitism

Elitism is a simple technique that allows fitness to be a non-decrescent function. It simply copies the best individual of the actual population into the new one, as its first element. Now, let the fitness function be unchanged since last generation, we are assured that the minimum best fitness we'll find will be that of our old best individual.

2.4.2 Selection

The selection operator's role is to find couples of partners for crossover or reproduction. This selection is based on both chance and fitness score of the individual.

One fundamental characteristic is that every individual should have a non-zero possibility to be picked. Like in Las Vegas, we only have to grant this very small possibility, and repeat over a big number of runs, to win.

The best known methods of selections are Roulette Wheel, Ranking and Tournament.

Roulette Wheel

The Roulette Wheel method gives every individual a selection probability of

$$P_i = \frac{\Phi(fitness(i))}{\sum_{J=1}^{N} fitness(j)}$$

Which means that the possibility to be selected for each individual is directly proportionate to its fitness. This method relies in the differences in fitness between the individuals.

Ranking

While the Roulette Wheel selection gives a probability in relation to the average fitness, the Ranking selection works on the median:

$$P_i = \Phi(getPosition(i, orderByFitness(population)))$$

So its probability to be picked is proportionate to its position on the population ordered by fitness. This method flattens the fitness differences between individuals.

Tournament

While the Roulette Wheel method requires for each individual to compute its fitness; and the Ranking method even requires to order the population; the Tournament method has less requirements.

With the Tournament method, you have to define a certain K number of individuals, each of which can be picked up at random again even if it had already been picked up. The method will then return the best individual fitness-wise from this pool.

2.4.3 Crossover / Reproduction

The crossover operator works exactly like the genetic crossover. Only, in this case we obtain two children, by crossing two parents, so they have some characteristics of both parents, being different from them. Our goal is to blend single useful parts from high-score parents, to build a new individual that's better than them.

Reproduction comes when crossover doesn't. That is, for each crossover, there's a small probability (usually ~ 0.10) that the parents won't cross. In this case, the two parents go untouched in the next phase: the mutation.

2.4.4 Mutation

The mutation operator is the genetic representation of randomness. Every bit of every individual has a small probability (usually ~ 0.05) to change, before being inserted in the new population.

2.5 Application

Every kind of information can be coded (in the straight sense of mapping) as a bit string, by adopting the right interpretation. To apply the GA to a problem, we have to define the characteristics of the problem, and then to run the standard algorithm. The interpretation translates our results in problem solutions.

First, we have to describe the search space. That is, the formal description of a problem solution structure, plus optional other boundaries. Then, We map the solution characteristics to a binary string structure, that can be computed by the GA. Last, we construct a fitness definition, through a rating system for the individual to solve our problem.

2.6 Algorithm

Done this, the algorithm is pretty simple to run:

- Generate the initial population, either from seeding (if present) and random individual generation
- Start main loop: repeat until maximum generation reached, or until the best individual of the population has sufficient fitness
  - Start the genetic operators loop: repeat until the new population has the same dimention of the starting one
  - Elitism: take the best element of the actual generation, and put it first in the next generation
    - Selection: find two parents in the actual generation
    - Crossover/Reproduction: apply crossover or reproduction to the parents, to create the two children
    - Mutation: apply mutation on the obtained individuals
    - Put the two individuals in the new generation; if the new population is now filled, return it
  - Find the best individual in the population. If its fitness is sufficient, return it, else get back to the loop and run another generation

The fundamental properties of the algorithm have been demonstrated on its convergence theorem [26].

Chapter 3

From GAs to GP

3.1 Generalized GA

We now have to look at the step taken from GAs to GP. It's a simple generalization step, it just add another layer of abstraction, but its implementation isn't as simple as it seems. It was hard to keep at focus, and in one occasion we stepped into a dead way because of this, as we'll see later.

Genetic Programming (GP for short) is a technique that takes the GA paradigm to higher consequences. It adds a new level of abstraction: individuals are no longer solutions, but solvers. We have a problem described as a series of data, and we don't know how to obtain it, so we can run a sort of GA that looks for a solutor.

We mount together instructions instead of bits, to build tree structured individuals instead of binary strings. Our hope is that one of the resulting individuals, on given input, will perform the expected output.

If not, we'll use the above EA techniques (fundamentally, the same genetic operators as GAs) on the individuals, to refine our results.

3.2 Search space redefinition

We start by redefining the search space. As I said, we're no longer looking for the right input for a solver (the fitness function), but we're looking for a solver that will perform as expected (or, at least, wanted) on given input.

3.3 Evaluation and Fitness

We have a different kind of evaluation for the individuals. We no longer have data to pass to a simple function that computes the fitness.

The concept of fitness is now a score of how near is our individual's behavior to the one we want. We have to define a certain number of test case with which try its behavior, and a concept of distance between the obtained outputs and the ones we would like.

We therefore need to re-define the fitness: we need some function that can score how good is an individual for our use. Only, that case is complicated enough to see the simplest fitness functions evolve into a standalone complex scoring program.

3.4 Individual redefinition

The individual structure can be no more a binary string. Well, yes, we could map a GP problem into a GA one, but if we explicit the individual structure we can operate more selectively and purposefully on it.

So, we have to define the individual as a tree structure: internal nodes are called functionals, that are functions with ariety ≥ 1, while leaves are terminals: constants, variables and 0-ariety functions. The arcs give us the order of evaluation and composition of the individual.

Now it's time to see how we make use of this new definition to manipulate an individual. We define two main structure interaction: for each node, we can alter either its content or its structure - and that means to alter the subtree rooted in the node. This way, we have full control of the individual composition, and we can alter or exchange logical subparts between individuals.

3.5 Note: on the bloat problem

The bloat problem my be regarded at this time as the fundamental problem of GP [1, 26].

While in GA the size of individuals and population are fixed, the size of the individuals in GP might change. We have two dimensions to consider in this case: the structural tree depth (the longest distance root-leaf, in number of nodes), and the individual size (the count of all the nodes in the structure).

This problem is double-edged. For one side, we want not to limit the complexity of the individuals. We do not know how complex would the optimal program be, so we cannot limit its dimensions. On the other side, the hardware on which the GP runs is limited, so we cannot let the individuals become as large as they want.

The dimension of the data structures in the system is demonstrated to diverge. We can calculate (and limit) the quantity of data in the process by both individual size, individual depth, and population size.

Taking the population size into consideration is something not widely used, because it usually is a fixed number during the execution. However, a widely accepted measure is the computational effort, as defined in [14]:

$$E_g = E_g = PE_g + PE_{g-1} + \cdots + PE_1$$

$$PE_g = \sum_{ind \in Pop} count_nodes(ind)$$

Where the partial effort at generation g (PE_g) is defined as the sum of the sizes (number of nodes) of all the individuals in the population at generation g.

Given that fitness calculation is often the most computationally expensive part of an EA and that in GP this calculation largely depends on the size of the individuals in the population, this measure clearly gives an idea of the computational complexity of executing a GP model (as claimed in [14]).

However, we will find more about it later.

3.6 Crossover redefinition

Redefining the crossover is now easier. Like in GA it was a break in the logical structure, and an exchange of subparts, the same is in GP. We just randomly find a different node in both individuals, and then exchange the full subtrees.

The crossover operator is one fundamental cause of bloat.

3.7 Mutations

The possibility to alter not only the content, but also the structure of an individual, introduces new different definitions of the mutation operator.

3.7.1 Subtree mutation

The standard used method is the subtree mutation. A node is randomly selected in the individual's structure, and then the subtree rooted in it, is substituted by a random generated subtree. This randomly affects both content and structure.

The subtree mutation too is cause of bloat.

3.7.2 Point mutation

The point mutation affects only contents. The content of a node is randomly substituted with a terminal or functional of the same arity.

It doesn't affect bloat.

3.7.3 Shrink mutation

The shrink mutation tries to put some control in the mutation process. A random node in the individual, and the subtree rooted in it, are substituted with a descendant of this same node. This assures that the final individual will be shorter - that is, of inferior size. Note that to decrease an individual's depth we have to shrink its longest branch.

Because of its peculiarity, the shrink mutation is often used to control bloat on single individuals - for example, by forcing it on individuals over a certain maximum size (or depth) until it returns behind the line.

3.7.4 Swap mutation

The swap mutation is another controlled mutation technique. It permits to alter both content and structure of an individual, without the need for external material. It simply swaps two random nodes in the individual. The tree depth might change, but not the individual size.

Because of this, it doesn't affect bloat.

Chapter 4

Previous applications of GP to DOPs

4.1 The road so far

Over the past few years, a number of authors have addressed the problem in many different ways. Surveys of these studies can be found for instance in [3, 4, 5, 6]. Interestingly, in [4] Branke proposes a classification of these approaches into four broad categories:

- Approaches that run EAs in a standard fashion, but, as a change in the environment is detected, take actions to increase diversity and thus facilitate the shift to the new optimum [9, 31]
- Approaches that reinject diversity in the population at all the time, hoping that this can allow the EA to adapt to changes more easily [17, 23]
- Approaches that supply EA with memory, able to recall useful information from past generations, which seems especially useful when the target repeatedly returns to previous locations [16, 25, 11, 2]
- Approaches that split the EA population in multiple subpopulations, some to track known local optima, some to search for new optima [7, 30].

More recent contributions include [32] where, based on the concept of problem difficulty, a new dynamic environment generator using a decomposable trap function is proposed; [19], where a coevolutionary agent-based model is used; [27] where the use of mutation for diversity maintenance is investigated and [12], where the use of artificial im-

mune networks for multimodal function optimization on dynamic environments is studied.

4.2 The main difference

All the above quoted contributions treat the problem of tracking the extrema in a dynamic environment, where the target function changes with time and concern GAs, PSO or other EAs variants. Very few contributions have appeared to date dealing with the (more complex) problem of approximating/reconstructing target functions that change with time by means of GP.

The goal of this thesis is different: first of all, we want to study standard tree-based GP [22], and one variant thereof using variable size populations; secondly, we want to employ more simple, and thus easier to study, test problems.

Part III

Phase 2 - Implementation

Chapter 5

ANSI Common Lisp

5.1 “Lisp?”

“Lisp?”. This one-word question is the one I heard most in the past few months. You must understand, it’s not a question about something - it’s an automated repetition in search of confirmation, from someone that judges more likely to have misheard the word than to accept what it implies. Yes, I used Lisp.

Lisp is a tool. A tool that’s more than 50 years old, in a field - Computer Science - that totally revolutions its tools every decade or so. Or isn’t it?

Why Lisp? Because while Fortran started from hardware and then tried hard to represent problems, Lisp was born to explain problems, and then got to be faster. It was born from math, and in 50 years it has done a pretty work about speed.

Lisp is about math. We should not compare Lisp to Cobol, or to an 80486 chip. We shoul more likely compare it to the Quicksort algorithm, that after 40 years of revolutions is still the fastest ordering method known.

Just one note on naming: Lisp is merely a model, that you can use to build an interpreter/compiler, and run as a language. There are lots of Lisp languages out there, the most known of which are Scheme, Autolisp and eLisp. And, obviously, Common Lisp, which ate all previous dialect into its implementation. In this work, while naming only Lisp for short I’ll always talk only about Common Lisp, as described on Common Lisp - The Language 2 (CLTL2).

5.2 Speed in LISt Processing

Everyone's first critic on Lisp is about speed. I never meet anyone who experienced first-hand that Lisp is effectively slow, but everyone seems to share the same opinion.

Of course Lisp is slow, from a C point of view. But that's because C forces you to tailor the program for the compiler, you have no other paradigm that the imperative one, and the most complex data structure at your disposal is a contiguous ram allocation called array. And that's why now language designers try to highen the abstraction layer - that is, move toward Lisp. First with C++, then with Java, with an incredible number of minor branches in between. Not to count the awesome new languages born on the Internet - Perl, Python and Javascript leading them.

You want a slow language? Java is one. Lisp can be optimized and compiled to reach C-like speeds (yes, in some cases even faster); programming in C you have to, and programming in Java you can't.

But with Lisp programming complex concepts is much simpler. And if you can save one month on three of implementation, you usually don't care if your program takes two hours instead of one to run.

5.2.1 Programmer time

During the last couple of decades, Moore's law worked flawlessly. While still 20 years ago the critical time was computational time, modern hardware is dragging the concern towards programmer's time. Brutally.

The cost now is about the programmer, not the hardware. Computational time is cheap, hardware evolves fast and is easily changed. A new equation enters the system: less programmer time on a slower but more powerful language, plus the cost for new hardware, is cheaper than having the same programmer working for more time to achieve the same prestation on the old hardware.

5.2.2 Necessity vs. Possibility

The first key point in Lisp is about flexibility. You're not required to care about anything - but you can. This permits you to concentrate first on the problem, and you can think about the details just after your prototype works.

Dead branches are throw away after very small effort, while you can optimize a working solution down to assembly instructions. You start with a Read-Eval-Print Loop (REPL for short) in which you can type

(+ 3 4) and immediately get 7 as an answer, but you can use the same environment to design with Object Oriented paradigm.

Script manipulation with the FORMAT sublanguage, or iteration control with the LOOP one, or simply use plain good old Lisp. The suggested paradigm is the functional one, but you can write totally imperative programs. Recursion, iterations, even gotos:

the point here, is that you have 50 years of reliable experience under the hood, so you can play up and down the ladder as much as you like.

5.2.3 Fast prototyping

The opulency of Lisp tools is amazing. It's not about libraries: it's a whole language that's structured to feel like one single entity, that you can seamlessly expand with new operators and macros.

The fast prototyping approach is what pays most here. You can build a working throwaway example in minutes. Then, just by addin astraction, generalizing and extracting the core operands, you can start a project. Which you can test one single function at a time by calling it directly from the REPL. Stepper, debugger and profiler are right there waiting to help you. Programming in a similar environment becomes really, really fast.

By the time your program grows, you can model the project around the problem. Would it be better to use only structures or a full object oriented implementation? How would it best to separate the code in files? What about using packages to avoid name issues, and use only generalized functions from the OO paradigm? You can postpone those questions until the answers present themselves, and the language will adapt itself seamlessly. Again, the beautiful thing is that you have to do nothing, just concentrate on the problem.

5.2.4 Dynamic typing

Dynamic typing means that you don't have to declare the type of a variable. Not that you can't.

In Lisp, values have type, not variables. You can bound a variable to anything, and then rebound it to something else. You don't have to care about that, it's the compiler that checks the variable content.

So there are only rare cases in which you would like to declare the type of a value: you can declare the number type of arguments in number crunching program sections. You can declare the content of an array, to let it use smartly the space (malloc-style). Or little more real-world cases: the chances that you could really need those declarations are really small.

That could be about 5 to 10% of the program, and the declaration can be automatically done (see Macros), leaving about the 90% of the code free from type declaration. So, if you have half the words to type in, you could spare some big time. And if you have half the rows of code to check, you could do half the errors.

I'll explain one last feature with an example. If you want to define a function that calculates the average in a list of numbers, with dynamic typing you don't have to care about which kind of numbers you'll receive. The $+$ and $/$ functions will also accept every kind of number, so you'll need not to write two different implementation to get the average of floats and integers. You can calculate the average over a list of mixed type numbers, even ratios, even complex, without any change to your code, and mantaining every number to its precision without casting anything.

5.2.5 Data structures

Not only type declaration errors are rare to be seen in Lisp, but pointer errors are too. The two most common errors of more common languages, are pretty unknown in the Lisp world.

Lisp does all the pointers work between stack and heap under the hood. It has garbage collection since over 20 years before Java was born. Some implementations have so fine-tuned garbage collector that is actually faster to use some memory and than throw it away that to keep it to re-use it.

The data structures and types in Lisp are many and very flexible. The most commonly used are cons cells, arrays and hash tables.

While array and hash tables are implemented exactly how a programmer would dream about (I'm primarly talking about length methods, iterators and modify functions), cons cells are pairs of pointers fundamentally used to construct lists. But lists of lists are trees, LIFO structures are queues, and FIFOs are stacks: they're very easy to build by cons cells, and thanks to dynamic typing you can use almost all predefined operators on those structures.

5.2.6 Macros: programs that write programs

If you think so far that Lisp doesn't seem so much alien, you still haven't see macros.

Many languages have some sort of macros, and many really advanced programmers simply consider the pre-processor that executes them... well, not sound, in the logic sense of speaking. Something not to trust, something that will behave very dangerously. A risk.

Lisp macros take fundamentally advantage of two characteristics. First, Lisp code is made of lists, hence data. Lisp is a list processor, don't forget. The second characteristic, is that you can add how many abstraction layers as you want, just explicitly calling the EVAL function on a list, or its derived FUNCALL and APPLY: that list will be interpreted as code, and executed (evaluated).

So, macros do exactly this: they are programs that write programs. When the REPL finds a macro, it first gets expanded, and then evaluated. The expansion process is the execution of the code in the macro; you have the full extend of the Lisp language at your disposal. The resulting list (macros have to return lists) will be put in the code in place of the macro, in compile-time, and the code will then be evaluated.

This grants you a power other languages don't even understand. But a Lisp programmer never has to write by hand the standard GET and SET methods over a structure: an automated macro defines them without needing partecipation from the programmer, that can stay focused on the problem.

5.3 Individual interpretation

So, enough about Lisp: here, we're talking about Genetic Programming. What's so good in using Lisp to write GP implementations?

One over all: you have the Lisp evaluator ready to evaluate your individuals.

In GP, you have to express the individuals as trees, modify their structure and then evaluate them on determined inputs. Incidentally, that's exactly how Lisp usually works.

Thanks to the functional paradigm, Lisp programs are naturally trees (lists of lists), and lists are the easiest structure to modify in Lisp. And there's plenty of functions that take a Lisp programs with some data as input, returning as output the application of the program on the data.

It's as easy as that, if you express individual as Lisp programs, you have the power of a full programming language at your hand to define your individuals, solutions for your problems.

5.4 Modularity

Lisp is about flexibility; flexibility is about modularity. You can flex something only if it's structure can be easily stretched: in Lisp, you can alter the structure of your code at your will.

First, you always have plenty of different ways to do anything. Lots of different functions that do roughly the same, lots of paradigms, lots

of controls. Then, you can test every single piece of code in the REPL, before calling it from your program. Last, functional style let the code be self-enclosed, and the order of the calls is easy to be tracked or built.

So, after you write fast a big chunk of code that realizes an example of what you want to do, you can extract all functionalities from it, and put them in different functions, and even abstract groups of them in macros. Last, you can put them at any point of the code files, while the declaration of a function is made before using it.

The feeling is that of construct a simply wooden miniature, declare it of rock and stone, and expand it to a big temple or pyramid. You end somethimes thinking “that’s too simple...”.

5.5 Different languages integration

The GP framework I worked to in this experience, made me think about language integration in a totally different way. I want to share here my experience.

With the SCILX cluster (see later), I could not simply call directly my program, but I had to personalize the single run for each test and test part, before doing so. But I had no problem with that: I simply called the REPL from the Bash shell, telling it to load my code, alter some variables, then run the test. Yes, from the shell, and without putting the hands into the code.

Then about the running. From most Lisp implementations, not only syscalls are possibles, but you can simply run others programs too. So my program plays a tune with *aplay* when the computation is done, generates some execution reports with *gnuplot*, and opens them with *gthumb*.

At the end, it wrote the stats in MatLab’s M-file format. With MatLab I ran those files, resulting in the graphs you’ll see in the Results section.

Again, the only word in my mind was “seamless”.

Chapter 6

Building a GP framework

6.1 A programmer's job

We'll start here a brief description of the main characteristics of the code we produced, as we organized it.

Please, when you'll find how short this program is in lines of code, try to see my effort to achieve that. An olympic gymnast will pull himself up the rings whith the most serene expression on his face, and without a tremor. But that's only because he did the same thing so many times before that only beauty remains in the act.

Lisp gives you great power of expression; but it really is hard to train. I re-wrote every single piece of this program from three to fifteen times. What you see here is just the demonstration of this program's and Lisp's capabilities, not the effort that lays behind. I'm proud of it, but let's face it: it's been very, very hard.

6.2 Tools, loader and test

The head of our project is composed of the tool suite, the loader and the test. Those three files contain all small utilities that are not inherent this particular implementation, but that are indispensable to the correct behaving of all other parts.

- The tool section contains all functions that are self-contained and can be directly re-used in other projects. You can find here short-hand macros, calculation functions and list manipulators.
- The loader section contains all parts devoted to code-loading and manipulation. You can find here all the optional DEFVAR declarations, the ordered load of all single files in the project, and the loading and compiling procedures.

- The test section contains all throwaway programs that have been fast-coded to test the environment. You can find here methods that build every kind of stat, profiler utils and variable backup methods.

6.3 Tree management, individuals and population

This sections are devoted to the management of the population's data structure. Since it's the fundamental data handled by the algorithm, we had to put special care in this method.

We studied many kinds of data structures, implementations, control function and hooks for future feature implementations (like multipopulation, see later). We finally decided for a simple and flexible cons-based tree structure.

- The tree management section contains all instruments that allow the manipulation of the underlying structure. That means for example the methods to alter the individuals' structure in the genetic operators.
- The individual and population section contains all basic management methods to generate, control and alter a collection of individuals. For example, methods to generate individuals, order populations and add or remove individuals from them.

6.4 Individual evaluation and fitness

Individual evaluation and fitness calculations come together. It's a single and small section, but one of the hardest to write. I experimented various techniques to evaluate an element, and finally chose to enclose it in a lambda expression via macro. That makes every individual to be treated like a function that was in the code since compilation.

One particolar issue came from the choice if it was better to compile the individual before evaluate it, or not. In Lisp you have a continuous control during all stages of execution: load-time, compile-time and run-time. So, as it's pretty normally to create a new function and load it at run time, it's also normal to compile it before loading.

In this case we choose not to do so. The true vantage of compiling vs. interpreting comes from the shift of some operations from run-time to compile-time. But being already in run-time, and being the single individuals pretty small programs in front at the GP execution, with its full population, it would be more the lose on compiling + loading + evaluating, than it is in just evaluate it.

To evaluate an individual, we evaluate its value as interpreted function, on given test inputs (see MFFL).

To evaluate the fitness of an individual, we do some simple steps:

- For each test case, we compute the output of the individual on the given inputs
- We extract from the test cases the target values (again, see MFFL)
- We calculate the root mean square error between the first series and the second. That will be the individual fitness.

The root mean square error is calculated as follows:

$$\frac{\sqrt{\sum_{i=1}^{N} (res_i - obj_i)^2}}{N}$$

With N number of test cases, res_i is the result given by the individual on test i, and obj_i is the wanted output on test i.

6.5 GP operators

The GP operators section simply implements all the genetic operators seen so far. There are also included the selection methods, and the methods to randomly chose a path through an individual's structure, to perform actions on the reached node and its substructure.

6.6 DOP simulator: MFFL

Our Dynamic Optimization Problem simulator, is called Multi Function Fitness Landscape. It is a complete simulator, in the sense that can easily simulate any kind of DOP. It's fully customizable to act as any function or composition of functions expressible in Lisp, to change its behaviour over time, and its behaviour can be manually or automatically changed at run time.

The Church-Turing thesis assure us that any computable function can be expressed by Lisp code; so there's simply no limit to what we can emulate with this software.

We'll talk more about MFFL section in the Innovation part.

6.7 Flow control and report

The flow control is the backbone of the program. It implements the algorithms that control the execution, plus every additional control

we coded - like particular tests, or the subpart launcher to be called in SCILX scripts.

The report part fulfills the goal of this paper as a reasearch effort: it constructs many kinds of reports and statistics, and serve them into ready-to-graph MatLab M-files.

6.8 Problem definition

The problem definition is one part that's enormously grown over versions.

We started with the basic GP parameters, like (initial) population size, or crossover and mutation probabilities, and the we started to go deeper. First on simple controls, like the limit depth to generate an individual, and the generation method, and down until functions set to the DOP simulator and the function to use to compose them.

One beautiful evolution to see has been the birth of a large number of control parameters, that have later disappeared from the file because hard-coded as automated controls. There are lots of them, lying in the code, and both grant an optimal value to be selected and relieve the programmer from the choice of values not directly related to the problem.

The great customization is one point of strength of this program, and the automatic adaptation of a big part of the code is another.

Chapter 7

Testing the framework

7.1 Function regression

Ok, so now we have a full GP implementation, and individuals can be everything that can be expressed in Lisp language. But, to test it, it's better to start small: let's see what can be done about function regression. The first implementation were about one simple function, with basic algebric operand and only two variables - for the sake of fitness landscape representation.

7.2 On function regression and Lisp evaluation

But every time the framework computes the regression of a simple function, we have to keep in mind that it's doing the same work that it would if it were to regress a computer program for a given problem. There's always the Lisp evaluator at the core, if we just change the algebric operands with language construct we can regress anything that can be expressed as a Lisp expression. And, because of Turing equivalence, that's the pretty large search space of all the computable programs.

7.3 Again on Lisp speed: optimizations

We've talked about optimizations when first discussing about Lisp. Nonetheless, very few optimizations have been applied to this program.

7.3.1 Why still lists

After long discussions, we've decided to keep lists (that is, chains of cons cells), for both individuals and populations. Let's take a look to our motivations.

First, individuals. We better keep the individual as lists, because this way they can be directly evaluated from the Lisp EVAL function, and because to keep the structure flexibility needed (crossover, mutations, etc.) cons cells are small prices to pay in terms of memory and speed - they're only ordered pairs of pointers.

Second, population. We've kept the population as a list, because our aim is to change its size dynamically. With list is very simple to insert or remove or even order elements, just changing the NEXT (cdr) pointers of the selected cells. Very handy.

Third, multipopulation. That's one of the future goals of the project: having multiple populations running on the framework, and having the individuals pointed by cons cells, we could swap individuals between populations as easy as swapping the two pointers.

7.3.2 Why not an ad hoc evaluator

Koza, in an early Lisp implementation, proposes an ad hoc basic algebra evaluator for function regression. That sure is faster than the function EVAL on evaluating aritmethic individuals. But the aim of this project was a broader objective, so we choose to keep the power and flexibility of Lisp's EVAL, even if paying in speed on number crunching.

7.3.3 Type declarations

All our actual tests are about function regression, whith algebric operators and trivial float numbers. So, why don't add at least type declaration, to every value and every intermediate step?

As simple as it sounds, it's all done through a macro. Before evaluate the individual on the test cases, there's a macro that declares every number present in the expression as float, and same for all the intermediate results.

While all the number crunching part gets the vantage of type declarations, there is not a single type declaration done by hand in all the program.

7.3.4 Reload and rebuild

A little aside note goes to the RELOAD and REBUILD functions.

Logics studies teaches that, when testing a program, you fundamentally need to look for two things: completeness and soundness. That means that it must do everything it's supposed to do, and nothing that it's not supposed to do.

To test completeness, you need an environment that has a good and verbose debugger, a step execution utility, and as much information as you can. For this reason, you could prefere interpreted code to compiled code, and because it's just about "does it start or doesn't it", you don't need fast code.

Testing soundness means that you have to check if the portion of code behaves like expected. You only need a profiler, to check bottlenecks, but you need fast, compiled code, to have multiple runs and check for unexpected outputs.

The RELOAD function is for consistence check: it deletes all compiled files (through shell interaction) to assure code interpretation, ask the compiler for slowest and least optimized code, and prepare the debugger for maximum verbosity. It then reloads the very files of the project, in which itself was written.

The RECOMP function is for run check: it deletes all old compiled files, ask the compiler for maximum code optimization and speed, removing debugger informations, and then re-compiles everything firsthand. It then load into the same REPL from which it was called, the same files that it has compiled, and in which it was written.

So, while working at a new feature, after defined a new function I would just call RELOAD to have all the system set to debug and the code reloaded anew. If there's a lexical error, the REPL would directly drop me into the debugger. When the feature works, to see the behavior of the feature in a couple of generation, I would just hit RECOMP, and then launch the main algorithm.

RELOAD and RECOMP are two example of hou you can construct a very programmer-friendly environment from Lisp, and are a classical example of how to tailor this beautiful language to your own taste.

Part IV

Phase 3- Innovation

Chapter 8

GP, DOP and DSP

8.1 DOP - Dynamic Optimization Problems

An optimization problem is a search for the best solution among the solutions' search space. Dynamic optimization problems are optimization problems whose target varies across time.

Most research on EC focuses on optimization of static, non- changing, problems. Many real-world optimization problems, however, are actually dynamic, and optimization methods capable of continuously adapting the solution to a changing environment are needed.

The aim of a DOP is to have at any time the best solution of the moment, even if we know it will change in time. We also have to be ready to follow those changes, and to find the new optimum as soon as possible.

Apply a GP to a DOP means that the fitness function can change, and we can't foretell when or why. We have to track it somehow, and quickly adapt our data.

8.2 DSP - Dynamic Size Populations

I should here talk about variable size populations, not dynamic. However, I choose this definition on my program to stretch out the correlation between the problem and population size dynamics. As we'll see in implementation, those are fundamental.

Dynamic size populations in GP are scarcely used. There are several explanations to this. It's usually easy to keep a fixed population size, so you can tune this size to the hardware capabilities and store the individual pointers in an array.

That's also an early optimization problem dragged from the evolution from GA to GP. In GAs, you have fixed individual length, so you can use

a population array to directly store the individuals. That's very memory efficient and very fast. But once you step from GA to GP, individual sizes are no more granted, and the best you can store in arrays are individual pointers. Nonetheless, there was no necessity to study variable size populations on GA, and the shadow of that still covers GP studies.

One last point, can be that those techniques are still work in progress, so you should not need a variable size population unless you're studying directly those. That means, if you're for example studying a new crossover operator, you first need to implement the simplest GP framework possible, and then you have to concentrate on crossovers. There's no spare effort in research.

8.3 Applying DSP on DOPs

But this study is directly aimed on dynamic variant size populations. The first thesis from dr. Vanneschi, which gave birth to this work, was amazingly simple.

Usually, on GP, an human can try different parameter settings to fine-tune the algorithm to the particular problem. But now we have a problem that changes at run-time. We have to get as many automated controls as we can and can't limit the population size to an initial extimation.

The fundamental intuition is divided among two cases.

- Let there be, among the best individuals of the population, the needed genetic material to regress the actual optimum. Then the algorithm will blend them generating better individuals at each generation. And the worst individuals of the population are of no use, and are slowing down the computation - and time is crucial on DOPs.

- Let there be not the needed material. Again, speed is crucial: we can wait and hope that crossover - or better more, mutation - will bring the needed material. Or we can add new individuals into the population, broadening the pool of genetic material, and adding new random individuals freshly generated.

8.4 Expected results

That's as simple as it sound, and the results are better than imagined. If we succeed to find an automated control that adapt the population size to the actual problem, we could have an algorithm that rapid adapts itself to both complex and simple functions. And, more important, to any number of variations between them.

We expect to improve the performances in two currencies. One is, we want to make less effort to regress a solution that is simply regredible. The other is, we want to be able to regress solutions that are so complex that require a bigger pool of genetic material.

Studies in this field have been done which are very encouraging. For example, Vanneschi's DIV and SUP algorithms [29] clearly demonstrated the vantages of an adaptable population size, giving us a strong fundation from which start.

8.5 DIV and SUP algorithms

The key to understand DIV/SUP algorithms is that they're functions with state that keep trace of the fitness progress in the last *period* generations. This trace is called pivot, and in function of it the algorithm decides if insert or subtract a pre-coded percentage of the actual population size.

Being % the protected division operator:

$$A \;\%\; B = A/B \text{ sse } B \neq 0, \text{ 1 altrimenti}$$

We can describe the difference between DIV and SUP by the way they update the pivot in generation g:

$$\text{DIV: } pivot_g = bestfit_{g-1} \;\%\; bestfit_g$$

$$\text{SUP: } pivot_g = bestfit_{g-1} - bestfit_g$$

8.6 Why not DIV and SUP

The problem is, here we are on dynamic optimization problems. This takes away two major basis for the DIV and SUP algorithms. First, the fitness of the individuals depends no more only on algorithms prestations. Our fitness function changes, and we have to recognize and track those changes. Second, with time becoming crucial, we can no longer let the population size without control. We have to limit it, or the algorithm will be get an answer to an old problem with the fitness change.

But that's a good start: we have now precise informations on what we need.

8.7 Necessity and birth of a new function

We know that the automated control system for the population size must be based on DIV and SUP, plus a control on the population size that should keep it as small as possible.

We should also look for a simple tunable function, because we have no experience on which rely to set this function's parameters. And it must be easily expansible, so that different implementations can put an accent on different characteristics than fitness and population size.

Now it's time to get back to paper and pencil.

Chapter 9

Growing a new control

9.1 Note: on orthogonality

The concept of orthogonality here is intended as a like of linear separability. This function was constructed keeping in mind that every variable should give a different and singolarly tunable contribution on the final output. And that every component should be totally indipendent from the others, so that adding and removing control could be a very simple experience.

9.2 The pivot

The first control is the pivot. The name comes from DIV and SUP: it's the part of the function that decides if add or subtract individuals to the next generation's population.

This implementation is very naive. We could no more rely on the stability of the fitness function over time, so we cannot build reliable statistics over the last N generations.

Three things we know for sure:

- If the population's best individual's fitness is improved from last generation, we can subtract some of the worst individuals.
- If it's unchanged, we can add some new individuals to help the algorithm to unlock from the dead point.
- If it's worsen, then the elitism grant us that the fitness function is changed, and still we have to add new individuals, possibly a bunch of them.

9.3 The fitness function

This control has to react to the absolute fitness of the best individual in the actual population (best fitness for short). We've identified three different behaviors:

- Acceptable fitness. If the best fitness fulfits the requirements, we have no need to waste computational power. Moreover, blocking the genetic operators helps maintain genotipic diversity in the population.

  In this case, we can keep just a fraction of all the genetic material at our disposal. A fraction that we consider significant, and that will partially seed the next population that will be created.

- Very bad fitness. That's a fitness that simply is too far from the objective. It means that we don't have very good material in the pool, and that there must be lots of improvement that can be done.

  So we have to alter the population as much as we can, by having a large number of individuals inserted or removed at every generation.

- Between the two stages, we can have a smooth transaction. We can have it by a line through the two previous point, in the function of fitness. We can use the following equation to generate the desired line:

$$\text{line_through}(x_1, y_1) :: y = y_1 + \frac{y_2 - y_1}{x_2 - x_1} \cdot (x - x_1)$$

We have so defined a limited line through two points, in the function of fitness.

This will lead to a coefficient that will be blended together with the others at the end.

9.4 The population function

This control has to react to the actual population size. We've again identified three different behaviors:

- Minimum population. We do not want to take the population size under a certain number. That's because that number ensures that the population holds enough genetic material to assure a fast reaction in case of fitness function change.

That number varies on hardware and problem definition: should be small enough to speed up the algorithm execution, but we have to take with us enough material to seed a new population when the fitness changes.

When the population hits the minimum, it's a good thing: the algorithm is having an easy life. So it really doesn't need to play much around with the population size: unless a fitness function change occours, the population size changes must be at minimum.

- Maximum population. That's the limit imposed by hardware. It must be a number of individuals that, hypothesizing them of average size, can be still computed in a fair time. It must be a number of individuals over which the algorithm starts to be too heavy for real applications.

  In this case, we have to play a lot with the population size, to get rid of as much useless material as possible and get new one as soon as we can. So the output coefficient must be high.

- Last point again for the smooth transition case: between the former two point there's a line of behavior that can work just well. This time, it's a function of the population size. Again, we can use the same equation to generate the desired line:

$$\text{line_through}(x_1, y_1) :: y = y_1 + \frac{y_2 - y_1}{x_2 - x_1} \cdot (x - x_1)$$

It's another limited line through two points, even if with opposite slope and in the function of fitness. This other coefficient too will be blended together with the others.

9.5 Strength

Up until now, we have different voices in the chorus. We have told anyone which tune to sing, because every one has an individual minimum and maximun coefficient to drop into the output.

The strength coefficient is the master gain. It amplifies or decrements the control output of the deciding functions. So we can use human readable values to judge the output limits, but still have a computation friendly value at output.

9.6 Blend together

To compose this values together, there's a simple product operation in place. So, every control issues a coefficient, that alters the global answer of the function among with the others and with the scale factor of the strength.

The result is a function that can be very easily controlled. You just have to decide how relatively important will be the fitness judgment, how the population size, and how rude will the function be in issuing big numbers.

The output of the function will simply be the number of individuals to be added (if positive) or removed (if negative) from the population, only in function of previous best fitness, new best fitness, and population size.

Chapter 10

Limitations

10.1 Again on bloat: focus on population

After working on the details, it's always good to take a step away and look at the work in its completeness.

What we are doing here is changing the dimension of the population. The population is the pool of genetic material used by the algorithm. So we can measure the quantity of data in play as the population size times the individuals' mean size. We are measuring the amount of data from two variables: individual size, and population size.

Again on bloat. Bloat is a useless and explosive growth in the genetic pool [1, 26]. It can come from two sources: individuals whose size has become very big, and a population incontrolled growth.

The effect is as if it was one. If we decrease the population size, it will cover the individual growth. But if we increase it, we can interpret it as a new kind of bloat: a big chunk of population is just composed of useless individuals. So we're adding data without gaining, in respect of have the useless individuals removed and swapped for new ones.

But we have to be careful, for the individual growth is not always bad. If we can't regress the function with individual of a certain size, maybe if we increase the individual size, and so the individual complexity, we would.

Think for example of the regression of a non-linear function by Tailor's polynomial approximation. The more the formula is long, the better the approximation. So if we limit the individual complexity, we cannot reach an ipothetical very hard fitness target.

We have to control both cases of growth, and to adjust one kind with the other. That's when the limits enter the play.

10.2 Upper limit: Flush operator

The flush operator is an upper limit for our DSP control. If the population tries to grow over a certain size, it cuts off a certain percentage of the population among its worsts, and then replace them with an equal number of newly generated individuals.

It does something similar to what the shrink mutation does on individual bloat, but instead of only shrinking the population size, it replaces the useless part with fresh generated individuals. That's because, if the population size climbed up to this limit, the algorithm isn't doing very well. We have to take a good pull to the execution to get it back on road.

10.3 Lower limit: Stand-by

The opposite, lower limit, is the stand-by. We reach stand by when we get to the target fitness.

In this case we're satisfied with the algorithm output, until the fitness function changes. So we can just stand back and let the time pass. Further genetic operators application would be deleterious, as they reduce genotipic variety. And, of course, those programs are too eager of computational power not to try to save as much as we can - think about multipopulation, or multithreading: every clock saved is a clock earned.

Chapter 11

On modularity and parallelism

11.1 Real time and testing

Lisp is actually two languages: a language to write slow programs fast, and a language to write fast programs slow. As for this early stages of developement, we just said that we'd buy programmer's time with computational time. So, now that the framework is ready on the road, we need the right engine to pull it.

11.2 The SCILX cluster

The SCILX cluster was born from a multy-faculty project in Bicocca, between Computer Science, Materials Science and Biotechnology. Is a Beowulf Cluster of 20 1.6GHz dual-core nodes, with 1GB ram per node, and a lot of high performance computing solutions. Everything is under the control of a front node and a Debian Lenny OS. It gets work for about 20 people.

The result from our point of view was to have 40 cores at disposal, with dedicated ram and shared persistent memory. And a scheduler that can take up to 200 jobs at a time, with walltime of 12 hours. When I learned of it, I almost feel McCarthy jumping with joy behind me. If a similar computational power were to be at disposal at the time of the Lisp Machines, maybe computer's history would have end up differently now.

All my part was done in just a couple of months. Now, if I was able to tame this monster, the test answers could be ready in a couple of hours.

11.3 Breaking the jobs

When I first choose Lisp, it was because I knew it would be the best instrument for me. Even if I didn't know exactly why - that is, what feature would come at my help in time of truble.

Many came to use before, but its modularity made the parallelization work like a charm. Using the functional paradigm, I added, aside to the main test call of the algorithm, a new parametrizable subcall that would make only part of the job - and save it in a format that could be easily recomposed.

So what I had to schedule as a SCILX job was no more a full test, but a fraction of it. That would permit me to use not only all 40 cores of the cluster, but really all 200 job schedules, leaving the cluster system to schedule them for the best, and having my computation saved every very few disposable results.

11.4 Rebuilding the output

The rebuilding now still used the flexibility of the Lisp. The key point was that backup and stats construction would be not time critic, in front of the computation time. So we could buy all the flexibility I'd like.

We backupped all partial works on separate files, and on Lisp-readable format. Using a simple LS from shell we selected all the files of the same test, and looped through them reading the content, appending it to a single test-output variable.

After the variables reading, we had all global informations about the test. So all we had to do was construct the relative statistics, and print them in M-file format, ready to graph generation by MatLab.

The whole statistics generator took about one more stand-alone program, but again, we're talking in Lisp terms: that's about 180 rows of well-indented code. Again, a charm.

Chapter 12

Back to implementation

12.1 A closed road - the first DOP-sym

The first DOP symulator implementation took us to a wrong road.

The search for DOP GP applications leave us with little clues. The most promising of all was [24], which at once I implemented. The strongest point made, which convinced me to implement it, was that he said it was a method to simulate any kind of DOP behavior.

It's core was the implementation of multiple "cone" functions. Those cones would be controlled each in shape via four parameters: x and y coordinates of the center, height and slope. By placing multiple cones in the same coordinates interval, we would construct the desired fitness landscape drawn by computate the cone response on each point, and then taking the max.

The DOP control comes in place by variating the cone parameters: by deforming and moving around the cones, we could easily change the fitness landscape to our will.

The problem rose from a critic we had to move and could not deny. Yes, the first critic born was the strict bound to two dimensional DOPs, making it just a toy.

Anyway, if it were to use in 2D GAs, this DOP simulator would assure total control on the fitness landscape peacks and slopes. So, while searching to the highest landscape point, we could change it by moving the peak, or simply lowering the highest or highening another to a new max.

But we are not working on GAs; we are working on GP. We have not to find the coordinates of the highest point, but a function that returns the same landscape shape for every point. And we have to fundamentally change this shape to simulate a DOP problem. The cone simulator would

change only constants in the landscape function, so limiting very much our power to alter it to our needs.

We had to start back from scratch, and to build a more general simulator of real life dynamic optimization problems.

12.2 A new DOP simulator: Multi Function Fitness Landscape

MFFL - Multi Function Fitness Landscape. We took in our hands all the experience we got from the cone simulator implementation, and used it to give birth to a generalization that would fulfil our goals.

The critic to the cone simulator was that the only changes we could make were about constants. So we had to base the MFFL simulator on deeper foundations.

- It should take any number of functions
- It should take functions of any number of inputs, even of different number among themselves
- It should compose the functions together with a function that can be passed in as a parameter, and even changed during execution
- It should return a function that can be computed for every point of the N-dimensional landscape it defines

We so used Lisp's function manipulation capabilities to let it construct our function at run time, based on the following parameters:

- A circular list of function to compose together
- The number of functions to be used at a time
- The step on used functions change
- The compositing function

We can explain this with an example. Let the function list be the following three-element circular list:

$$fns = \{(x+y), (x \cdot y), (x^2)\}$$

Let the number of function to be used at a time to be two; let the step to be one; let the compositing function to be the sum function.

$$n_{fn} = 2, step = 1, comp = function(+)$$

So, when the algorithm starts, the objective function will be the compositing via sum of the first two functions, hence

$$trg = (x + y) + (x \cdot y)$$

The fitness function will be on the function of some distance between the output of this function and the output of the individual, on the same input (same given x and y). Obviously, we should use multiple input as multiple test cases to interpolate, for each individual. By the way, the interpolation function of our choice has been the root mean square error.

Now, let's say the fitness function has to change. At given generation, an update method computes the next objective function. It does so by first stepping on the functions list by the given step, in this case one.

$$fns = \{(x \cdot y), (x^2), (x + y)\}$$

Then, it constructs the objective function anew with the previously seen method, giving as result:

$$trg = (x * y) + (x^2)$$

That's as simple as it seems: we have total control on which functions will be composed together, how they will, and when.

By controlling the compositing function, the step and the number of function at a time, we can smoother the fitness change, which translate on more or less effectiveness of the old population seed. And if we want to go wild, we can randomly decide the period before the next function change at every function change, and even add new functions on the functions list generated at random - we just have to generate a new individual and treat it as a function.

Lisp control is very elegant, when it comes to functions manipulation. Those are concept totally unthinkables in most other languages, but this kind of reasoning comes absolutely natural after dancing with Lisp a couple of years.

12.3 Controls

A great teacher once told me, talking about libraries: "When you're designing code that will be used from others as a tool, you simply cannot imagine how it will be used.

"People tend to use instruments not for the purpose they were built for, but for the purpose they are facing in the moment. If you want your software to be used, you cannot rely on the user, you must write robust

software. But if you want your software to be useful, you have to help the user hack it."

When writing this framework, I always kept this in mind: I will not use it. I will write it, I'll use it to test some feature, hopefully implementing it. But I won't use it in real world applications. More than that, I cannot imagine all the ways this software could be used for.

So, I said, let's write something very easily hackable. I designed a quantity of controls that still astounish me. I didn't think a project started so small would take so much parameter to fill alone a 300-rows separate file.

The input parameters are controls. From the problem definition file, you can control the problem parameters like terminals and functionals sets, the genetic operators' behaviour, the population parameters, the DOP MFFL simulator, the DSP controls, the stat generation and much more. You can go as high as manually seed a first generation population (even partially), or down as input the combining function for the fitness landscape functions. You can input a distance function to be used in the fitness definition, or you can write a fitness definition by hand that will overcome the default one. You can tailor this program to any form.

Really, this program can be anything, and the Lisp evaluation control permitted me to write beautiful automated controls, for many of those values that come only from try and error or from adaptation from other values

Two examples can be the delta-pop function, that adapts the population size on the actual algorithm need, generation by generation; or the adj-pop functions, that updates the population from the actual content (or seed) to a required size, adding or removing elements by automated criteria.

They work together, so you can just tell the algorithm to start with 200 individuals, pass to it 50 chosen individuals as seed, and watch it adjusting the population to 200 individuals, and then increasing or decreasing this start size to the number that most suits the actual execution.

People call it Machine Learning. I think it's beautiful.

12.4 Stats

This program is not about executions. Nor about tests. It's about stats.

We had a theory. We need to prove it. So we needed graphs on paper. Implementation, simulation and test were only consequences to

this need.

Because of this, great attention had to be put in the stats creation, manipulation and display. After Lisp's number crunching, we had multiple instruments at our disposal, and we needed to choose the fittest.

12.4.1 GNUplot vs MatLab

For the sake of semplicity, the first instrument we called out was GNUplot. It's a simple and relatively powerful plotting tool. It interacts naturally with Bash shell, so it was easy and fast to use directly from Lisp REPL. We just had to call it via Bash script to generate the requested graphs.

But the rival to GNUplot was from another league. After the graph generation complexity had grown as much as sight-placing errorbars on graphs, we had to look higher. There was MatLab waiting for us.

What GNUplot does with long commands, that have to be repeated every time, MatLab does by first calling a function easily defined in a separated time, and then by point-and-click. The graph customization is total and very easy, one of the great strength of this program.

Sadly, even if more direct and easy to call from Lisp environment, GNUplot simply stood no chance.

12.4.2 On MatLab reports

With MatLab reports, we could benefit from Vanneschi's previous MatLab experience on tracing graph for his papers. So, when he handed me some previously used M-files, there was little to do more than change the data inside.

It was only out of perfectionism that I separated data from function, to ease the M-file writing from Lisp. A little more research in the field was done to have minor adjustments - like printing errorbar only if a standard deviance was present among the inputs, and similars.

The result is a function M-file that works great with many kind of input, giving neat graphs. And a couple of Lisp's FORMAT tricks to write any kind of statistics on separated M-files that would use the function - with the right inputs.

This works great, and it's a very clean implementation.

12.5 Beyond mere Lisp: integration of three languages

Being this thesis so Lisp-oriented, I would like here to explain my position.

Usually, someone doesn't even think of trying another programming language than the one he knows best, because of the effort it would require, and the time it would need to build the same experience than in the old one.

Many people discuss wich language is the best, and usually the more they yell the less they know. I've met many people that say that Lisp is equal to any other language; but I never met anyone who claimed that another language was more powerful than Lisp. Parallely, every Lisper I ever met, told me that Lisp had something more. And, after two years of practice, I am of the same advice.

Even so, this thesis' work is written in three languages (not to take into account the indispensable LaTeX). The core is in Lisp, but the graphs come from MatLab, and the execution control are in Bash. And not out of will: I needed them.

Graph generation like MatLab's are yet to be seen in any command-line interface. And controlling SCILX in Bash was natural, while it would take me a nice deal of effort to code the same controls in Lisp.

Languages are tools. Lisp is more than a Swiss Army Knife, is half a garage alone. But can't paint better than brush and palette, and can't drive better than with steer wheel and pedals. I choose Lisp exactly to have the most provided garage ad disposal, the one which can do all others major garages can and even some things only it knows. But to be a fanatic of a tool, you lose the sense of reality. You can't paint nor move with a garage.

MatLab is my second preferred language: it's very intelligent and intuitive, and it allows you to treat big chunks of data as a single piece, doing all the detailed control under the hood.

And it was simply a pleasure for me to sink deep into Bash base OS control, and use it to tame SCILX - I found pleasure on some complex scripts.

So please, dear Reader, be sure that my objective in this dissertion wasn't to advocate the primate of Lisp, or to convince you to use it.

I've had a very complex problem to work on, and some wonderful tools that let me feel powerful. Here, I only wanted to share with you the beautiful pleasure of creation, to transmit to you how great I felt by overcoming those problems with subtle hacks.

I hope you can sense it.

Part V

Make a move

Chapter 13

Test session

13.1 Test definition

That is the time to ask ourselves the main question: are variable size populations really valuable on dynamic optimization problems?

The answer to this question is in our tests. Let's take a closer look.

We defined a test as a race between two executions. One would run the fixed population size algorithm (standard for short), while the other will run our dynamic size population algorithm (dynpop for short).

Each run would consist of 100 indipendent execution of the algorithm. Each execution would consist of 100 generations, with initial population of 200 individuals randomly generated with ramped-half-and-half technique. The fitness function will change every 20 generations: we force a particular case from the random possible set, for the sake of graph printing. We will exactly see the reaction of the program to the fitness function changes.

The real executions that took place on SCILX were subpart of this test, but after recomposition there would be no difference, so we can study this simplified version.

13.2 Generating the test cases

To test the framework, we need a set of functions to regress. Some benchmark problems have been defined for testing the performances of optimization methods in dynamic environments. In particular, Branke [3, 4, 5, 6] defines and uses *moving peaks* types of functions. In these benchmarks, hand-tailored fitness landscapes are defined and the positions of the extrema and their basins of attraction are modified with time. Similar problems are also used in [24, 32, 19, 27, 12].

However, this type of benchmark is not suitable for the present study.

In fact, in this work, we want to study the ability of GP to reconstruct dynamic target functions and not follow moving extrema. With this goal in mind, it would make no sense to use moving peaks benchmarks as the ones presented in [3, 4, 5, 6], given that, in those kinds of benchmark, extrema are moved by changing some additive or multiplicative constants to a (otherwise not changing) target function.

If one uses GP with linear scaling (introduced in [20]), the moving peaks problem reduces to a static GP problem, given that linear scaling allows to reconstruct the shape of the target functions, offering a method to automatically determine additive and multiplicative constants.

In this thesis we have define a new set of benchmark problems that can be used to test GP ability to reconstruct target functions in dynamic environments. We used those same functions [20] as the bricks for our construction.

The functions are defined as follows:

$$ls12(x,y) = x \cdot y + \sin((x-1)\cdot(y-1)) \tag{13.1}$$

$$ls13(x,y) = x^4 - x^3 + \frac{y^2}{2} - y \tag{13.2}$$

$$ls14(x,y) = 6 \cdot \sin(x) \cdot \cos(y) \tag{13.3}$$

$$ls15(x,y) = \frac{8}{2 + x^2 + y^2} \tag{13.4}$$

$$ls16(x,y) = \frac{x^3}{5} + \frac{y^3}{2} - y - x \tag{13.5}$$

We then take in group of a given size, and compose its result on the point with a sum function. As in [20], the fitness cases are created by generating 20 random values (with uniform distribution) for x and y in the range $[-3,3]$.

How many functions we get at a time? The more we take, the smaller the change will be when we throw away one to change with the next one. And the smallest the change, the more the initial seeding from last function's population will be effective. We hence defined three tests (BENCH1, BENCH2, BENCH3), to prove the effectiveness of the method through differently shaped DOPs.

The tests were crunched by SCILX in about 6 hours, with an average of two hours each. Each of the two runs of each test was divided into 25 jobs of 4 executions each, and the cluster took care of the rest.

13.3 Experimental results

Time for us for some execution. We thereby report here the results of the previously described benchmarks.

In Figure 13.1 we report average best fitness values against generations for all three tests. This figure clearly shows that the two GP models find solutions of similar qualities at corresponding generations for all the three studied benchmarks.

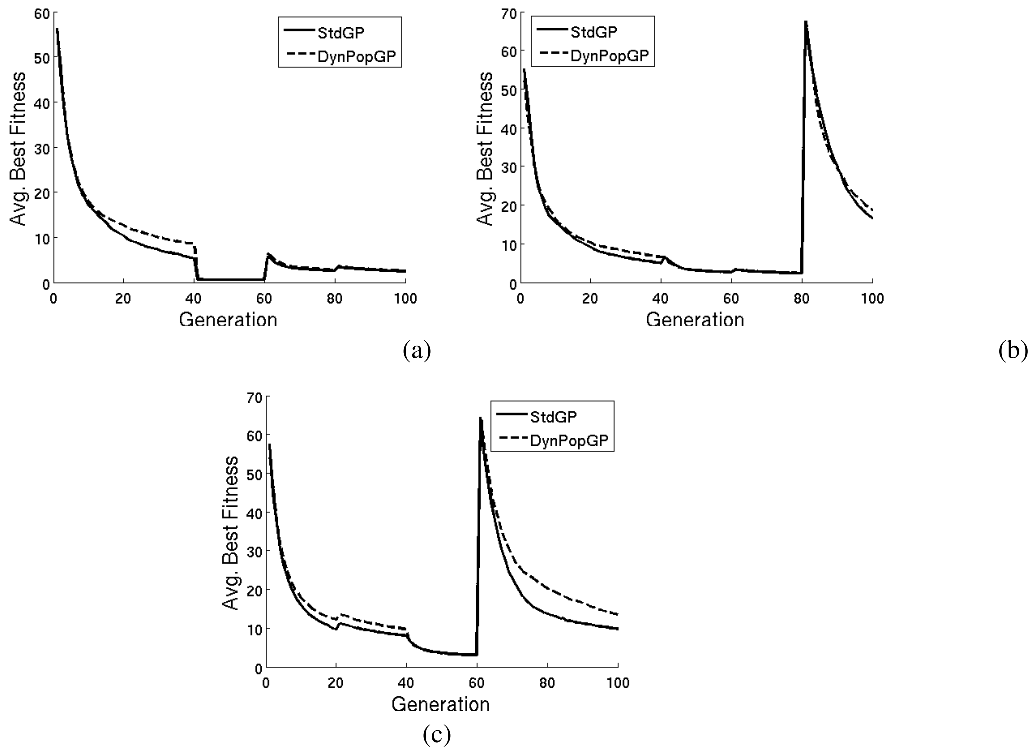

Figure 13.1: Average best fitness against generations for *StdGP* and *DynPopGP*. (a): BENCH1; (b): BENCH2; (c): BENCH3.

Seen from this perspective, the two GP models might seem equivalent. However, as reported for instance in [14], comparing the performances of two GP models against *generations* may lead to wrong conclusions. GP individuals have variable size, and thus evaluating a generation for the two models may request a very different amount of computational resources.

So, let's take a look at Figure 13.2, where we report the values of the computational effort against generations.

Figure 13.2 shows that the effort spent by *DynPopGP* is smaller than

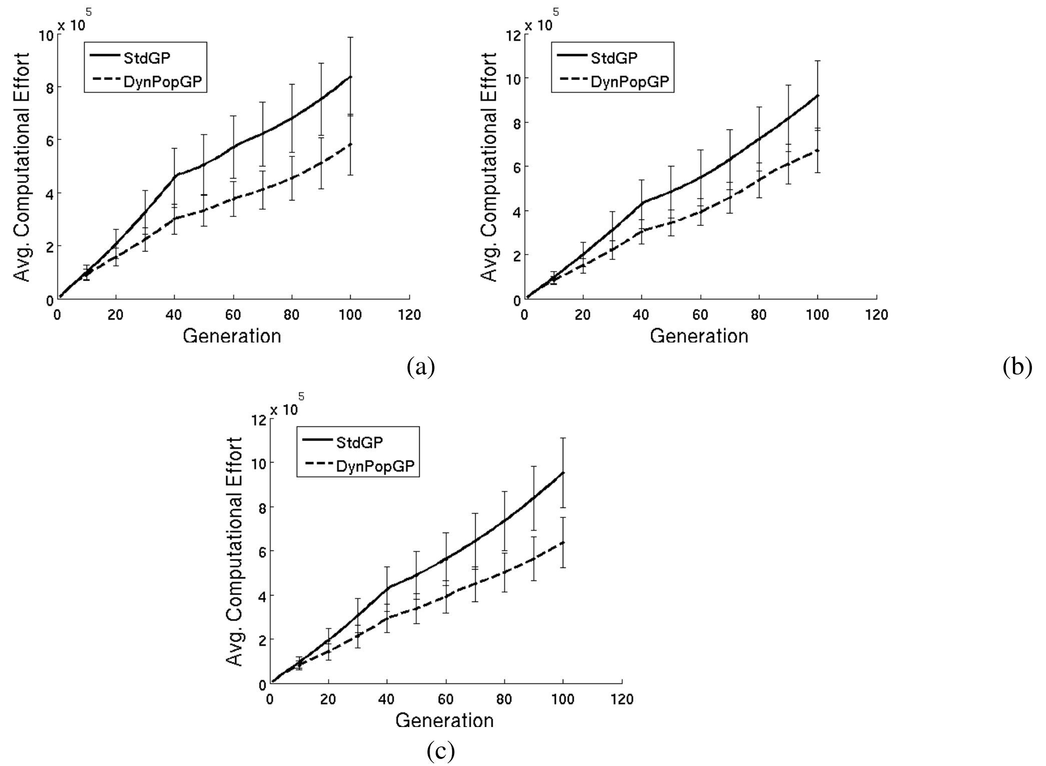

Figure 13.2: Computational effort against generations for *StdGP* and *DynPopGP*. (a): BENCH1; (b): BENCH2; (c): BENCH3;

the one spent by *StdGP* for all the three studied benchmarks. Standard deviations reported in figure as error bars seem to hint that these results are statistically significant.

Authors of [14] report results of the average best fitness against computational effort. We do the same in Figure 13.3, where it is clear that *DynPopGP* finds solutions of similar quality with a smaller computational effort than *StdGP*.

In Figure 13.4 we report the average population size at each generation. We can see that the population size of *DynPopGP* is always smaller than the one of *StdGP* for all the three studied benchmarks. Nonetheless, we can notice that the population size of *DynPopGP*, tends to grow at each *period* generations (with *period* multiple of 20), because of the change of target function. In some cases this growth begins slightly before the end of a period, probably because the particular target function had already been optimized and the stagnation phase was beginning.

Another interesting thing to remark is that, after a first phase of population shrinking, which is common in the three benchmarks, the popula-

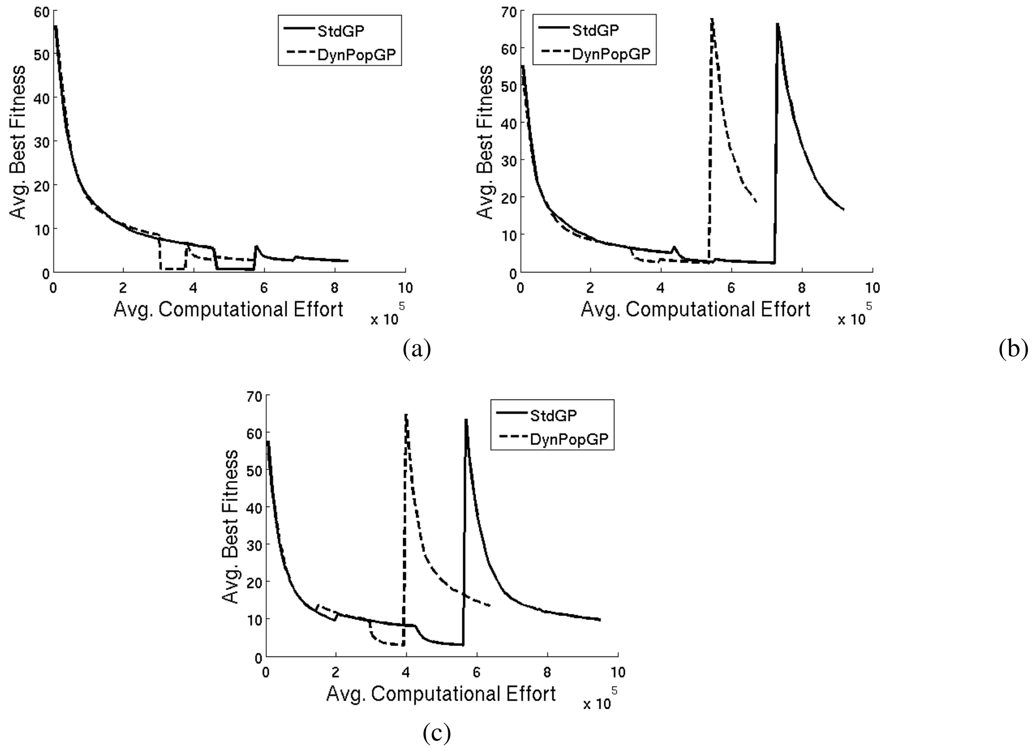

Figure 13.3: Average best fitness against computational effort for *StdGP* and *DynPopGP*. (a): BENCH1; (b): BENCH2; (c): BENCH3.

tion growth is stronger for BENCH1 (which has the more violent target modifications) than for BENCH3 (which has the less violent target modifications), while the behavior of BENCH2 is intermediary. Furthermore, it is possible to see that for BENCH1 the population size continues to grow until the end of the run, while for BENCH2 and BENCH3 there is a new phase in which the population starts shrinking once again (at about generation 80 for BENCH2 and generation 60 for BENCH3).

13.4 Conclusions

Many real-life applications are anchored in dynamic environments, where some elements of the problem domain, typically targets, change with time. Despite the importance of dynamic optimization problems, few contributions on GP for these problems have appeared until now.

This thesis is inspired by the idea that DSP GP should perform better than the fixed size population GP on these problems. The idea of using variable size populations for dynamic optimization problems is not new

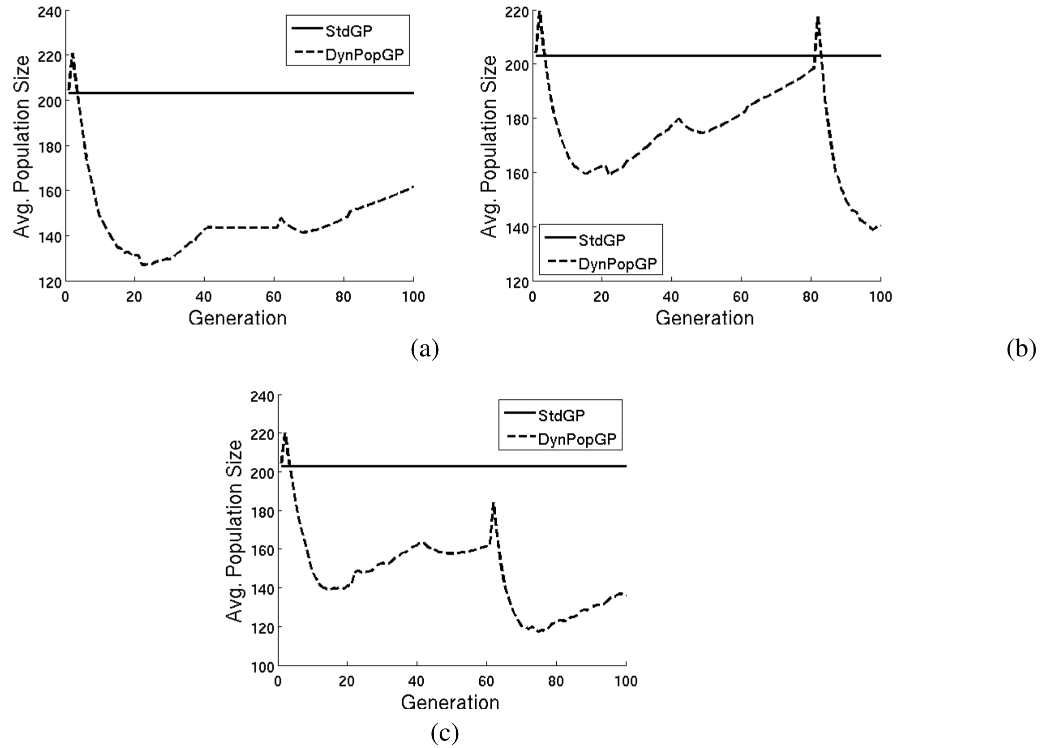

Figure 13.4: Population size against generations for *StdGP* and *DynPopGP*. (a): BENCH1; (b): BENCH2; (c): BENCH3.

in Evolutionary Computation [13], but, to the best of our knowledge, it had never been extended to GP before. In particular, we believe that a model inspired by the one presented in [29] should be suitable for this kind of problems. In fact, in that model, population shrinks while GP optimization is progressing and it grows in size when it is stagnating. Thus, it makes sense to imagine that when the target function changes, the reinjection of new genetic material in the population (due to the population size growth) should be beneficial.

13.5 Contributions

Various contributions have been made in this thesis. Let's have a closer look at the fundamental ones:

- We have motivated the fact that the GP model presented in [29], taken as it is, is not suitable for dynamic optimization problems
- We have presented a GP model that extends the one introduced

in [29] and that we candidate for suitably solving dynamic optimizaton problems. We have called that model *DynPopGP*

- We have defined a new set of benchmarks to test GP models for dynamic optimization, based on some symbolic regression problems used in [20]

- We have experimentally shown that *DynPopGP* allows GP to save computational effort compared to standard GP, while finding solutions of the same accuracy, at least for the benchmarks studied.

This work is clearly a first and preliminary step in this research track. The usefulness of GP (and in particular GP with variable size population) for dynamic optimization deserves further investigation.

In particular, GP models have to be tested on hard real-life applications, typically characterized by a large number features an few samples and the issue of generalization to out-of-sample data deserves to be investigated.

13.6 Note: on GECCO 2009

Our work, results and contributions have been resumed into a paper, by the same name of the thesis. This paper has been submitted to GECCO 09, and we're now waiting for an answer from the paper selection committee.

We would like to publish this work because of the great benefits it grants in spite of its semplicity. We're hence looking forward for the conference, that will be hold in Montreal, Canada, in June 2009.

Chapter 14

Walking towards the future

14.1 Implementation status

The actual implementation of the framework includes:

- A complete Genetic Program framework, with all the operators described before
- A multi function fitness landscape dynamic optimization problem simulator, which can emulate every kind of real DOP
- A variable size population automated control
- A complete stats and reports building suite, which outputs MatLab M-files
- A complete suit to shrink, serialize and rebuild tests on Beowulf type clusters (qsub-based)

14.2 On computational effort and best fitness

Our initial goal of gaining better prestations on computational effort has been fulfilled. We still have to improve our best fitness results - but, as I said, you can buy one currency with the other.

It should not be difficult to tune the delta pop function to keep the population size higher. This bigger pool would directly lead to better search performance, paying in the effort money we have.

14.3 Multipopulation

The next step of the project would surely be to add multipopulation management to the framework. All possible accomodation in merit have

been taken. We just need to keep track of a list of populations, and to implement the function necessary to share individuals between them and give the right population as input to the control functions.

The actual studies on multipopulation GP implementation are pretty rare. Nonetheless, they are pretty good too. We think it would be of most use in dynamic optimization environments: mantaining separated genetic pools would naturally lead to enhanced genotipic diversity, a sure ace in the sleeve on the moment the fitness function will change.

Final Thoughts

It's been a beautiful experience. It really has. I waited for two years to have a project of such proportions in which to sink, and now that I'm out on the other side all I can say is "Wow! Let's do it again!!".

It really has been fun. Hard work, very hard work, hours and hours stolen to sleep, eat and social interaction; but fun. I started from scrap, loaded my bag of the experience of some of the sharper minds on contemporary artificial intelligence, and built a robust and flexible instrument, that I hope will be of use to anyone who would like to walk this path after me. It's a great feeling.

Acknowledgements

- First of all, I must thank my parents. Per voi, che primi mi avete fatto sentire prezioso, ho poche parole e mille significati. Grazie, di cuore, di ciò che mi avete permesso di essere.
- I'd like to thank here Leonardo Vanneschi as a person. Not as the professor which taught me about evolutionary algorithms, but as the hard-working man that never denies a smile or a good word to anyone. It's been a great pleasure to work with you, a pleasure which I hope to repeat soon.
- Maria Grazia Vanola is the real backbone of DISCo, its nervous system. If everything works flawlessly, it's because of her full commitment. She teached me the difference between job and service, and that's a valuable lesson.
- The same words I can use to describe Antonio Bichiri. To take scolarship for five years in a row, you must not only fully apply, but you need the possibility to do your best to earn it. Antonio gave me this opportunity, without which I simply would not be here today.
- Paul Graham teached me lots of things, beside Lisp. Writing, economy, team management: he changed my life with just one book. He gave me the possibility to understand what I wanted.
- McCarthy is said to be the Alpha Nerd. While everyone around him thought that languages were to use hardware, he said that languages were to solve problems. Without Lisp, I don't think I could have accomplished what I did in so short time.
- Sun Tzu, Musashi, Chang Dsu Yao, Lauria, Fanfani: they can be born in Japan, China or Italy, they can be said to teach strategy, swordsmith or martial arts. But I know what they're been to me: shifu. Teaching, guidance, strength, respect: I owe them some of the best parts of me. Five years seem to have flown in all aspects but in their lessons: I learned so much from them to make five years seem a really short period. I won't forget their lessons.

A final acknowledgement must go to Rita.

She was there when I slept late and woke up early, when I worked too much, when I didn't eat and when I ate too much, when I was sick, tired, discouraged, oppressed. She was there, by my side.

You took care of me, like only a great woman could do.

And now that I've made it, the memory of your visage comes to my mind as a trophy. WE made it!

And, well, ok, I promise: no more degrees, ok? ;)

Thank you my love.

"I suppose I should learn Lisp, but it seems so foreign."
- Paul Graham, Nov 1983

Bibliography

[1] W. Banzhaf and W. B. Langdon. Some considerations on the reason of bloat. *Genetic Programming and Evolvable Machines*, 3:81–91, 2002.

[2] J. Branke. Memory enhanced evolutionary algorithms for changing optimization problems. In *Congress on Evolutionary Computation CEC99*, volume 3, pages 1875–1882. IEEE, 199.

[3] J. Branke. Evolutionary algorithms for dynamic optimization problems - a survey. Technical Report 387, Insitute AIFB, University of Karlsruhe, Feb. 1999.

[4] J. Branke. Evolutionary approaches to dynamic environments - updated survey. In *GECCO Workshop on Evolutionary Algorithms for Dynamic Optimization Problems*, pages 27–30, 2001.

[5] J. Branke. *Evolutionary Optimization in Dynamic Environments*. Kluwer, 2001.

[6] J. Branke. Evolutionary approaches to dynamic optimization problems – introduction and recent trends. In J. Branke, editor, *GECCO Workshop on Evolutionary Algorithms for Dynamic Optimization Problems*, pages 2–4, 2003.

[7] J. Branke, T. Kauler, C. Schmidt, and H. Schmeck. A multi-population approach to dynamic optimization problems. In *In Adaptive Computing in Design and Manufacturing*, pages 299–308. Springer, 2000.

[8] M. Clerc. *Particle Swarm Optimization*. ISTE, 2006.

[9] H. G. Cobb. An investigation into the use of hypermutation as an adaptive operator in genetic algorithms having continuous, time-dependent nonstationary environments. Technical Report ADA229159, Naval Research Lab, Washington DC, 1990.

[10] C. Darwin. *On the Origin of Species by Means of Natural Selection*. John Murray, 1859.

[11] D. Dasgupta and D. R. Mcgregor. Nonstationary function optimization using the structured genetic algorithm. In *Parallel Problem Solving From Nature*, pages 145–154. Elsevier, 1992.

[12] F. O. de França, F. J. V. Zuben, and L. N. de Castro. An artificial immune network for multimodal function optimization on dynamic environments. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 289–296, New York, NY, USA, 2005. ACM.

[13] C. Fernandes, V. Ramos, and A. Rosa. Varying the population size of artificial foraging swarms on time varying landscapes. In *International Conference on Artificial Neural Networks: Biological Inspirations*, volume 3696 of *LNCS*, pages 311–316. Springer, 2005.

[14] F. Fernández, M. Tomassini, and L. Vanneschi. An empirical study of multipopulation genetic programming. *Genetic Programming and Evolvable Machines*, 4(1):21–52, 2003.

[15] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[16] D. E. Goldberg and R. E. Smith. Nonstationary function optimization using genetic algorithms with dominance and diploidy. In *ICGA*, pages 59–68, 1987.

[17] J. J. Grefenstette. Genetic algorithms for changing environments. In *Parallel Problem Solving from Nature 2*, pages 137–144, 1992.

[18] J. H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, Michigan, 1975.

[19] C.-F. Huang and L. M. Rocha. Tracking extrema in dynamic environments using a coevolutionary agent-based model of genotype edition. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 545–552, New York, NY, USA, 2005. ACM.

[20] M. Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In C. Ryan *et al.*, editor, *Genetic Programming, Proceedings of the 6th European Conference, EuroGP 2003*, volume 2610 of *LNCS*, pages 71–83, Essex, 2003. Springer, Berlin, Heidelberg, New York.

[21] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proc. IEEE Int. conf. on Neural Networks*, volume 4, pages 1942–1948. IEEE Computer Society, 1995.

[22] J. R. Koza. *Genetic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.

[23] N. Mori, H. Kita, and Y. Nishikawa. Adaption to a changing environment by means of the thermodynamical genetic algorithm. In *PPSN IV: Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*, pages 513–522, London, UK, 1996. Springer-Verlag.

[24] R. Morrison and K. De Jong. A test problem generator for non-stationary environments. *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, 3:–2053 Vol. 3, 1999.

[25] K. P. Ng and K. C. Wong. A new diploid scheme and dominance change mechanism for non-stationary function optimization. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 159–166, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[26] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[27] W. Rand and R. Riolo. The problem with a self-adaptative mutation rate in some environments: a case study using the shaky ladder hyperplane-defined functions. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1493–1500, New York, NY, USA, 2005. ACM.

[28] Y. H. Shi and R. Eberhart. A modified particle swarm optimizer. In *Proc. IEEE Int. Conference on Evolutionary Computation*, pages 69–73. IEEE Computer Society, 1998.

[29] M. Tomassini, L. Vanneschi, J. Cuendet, and F. Fernández. A new technique for dynamic size populations in genetic programming. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation (CEC'04)*, pages 486–493, Portland, Oregon, USA, 2004. IEEE Press, Piscataway, NJ.

[30] S. Tsutsui, Y. Fujimoto, and A. Ghosh. Forking genetic algorithms: Gas with search space division schemes. *Evol. Comput.*, 5(1):61–80, 1997.

[31] F. Vavak, K. Jukes, and T. C. Fogarty. Learning the local search range for genetic optimisation in nonstationary environments. In

In IEEE Intl. Conf. on Evolutionary Computation ICEC'97, pages 355–360. IEEE Publishing, 1997.

[32] S. Yang. Constructing dynamic test environments for genetic algorithms based on problem difficulty. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 2, pages 1262–1269. IEEE, Piscataway NJ, USA, 2004.